Testing

The objective of this lessonis to ensure that you understand the fundamentals of testing and are capable of reading and solving specs. **This lesson is relevant**to you because good testing is one of the foundations of being a good developer.

When you finish, you should be able to:

- Explain the "red-green-refactor" loop of test-driven development.
- Identify the definitions of SyntaxError, ReferenceError, and TypeError
- Create, modify, and get to pass a suite of Mocha tests
- Use Chai to structure your tests using behavior-driven development principles.
- Use the pre- and post-test hooks provided by Mocha

All About Testing!

In your daily life you have encountered tests before - though school, work, or even through trivia, a test is a way to ensure something is correct. In your programming careers so far you've tested most of your work by hand. Testing one function at a time can be tedious, repetitive, and worst of all, it is a method vulnerable to both false positives and false negatives.

Let's talk about *automated testing* - the how, the what, and most importantly the **why**. The general idea across all testing frameworks is to allow developers to write code that would specify the behavior of a function or module or class. We've reached a point in software development where developers can now run test code against their application code and have confidence that their code will work as intended.

When you finish this reading you should be able to paraphrase the how and why we test as well as how to read automated tests without necessarily knowing the syntax.

Why do we test?

Yes, making sure the dang thing actually works is important. But beyond the obvious, why take the time to write tests?

- *To make sure the dang thing works*

  Yes, that's obvious, but dagnabit, it's important!

- *Increase flexibility & reduce fear (of code)*

  You've written a whole bunch of functionality, multiple other developers have worked on the code, you're deep into the project... And then you realize you have to refactor big chunks of it. Without automated tests, you'll be walking on eggshells, frightened of the codebase and the various landmines that are surely lying in wait.

  With tests, you can aggressively refactor with confidence. If anything breaks, you'll know. And you'll know exactly what the expectations are for the module you're refactoring, so as long as it meets the specs, you're good.

When you are writing automated tests for an application you are writing thespecficationsof how that application should behave. In the software industry automated tests are often called "specs", which is short for the word "specification".

- *Make collaboration easier*

  Complex applications are built by teams of developers. It may be that not all those developers will actually get the chance to talk to one another (they're busy, they may live in different places, some of them may have left the company, new people just joined, it's a huge project, etc.).

  Specs allow teams to have confidence that each module performs a specific task and reduces the need for expensive coordination. The specs themselves become an effective form of communication.

- *Produce documentation*

  If the tests are written well, the tests can serve as documentation for the codebase. Need to know what such and such module does? Check out the specs. This is related to easing collaboration.

How we Test

Testing frameworks vs Assertion libraries

An important distinction to understand is the difference between a *testing framework* and an *assertion library*. The job of a testing framework is to **run** tests and present them to a user. An assertion library is the backbone of any written test - it is the code that we use to **write** our tests. Assertion libraries will do the heavy lifting of comparing and verifying our code. Some testing frameworks will have built in assertion libraries, others will need you to import an assertion library to use.

Mocha

Mocha is a JavaScript *testing framework* that specializes in *running* tests and presenting them in an organized user friendly way. The `Mocha` testing framework is widely used because of it's flexibility. `Mocha` supports a whole variety of different assertion libraries and DSL interfaces for writing tests in the way the best suites the developer.

When writing tests with Mocha we will be using `Mocha`'s DSL (Domain Specific Language). A Domain Specific Language refers to a computer language specialized for a particular purpose - in `Mocha`'s case the DSL has been engineered for providing structure for writing tests. A DSL is it's own language that will usually be familiar but syntactically a little different from the languages you know. That being said you don't have to worry about memorizing every single piece of syntax for writing tests - just get a good grasp of the basics of testing and use the documentation to fill in any knowledge gaps.

You've seen what `Mocha` looks like already because all the specs for your assessments and projects so far have been written utilizing `Mocha` as the testing framework.

We'll be talking more about different assertion libraries a little later when we talk about *writing* tests.

What do we test?

So now that we talked about why we test and what we use to test...what exactly do we test?

Test the public interface

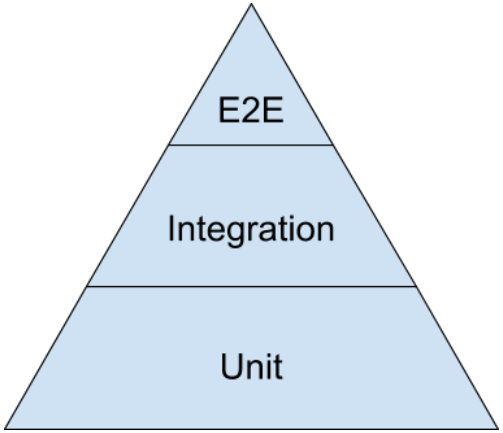
When you're trying to figure out what you should be testing, ask yourself, "What is (or will be) the public interface of the module or class I'm writing?" That is, what are the functions that the outside world will have access to and rely on?

Ideally, you'd have thorough test coverage on the entire public interface. When that's not possible, ensure that your tests cover the most important and/or complex parts of that interface - that is, the pieces that you need to make sure work as intended (and expected).

Kent Dodds has a great article on how to identify what you should be testing.

The testing pyramid

A common metaphor used to group software tests into separate levels of testing is the testing pyramid.



Let's quickly go over each level before talking about the pyramid as a whole:

- **Unit Tests**: The smallest unit of testing - used to test the smallest pieces of your application in isolation to ensure each piece works before you attempt to put those pieces together. Each unit test should focus on testing**one**thing. These are generally the fastest tests to write and run.
- **Integration Tests**: Once you have your unit tests in place you know each piece works in isolation - but what about when those pieces interact with each other? Integration tests are the next level up, they will test the interactions between two pieces of your application. Integration tests will ensure the units you've written work coherently together.
- **End-to-End (E2E) Tests**: End-to-end tests are the highest level of testing - these will test the whole of your application. End-to-end tests are the closest automated tests come to testing the an actual user experience of your application. These are generally the slowest tests to write and run.

For a real life example of how you'd utilize each of these tests imagine coding a Chess game and wanting to test it. Unit tests would be best for testing each class your wrote in isolation - like ensuring each piece's instance methods work as you expect them to before involving them with any other pieces. Next you'd write integration tests - so you'd want to ensure that each piece instance interacted correctly with the `Board` class. The final level would be End-to-End tests which would be like testing a round of chess - testing the `Board`, `Game`, and `Piece` classes all working together.

According to the testing pyramid - you want to have a solid base of a lot of Unit tests, then a medium amount of integration tests built upon that base, then finally a smaller amount of End-to-End tests. Writing tests in this way is practical for a couple of reasons. As we said before, unit tests ensure each piece of your application works in isolation - if you know each piece works then you can more easily find errors. Unit tests are also *fast*. The bigger your application gets the longer your testing suite will take to run - if all your tests are end-to-end tests your tests could be running for **hours**.

Here is a great blog from google about why they use the testing pyramid.

Reading Tests

No matter what kind of test you are encountering the most important thing about a test is that it is **readable** and **understandable**. Good tests use descriptive strings to enumerate what they are testing as well as how they are testing it.

We'll be diving more into the actual syntax of writing tests soon but for right now let's see what you can glean without knowing the syntax:

```
describe("avgValue()", function() {
  it("should return the average of an array of numbers", function() {
    assert.equal(avgValue([10, 20]), 15);
  });
});
```

So without knowing the specfic syntax we can tell a few things from the outset - the outer function has a string with the name of a function `avgValue()` which is most likely the function we will be testing. Next we see a description string `should return the average of an array of numbers`.

So even without understanding the syntax for the test above we can tell *what* we are testing - the `avgValue` function, and how we are testing it - `should return the average of an array of numbers`.

Being able to read tests is an important skill. You'll sometimes find yourself working with unfamiliar testing libraries, but if the test is well written you should be able to determine what the test is doing regardless of the syntax it uses.

Below we've re-written the above example using the Ruby language testing library RSpec:

```
describe "avg_value" do
  it "should return the average of an array of numbers" do
    expect(avg_value([10, 20])).to eq(15)
  end
end
```

Now you probably don't know Ruby - but using the same methods of deduction as we used above we can figure out what is being testing in the above snippet. The outer block mentions `avg_value` which is probably the method or function being tested and the inner block says how things are being tested - `"should return the average of an array of numbers"`. Without knowing the language, or the testing library, we can still figure out generally what what is being tested. That is the important thing about reading tests - having the patience to parse the information before you.

What you learned

We covered a high level overview of testing - the *why*, the *what* and the *how* of testing as well as the basics of how to read a test regardless of the syntax used in writing that test.

Test-Driven Development

At this point of the course you have all encountered an automated test also known as a "spec" - short for specification. In software engineering a collection of automated tests, also known as test suites, are a common way to ensure that when a piece of code is run it will perform the minimum of a specified set of behaviors. We've used the JavaScript testing framework, Mocha, up to this point to test the behavior of functions of all kinds from `myForEach` to `avgValue` to ensure each function runs as intended.

The main question we should be able to answer when writing any piece of code is: what does this code do? How should this code behave? One of the popular ways to answer this question is through a software development process called Test-driven development or TDD. TDD is a quick repetitive cycle that revolves around *first* determining what a piece of code should do and writing tests for that behavior *before actually writing any code*.

Test-driven development dictates that tests, not application code, should be written first, and then application code should only be written to pass the already written tests. When you finish this reading you should be able to identify the three steps of Test Driven Development as well as identify the advantages of using TDD to write code.

Motivations for TDD

Imagine being handed a file of 10 functions that all invoke each other and being told to add a new function to the mix and ensure all the previous functions work properly. First you'd have to figure out what each function actually did, then determine if they did what they were supposed to do. Sounds like a total pain right? A modern web application is thousands of lines of code that are worked on and maintained by teams of developers. Using TDD is one way for developers to ensure that the code written by every member of their team is testable and modular.

Here are some of the biggest motivations for why developers use test-driven development:

1. Writing tests before code ensures that the code written *works*.
    - Code written to pass specs is guaranteed to be testable.
    - Code with pre-written tests easily allows other developers to add and test new code while ensuring nothing else breaks along the way.

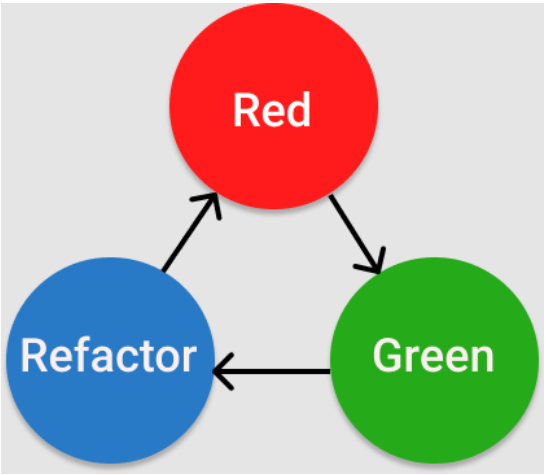
2. Only required code is written.
    - In the face of having to write tests for every piece of added functionality TDD can help reduce bloated un-needed functionality.
    - TDD and YAGNI ("you ain't gonna need it") go hand in hand!

3. TDD helps enforce code modularity.
    - A TDD developer is forced to think about their application in small, testable chunks - this ensures the developer will write each chunk to be modular and capable of individual testing.

4. Better understanding of *what* the code should be doing.
    - Writing tests for a piece of code ensures that the developer writing that code knows what the piece of code is trying to achieve.

Now that we've covered why developers would want to use TDD let's go into *how* to do TDD.

The three steps of TDD: red, green, refactor!



The Test-driven development workflow can be broken down intro three simple steps. **Red, Green, Refactor**:

1. **Red**: Write the tests and watch them fail (a failing test is red). It's important to ensure the tests initially fail so that you don't have false positives.
2. **Green**: Write the minimum amount of code to ensure the tests pass (a passing test will be green).
3. **Refactor**: Refactor the code you just wrote. Your job is not over when the tests pass! One of the most important things you do as a software developer is to ensure the code you write is easy to maintain and read.

Generally, the TDD workflow loop of Red, Green, Refactor is quick. TDD developers will write small tests ensuring each individual part of their application works properly and their code looks good before moving on - making for a short development cycle.

What you learned

TDD stands for test-driven development. TDD is a repetitive process that revolves around three steps: Red, Green, Refactor.

A Comedy of Errors in JavaScript

You know that feeling when you've just finished your perfect function then you go to run your code and: BAM! A big error is thrown? We all have felt that pain from the starting student to the experienced engineer. Runtime errors are a part of daily life when writing code. It is now time to dive into what each type of error you encounter means in order to more quickly and efficiently fix the problem that created that error.

When you finish this reading you should be able to: identify the difference between `SyntaxError`, `ReferenceError`, and `TypeError`s as well as create and `throw` new errors.

JavaScript Errors

In JavaScript the `Error` constructor function is responsible for creating different instances of `Error` objects. The `Error` object is how JavaScript deals with runtime errors and the *type* of error created and thrown will attempt to communicate *why* that error occurred.

Creating your own errors

Since the Error constructor is just a constructor function we can use it to create new `Error` object instances with the following syntax:

```
new Error([message[, fileName[, lineNumber]]])
```

As seen above you can optionally supply a `message`, `fileName` and `lineNumber` where the error occurred.

The `Error` constructor is also somewhat unique in that you can call it with or without the `new` keyword and it will return a new `Error` object:

```
const first = Error("I am an error object!");
const second = new Error("I am too an error object!");

console.log(first); // Error: I am an error object!
console.log(second); // Error: I am too an error object!
```

Let's take a look at what we can do with our newly created `Error` objects.

Throwing your own errors

Tired of JavaScript being the only one to throw errors? Well you can too! Using the keyword `throw` you can throw your own runtime errors that will stop program execution.

Let's take a look at the syntax for `throw`:

```
function giveMeNumber(num) {
  if (typeof num !== "number") {
    throw new Error("Give me a number!");
  } else {
    return "yay number!";
  }
}

console.log(giveMeNumber(1)); // prints "yay number!";
console.log(giveMeNumber("apple")); // Uncaught Error: Give me a number!
console.log(giveMeNumber(1)); // doesn't get run
```

Now as we can see in the above example throwing an error is a powerful tool that stops program execution. If we wanted to throw an error *without* stopping

program execution we can use a `try...catch` block.

Let's look at the syntax for using the `try...catch` block syntax:

```
try {
  // statements that will be attempted to here
} catch (error) {
  // if an error is thrown it will be "caught"
  // allowing the program to continue execution
  // these statements will be run and the program will continue!
}
```

We normally use `try...catch` blocks with functions that might throw an error. Let's look at an example where an error *will not* be thrown:

```
function safeDivide(a, b) {
  if (b === 0) {
    throw new Error("cannot divide by zero");
  } else {
    return a / b;
  }
}

try {
  console.log(safeDivide(30, 5)); // prints 6
} catch (error) {
  console.error(error.name + ": " + error.message);
}

console.log("hello"); // prints hello
```

Note: We can use `console.error` instead of `console.log` to make logged errors more noticeable.

Above you can see our `safeDivide` function ran as expected. Now let's see what happens when an error will be thrown and **caught** inside a `try...catch` block:

```
function safeDivide(a, b) {
  if (b === 0) {
    throw new Error("cannot divide by zero");
  } else {
    return a / b;
  }
}

try {
  console.log(safeDivide(30, 0));
} catch (error) {
  console.error(error.name + ": " + error.message); // Error: cannot divide by ze
}
// the above error will be caught allowing our program to continue!
console.log("hello"); // prints  "hello"
```

Those are the basics of creating and throwing your own errors. You can `throw` your newly created `Error` to stop program execution or use a `try...catch` block to catch your error and continue running your code. Now that we've learned how to create new errors let's go over the core errors built into JavaScript and what they signify.

Types of JavaScript errors

There are seven core errors you'll encounter in JavaScript and each type of error will try to communicate why that error occurred:

1. `SyntaxError` - represents an error in the syntax of the code.
2. `ReferenceError` - represents an error thrown when an invalid reference is made.
3. `TypeError` - represents an error when a variable or parameter is not of a valid type.

4. `RangeError` - representing an error for when a numeric variable or parameter is outside of its valid range.
5. `InternalError` - represents an error in the internal JavaScript engine.
6. `EvalError` - represents an error with the global `eval` function.
7. `URIError` - represents an error that occurs when `encodeURI()` or `decodeURI()` are passed invalid parameters.

For this reading we'll be going in depth of the three most common errors you have encountered so far: `SyntaxError`, `ReferenceError`, and `TypeError`.

SyntaxError

A SyntaxError is thrown when the JavaScript engine attempts to parse code that does not conform to the syntax of the JavaScript language. When learning the JavaScript language this error is a constant companion for any missing `}` or misspelled `function` keywords.

Let's look at a piece of code that would throw a syntax error:

```
funtion broken () { // Uncaught SyntaxError: Unexpected identifier
  console.log("I'm broke")
}
```

Another example with an extra curly brace `}`:

```
function broken () { // Uncaught SyntaxError: Unexpected identifier
  console.log("I'm broke")
}} // Uncaught SyntaxError: Unexpected token '}'
```

The examples go on and on - you can count on a `SyntaxError` to be thrown whenever you attempt to run code that is not syntactically correct JavaScript.

Important! One thing to note about Syntax Errors is that many of them can't be caught using `try` `catch` blocks.

For instance, the following code will throw a `SyntaxError` and no matter how hard you try, you can't catch it.

```
try {
    if (true { // throws "SyntaxError: Unexpected token '{'"
        console.log("SyntaxErrors are the worst!");
    }
} catch (e) {
    console.log(e);
}
```

The missing parenthesis after `true` will throw a `SyntaxError` but can't be caught by the `catch` block.

This is because this kind of `SyntaxError` happens at *compile time* not *run time*. Any errors that happen at *compile time* can't be caught using `try` `catch` blocks.

ReferenceError

Straight from the MDN docs: "The `ReferenceError` object represents an error when a non-existent variable is referenced." This is the error that you'll encounter when attempting to reference a variable that does not exist (either within your current scope or at all).

Let's take a took at some examples for the causes of this error. One common cause for this error is misspelling a variable name:

```
function callPuppy() {
  const puppy = "puppy";
```

```
  console.log(pupy);
}

callPuppy(); // ReferenceError: pupy is not defined
```

Another common cause for a thrown `ReferenceError` is attempting to access a variable that is not in scope:

```
function callPuppy() {
  const puppy = "puppy";
}

console.log(puppy); // ReferenceError: puppy is not defined
```

The aptly named `ReferenceError` will be thrown whenever you attempt to *reference* a variable that doesn't exist.

TypeError

A `TypeError` is commonly thrown for a couple of reasons:

1. When an operation cannot be performed because the operand is a value of the wrong type.
2. When you are attempting to modify a value that cannot be changed.

Let's look at a couple of examples that will each throw a `TypeError` for a different reason. Below we are attempting an operation (in this case a function call) on a value of the wrong type:

```
let dog; // Remember unassigned variables are undefined!

dog(); // TypeError: dog is not a function
```

In the above example we attempt to invoke a declared but not assigned variable (which will evaluate to `undefined`). This will cause a `TypeError` because `undefined` cannot be invoked - it is the wrong type.

Next let's look at a example of attempting to change a value that cannot be changed:

```
const puppy = "puppy";

puppy = "apple"; // TypeError: Assignment to constant variable.
```

Attempting te reassign a `const` declared variable will result in a `TypeError`. You've probably run into many other examples of `TypeError` yourself but, the most important thing to know is that a `TypeError` is thrown when you attempting to perform an operation on the wrong type of value.

Catching known errors

Now that we've covered the the names of common JavaScript errors as well as how to use a `try...catch` block we can combine these two ideas to catch specific types of errors using `instanceof`:

```
function callThatArg(arg) {
  arg(); // this will cause a TypeError because callThatArg is being passed a num
}

try {
  callThatArg(42);
  console.log("call successful"); // this line never executes
} catch (error) {
  if (error instanceof TypeError) {
    console.error(`Wrong Type: ${error.message}`); // prints: Wrong Type: arg is
  } else {
    console.error(error.message); // prints out any errors that aren't TypeErrors
```

```
  }
}

console.log("done"); // prints: done
```

What you learned

If you read an error and know *why* that error is being thrown it'll be much easier to find the cause of the problem! In this reading we went over how to create and throw new `Error`objects as well as the definitions for some of the most common types of errors: `SyntaxError`, `ReferenceError`, and `TypeError`s.

Writing Tests

For weeks you have been using one of JavaScript's most popular test frameworks, `Mocha`, to run tests that ensure a function you've written works as expected. It's time to dive deeper into **how** to write our own tests using `Mocha` as our test framework coupled with Assertion libraries such as the built-in `Assert` module of Node or the `Chai` library.

For the rest of the readings in this section we will be covering how to write tests. These readings will be done in the style of a code-along demo so make sure you follow these in order. When you have finished the next series of reading you should know how to:

- properly format and denote your mocha tests using `describe`, `context` and `it` blocks
- write tests for individual functions as well as writing tests for class instance and static methods
- test that functions will throw certain errors
- use `chai-spies` to test how many times a function has been called
- recognize and utilize the Mocha hooks: `before`, `beforeEach`, `after`, and `afterEach`

In this reading we'll be covering:

- the file structure of testing with `Mocha`
- testing with `Mocha` and Node's built-in `Assert` module

Part Zero: Testing file structure

We find that reading about testing is best understood when you can play around within the functions being tested so for that reason this reading will be in the style of a code along demo. We started this reading by created a directory called `testing-demo` where all the code within this reading will be written.

Let's start off with how to write tests for a basic function. Say we've been handed a directory with a function to test `problems/reverse-string.js`. Below is the named function we'll be testing, `reverseString`, which will intake a string argument and then reverse it:

```
// in testing-demo/problems/reverse-string.js

const reverseString = str => {
  // throws a specific error unless the the incoming arg is a string
  if (typeof str !== "string") {
    throw new TypeError("this function only accepts string args");
  }

  return str
    .split("")
    .reverse()
    .join("");
};

// note this function is being exported!
module.exports = reverseString;
```

How would you go about testing the above function? Let's start by setting up our file system correctly. Whenever you are running tests with `Mocha` the important thing to know is that the `Mocha` CLI will automatically be looking for a directory named `test`.

The created `test` directory's file structure should mirror that of the files you intend to test - with each test file appending the word "spec" to the end of the file name. So for the above example we would create `test/reverse-string-spec.js` which should be on the same level as the `problems` directory.

Our file structure should look like this:

```
testing-demo
  └──
  problems
    └── reverse-string.js
  test
    └── reverse-string-spec.js
```

Take a moment to ensure your file structure looks like the one above and that you've copied and pasted the `reverseString` function into the `reverse-string.js` file. Now that we've ensured our file structure is correct let's write some tests!

Part One: Writing tests with Mocha and Assert

The first step in any testing workflow is initializing our test file. Now let's make a clear distinction before moving forward - `Mocha` is a test framework that specializes in *running* tests and presenting them in an organized user friendly way. The code responsible for actually verifying things for us will come from using an *Assertion Library*. Assertion Libraries will do the heavy lifting of comparing and verifying code while `Mocha` will run those tests and then present them to us.

The tests we'll be writing for this next section will use Node's built-in `Assert` module as our Assertion Library.

So inside of `test/reverse-string-spec.js` at the top of the file we will require the `assert` module and the function we intend to test:

```
const assert = require("assert");
// this is a relative path to the function's location
const reverseString = require("../problems/reverse-string.js");
```

Take a moment to open up the Mocha documentation - it will come in handy as a reference for the syntax we'll be using. The `Mocha` DSL (Domain Specific Language) comes with a few different interfaces or "flavors" of their DSL for our purposes we'll be structuring our tests using the BDD interface.

The `describe` function is an organizational function that accepts a descriptive string and a callback. We'll use the `describe` function to **describe** what we will be testing - in this case the `reverseString` function:

```
// test/reverse-string-spec.js

const assert = require("assert");
const reverseString = require("../problems/reverse-string.js");

describe("reverseString()", function() {});
```

The callback handed to the `describe` function will be where we insert our actual tests. We can now use the `it` function - the `it` function is an organizational function we will use to wrap around each test we write. The `it` function accepts a descriptive string and callback to set up our test:

```
describe ('reverseString()', function () {
  it('should reverse the input string', function () {
    // a test will go here!
  })
}
```

The code written above will serve as a great template for future tests we wish to write. Finally, we can insert the actual test we intend to write within the callback handed to the `it` function. We'll use the assert.strictEqual function which allows you to compare one value with another value. We'll use `assert.strictEqual` to compare two strings - one from our function's result and our expected result which we will we define ourselves:

```
// remember we required the assert module at the top of this file
describe("reverseString()", function() {
  it("should reverse the input string", function() {
    let test = reverseString("hello");
    let result = "olleh";
    // the line below is where the actual test is!
    assert.strictEqual(test, result);
  });
});
```

Now if we run `mocha` in the upper most `testing-demo` directory we will see:

```
reverseString()
  ✓ should reverse the input string


1 passing (5ms)
```

We now have a working spec! Take notice of how `Mocha` structures its response in exactly the way we nested our test. The outer `describe` function's message of `reverseString()` is on the upper level and the inner `it` function's message of `should reverse the input string` is nested within.

Strictly speaking we aren't required to nest our `it` functions within `describe` functions but it is best practice to do so. As you can see yourself - it will make your tests a lot easier to read!

Let's add one more spec for `reverseString`, we'll do this by adding another `it` function within the `describe` callback:

```
describe("reverseString()", function() {
  it("should reverse the input string", function() {
    let test = reverseString("hello");
    let result = "olleh";

    assert.strictEqual(test, result);
  });

  it("should reverse the input string and output the same capitalization", functi
    let test = reverseString("Apple");
    let result = "elppA";

    assert.strictEqual(test, result);
  });
});
```

Running the `mocha` command again will return:

```
reverseString()
  ✓ should reverse the input string
  ✓ should reverse the input string and output the same capitalization


2 passing (11ms)
```

Looking good so far - head to the next reading to learn how to test errors.

Writing Tests

In this reading we'll be covering:

- how to test errors using `Mocha` and `Assert`
- test organization using `context` functions

Part Two: Testing errors

Let's jump right in where we left off! We've written a couple of nice *unit tests* - ensuring that this function works in isolation by testing the input we provided matches the expected output. One aspect of this function is not yet being tested - the error thrown when the argument is not of type `String`:

```
// str is the passed in parameter
if (typeof str !== "string") {
  throw new TypeError("this function only accepts string args");
}
```

Organizing tests

Now the above error actually sets up two *different* scenarios - one where the incoming argument is a string and the second where the incoming argument isn't a string and an error is thrown. We can denote these two different states by adding an additional level of organizational nesting to our tests. You can nest `describe` function callbacks arbitrarily deep - but this quickly becomes unreadable. When nesting, we make use of the `context` function, which is an alias for the `describe` function - the `context` function denotes that we are setting up the **context** for a particular set of tests.

Let's refactor our tests from before with some `context` functions before moving on:

```
describe("reverseString()", function() {
  context("given a string argument", function() {
    it("should reverse the given string", function() {
      let test = reverseString("hello");
      let result = "olleh";

      assert.strictEqual(test, result);
    });

    it("should reverse the given string and output the same capitalization", func
      let test = reverseString("Apple");
      let result = "elppA";
      assert.strictEqual(test, result);
    });
  });

  context("given an argument that is not a string", function() {});
});
```

Running the above test will give us this readable output:

```
reverseString()
  given a string argument
    ✓ should reverse the given string
    ✓ should reverse the given string and output the same capitalization
  given an argument that is not a string

2 passing (11ms)
```

Testing errors

Nice now that we have our `context` functions in place we can work on our second scenario where the incoming argument is *not* a string. When using an assertion library like Node's built in Assert we will have access to many functions that will allow us the flexibility to test all kinds of things. For testing errors using Node's built in `Assert` module we can use the assert.throws function.

Now we'll setup up our `it` function within the `context` function we setup above:

```
context("given an argument that is not a string", function() {
  it("should throw a TypeError when given an argument that is not a string", func
    assert.throws();
  });
});
```

The `assert.throws` function works different from the `assert.strictEqual` function in that it does not compare the return value of a function, but it attempts to invoke a function in order to verify that it will throw a particular error. The `assert.throws` function accepts a function as the first argument, then the error that should be thrown as the second argument with an optional error message as our third argument.

So following that logic, we can test the `TypeError` error thrown by `reverseString` with something like this:

```
context("given an argument that is not a string", function() {
  it("should throw a TypeError when given an argument that is not a string", func
    assert.throws(reverseString(3), TypeError);
  });
});
```

However, when we run the `mocha` command we will get:

```
reverseString()
#  etc.
  given an argument that is not a string
    1) should throw a TypeError when given an argument that is not a string


2 passing (11ms)
1 failing

1) reverseString()
    given an argument that is not a string should throw a TypeError when given ar
    argument that is not a string:
    TypeError: this function only accepts string args
```

We are failing the above spec because we passed the invoked version of the `reverseString` function with a number argument - which as we know will throw a `TypeError` and halt program execution. This is a common mistake made everyday by developers when writing tests. We can get around this by wrapping our error expecting function within another function. This will ensure we can still invoke the `reverseString` function with an argument but not throw the error until `assert.throws` is ready to catch it.

We can also add the explicit error message that `reverseString` throws to make our spec as specific as possible:

```
context("given an argument that is not a string", function() {
  it("should throw a TypeError when given an argument that is not a string", func
    assert.throws(
      function() {
        reverseString(3);
      },
      TypeError,
      "this function only accepts string args"
    );
  });
});
```

Now when we run `mocha` we will see:

```
reverseString()
  given a string argument
    ✓ should reverse the given string
    ✓ should reverse the given string and output the same capitalization
  given an argument that is not a string
    ✓ should throw a TypeError when given an argument that is not a string


3 passing (13ms)
```

Awesome! So we've covered writing unit tests using `describe`, `context`, and `it` functions for organization. We have also covered how to test for equality and thrown errors using Node's built-in assertion library, `Assert`.

Head to the next reading to learn about how to test classes using `Mocha` and another assertion library named `Chai`.

Writing Tests

In this reading we'll be covering:

- how to test classes using `Mocha` and `Chai`

Part Three: Testing classes using Mocha and Chai

Let's expand our knowledge of testing syntax by testing some classes! In order to fully test a class, we'll be looking to test that class's instance and static methods. Create a new file in the `problems` folder - `dog.js`. We'll use the following code for the rest of our tests so make sure to copy it over:

```
// testing-demo/problems/dog.js

class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    return `${this.name} is barking`;
  }

  chainChaseTail(num) {
    if (typeof num !== "number") {
      throw new TypeError("please only use numbers for this function");
    }
    for (let i = 0; i < num; i++) {
      this.chaseTail();
    }
  }

  chaseTail() {
    console.log(`${this.name} is chasing their tail`);
  }

  static cleanDogs(dogs) {
    let cleanDogs = [];
    dogs.forEach(dog => {
      let dogStr = `I cleaned ${dog.name}'s paws.`;
      cleanDogs.push(dogStr);
    });
    return cleanDogs;
  }
}

// ensure to export our class!
module.exports = Dog;
```

To test this class we'll create a new file in our `test` directory - `dog-spec.js` so your file structure should now look like this:

```
testing-demo
  └──
  problems
    └── reverse-string.js
    └── dog.js
  test
    └── reverse-string-spec.js
    └── dog-spec.js
```

Let's now set up our `dog-spec.js` file. For this example we'll get experience using another assertion library named Chai. As you'll soon see, the Chai library comes with a lot more built-in more functionality than Node's `Assert` module.

Now since `Chai` is another external library we'll need to import it in order to use it. We need to run a few commands to first create a `package.json` and then we can import the `chai` library. Start off by running `npm init --y` in the top

level directory (`testing-demo`) to create a `package.json` file. After that is finished you can import the `Chai` library by running `npm install chai`.

Here is what that will look like in your terminal:

```
~ testing-demo $ npm init --y
Wrote to /testing-demo/problems/package.json:

{
  "name": "testing-demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

~ testing-demo $ npm install chai
```

Now that we've installed `Chai` we can set up our test file. Create a new file in the `test` folder named `dog-spec.js` We'll require the `expect` module from Chai for our assertions, import our `Dog` class, and set up our outer `describe` function for testing the `Dog` class:

```
// testing-demo/test/dog-spec.js

// set up chai
const chai = require("chai");
const expect = chai.expect;

// don't forget to import the class you are testing!
const Dog = require("../problems/dog.js");

// our outer describe for the whole Dog class
describe("Dog", function() {});
```

So the first thing we'll generally want to test on classes is their `constructor` functions - we need to make sure new instances have the correct properties and that those properties are being set properly before we can test anything else. For the `Dog` class it looks like a name is accepted on instantiation, so let's test that!

We'll start with a nested describe function within our outer `Dog` describe function:

```
describe("Dog", function() {
  describe("Dog Constructor Function", function() {
    it('should have a "name" property', function() {});
  });
});
```

Now we are using a different assertion library so we'll be working with some different syntax. Open up the Chai Expect documentation for reference, we won't be going into tons of detail into every function we use with Chai because Chai allows for a lot of smaller chainable functions and we know you have lives outide this reading.

The nice thing about Chai is that the chainable functions available will often read like English. Check out the right column of this handy Chai cheatsheet for a quick and easy reference on chainable functions.

We'll start our first spec off by using the property matcher to ensure that a newly instantiated object has a specified property:

```
describe("Dog", function() {
  describe("Dog Constructor Function", function() {
    it('should have a "name" property', function() {
      const layla = new Dog("Layla");
      // all our of chai tests will begin with the expect function
      // .to and .have are Chai chainable functions
      // .property is the matcher we are using
      expect(layla).to.have.property("name");
    });
  });
});
```

Now to test our new spec we can run just the Dog class specs by running `mocha test/dog-spec.js` from our top level directory. Running that command we'll see:

```
Dog
  Dog Constructor Function
    ✓ should have a "name" property


1 passing (8ms)
```

Nice! We tested that the name property exists on a new dog instance. Next, we can make sure our name is set properly with another test:

```
describe("Dog Constructor Function", function() {
  it('should have a "name" property', function() {
    const layla = new Dog("Layla");
    expect(layla).to.have.property("name");
  });

  it('should set the "name" property when a new dog is created', function() {
    const layla = new Dog("Layla");
    // we are using the eql function to compare the value of layla.name
    // with the provided string
    expect(layla.name).to.eql("Layla");
  });
});
```

Running the above using `mocha` we'll see both of our specs passing! Now take extra note of the fact that we are defining the same variable twice using `const` within the above `it` callbacks. This is important to note because it underlines the fact that each of the unit tests you write will have their own scope - meaning that they are each independent of the specs that came before or after them.

Head to the next reading to refactor some of the code we just wrote using `Mocha` hooks!

Writing Tests

This will be the final demo in our writing tests series! In this reading we'll be covering:

- Using `Mocha` Hooks to DRY up testing
- Using `Chai Spies` to "spy" on functions to see how many times they've been invoked

Part Four: Mocha Hooks and Chai Spies

Let's jump right back in. We've written some nice unit tests up to this point:

```
describe("Dog Constructor Function", function() {
  it('should have a "name" property', function() {
    const layla = new Dog("Layla");
    expect(layla).to.have.property("name");
  });

  it('should set the "name" property when a new dog is created', function() {
    const layla = new Dog("Layla");
    // we are using the eql function to compare the value of layla.name
    // with the provided string
    expect(layla.name).to.eql("Layla");
  });
});
```

This is how unit tests are supposed to work, buuuut it will be annoying over time if we have to define a new `Dog` instance in *every single spec*. `Mocha` has some built in functionality to help us with this problem though: Mocha Hooks!

Introducing Mocha Hooks

Mocha Hooks give you a convenient way to do set up prior to running a related group of specs or to do some clean up after running those specs. Using hooks helps to keep your testing code DRY so you don't unnecessarily repeat set up and clean up code within each test.

Mocha Hooks have very descriptive function names and two levels of granularity - before/after each block of tests *or* before/after each test:

1. the hooks `before` and `after` will be invoked either before or after the block of tests is run (depending on which function is used)
2. the hooks `beforeEach` and `afterEach` will be invoked either before or after **each** test (depending on which function is used)

Let's look at a simple example that uses each of the available hooks to log a message to the console. Two placeholder tests are also defined to help demonstrate the differences between the available hooks:

```
const assert = require('assert');

describe('Hooks demo', () => {
  before(() => {
    console.log('Before hook...');
  });

  beforeEach(() => {
    console.log('Before each hook...');
  });

  afterEach(() => {
    console.log('After each hook...');
  });

  after(() => {
    console.log('After hook...');
  });
```

```
  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before hook...
Before each hook...
    ✓ Placeholder one
After each hook...
Before each hook...
    ✓ Placeholder two
After each hook...
After hook...


  2 passing (5ms)
```

Notice that the `before` and `after` hooks only ran once while the `beforeEach` and `afterEach` hooks each ran once per test.

Hooks are defined within a `describe` or `context` function. While hooks can be defined before, after, or interspersed with your tests, keeping all of your hooks together (before or after your tests) will help others to read and understand your code.

Defining hooks out of their logical order has no effect on when they're ran. Consider the following example that defines an `afterHook` before a `beforeEach` hook:

```
const assert = require('assert');

describe('Hooks demo', () => {
  afterEach(() => {
    console.log('After each hook...');
  });

  beforeEach(() => {
    console.log('Before each hook...');
  });

  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before each hook...
    ✓ Placeholder one
After each hook...
Before each hook...
    ✓ Placeholder two
After each hook...


  2 passing (6ms)
```

The order of your hooks only matters when you define multiple hooks of the same type. When a hook type is defined more than once, they'll be ran in the order that they're defined in:

```
const assert = require('assert');

describe('Hooks demo', () => {
  beforeEach(() => {
    console.log('Before each hook #1...');
  });

  beforeEach(() => {
    console.log('Before each hook #2...');
  });

  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before each hook #1...
Before each hook #2...
    ✓ Placeholder one
Before each hook #1...
Before each hook #2...
    ✓ Placeholder two


  2 passing (5ms)
```

You can also define hooks within nested `describe` or `context` functions:

```
const assert = require('assert');

describe('Hooks demo', () => {
  before(() => {
    console.log('Before hook...');
  });

  beforeEach(() => {
    console.log('Before each hook...');
  });

  afterEach(() => {
    console.log('After each hook...');
  });

  after(() => {
    console.log('After hook...');
  });

  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });

  describe('nested tests', () => {
    before(() => {
      console.log('Nested before hook...');
    });

    beforeEach(() => {
      console.log('Nested before each hook...');
    });

    afterEach(() => {
      console.log('Nested after each hook...');
    });

    after(() => {
      console.log('Nested after hook...');
```

```
    });

    it('Placeholder one', () => {
      assert.equal(true, true);
    });

    it('Placeholder two', () => {
      assert.equal(true, true);
    });
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before hook...
Before each hook...
    ✓ Placeholder one
After each hook...
Before each hook...
    ✓ Placeholder two
After each hook...
    nested tests
Nested before hook...
Before each hook...
Nested before each hook...
      ✓ Placeholder one
Nested after each hook...
After each hook...
Before each hook...
Nested before each hook...
      ✓ Placeholder two
Nested after each hook...
After each hook...
Nested after hook...
After hook...


  4 passing (7ms)
```

Notice that the `before` and `after` hooks defined in the top-level `describe` function run only once while the `beforeEach` and `afterEach` hooks run before and after (respectively) for each of the tests defined in the top-level `describe` function *and* for each of the tests defined in the nested `describe` function.

While the need to define nested hooks won't come up very often (especially when you're just starting out with unit testing), it is very helpful to be able to define a `beforeEach` hook in a top-level `describe` function that will run before every test in that block and before every test within nested `describe` or `context` functions (you'll do exactly that in just a bit).

You can also optionally pass a description for a hook or a named function:

```
beforeEach('My hook description', () => {
  console.log('Before each hook...');
});

beforeEach(function myHookName() {
  console.log('Before each hook...');
});
```

If an error occurs with executing the hook, the hook description or function name will display in the console along with the error information to assist with debugging.

Using the `beforeEach` Mocha Hook

Let's go back to our spec and see how we can use hooks to DRY up our code. Here's where we left off:

```
describe("Dog Constructor Function", function() {
  it('should have a "name" property', function() {
    const layla = new Dog("Layla");
    expect(layla).to.have.property("name");
  });

  it('should set the "name" property when a new dog is created', function() {
    const layla = new Dog("Layla");
    // we are using the eql function to compare the value of layla.name
    // with the provided string
    expect(layla.name).to.eql("Layla");
  });
});
```

Let's refactor our code to use a `beforeEach` hook to assign the value of our new dog instance:

```
describe("Dog", function() {
  // we'll declare our variable here to ensure it's available within the scope
  // of all the specs below
  let layla;

  // now for each test below we'll create a new instance to ensure each of our
  // dog instances is exactly the same
  beforeEach("set up a dog instance", function() {
    layla = new Dog("Layla");
  });

  describe("Dog Constructor Function", function() {
    it('should have a "name" property', function() {
      expect(layla).to.have.property("name");
    });

    it('should set the "name" property when a new dog is created', function() {
      expect(layla.name).to.eql("Layla");
    });
  });
});
```

Now let's write a test from the next method on the `Dog` class: `Dog.prototype.bark()`. For testing classes we'll create a new `describe` function to test each individual method. We'll now write our unit test inside taking advantage of our `beforeEach` hook:

```
describe("Dog", function() {
  let layla;

  beforeEach("set up a dog instance", function() {
    layla = new Dog("Layla");
  });

  // etc, etc.

  describe("prototype.bark()", function() {
    it("should return a string with the name of the dog barking", function() {
      expect(layla.bark()).to.eql("Layla is barking");
    });
  });
});
```

Not only are we avoiding repeating our setup code within each test but we've improved the readability of our code by making it more self-descriptive. The code that runs before each test is literally contained with a hook named `beforeEach`!

The `after` and `afterEach` hooks are generally used less often than the `before` and `beforeEach` hooks. Most of the time, it's preferable to avoid using the `after` and `afterEach` hooks to perform clean up tasks after your tests. Instead, simply use the `before` and `beforeEach` hooks to create a clean starting point for each of your tests. Doing this will ensure that your tests run in a consistent, predictable manner.

Using Chai Spies

Sweet - let's now look to the next method on the `Dog.prototype`- `Dog.prototype.chainChaseTail`. This instance method intakes a number (num) and will then invoke the `Dog.prototype.chaseTail` function `num` number of times. The `chaseTail` function will just `console.log` a string - meaning that we have no function output to test. The `Dog.prototype.chainChaseTail` function will additionally throw a `TypeError` if the incoming argument is not a number.

We'll start by setting up our outer `describe` block for the `prototype.chainChaseTail` method. Next we'll add two `context` functions for our two contexts - valid **or** invalid parameters:

`context` is just an alias for `describe` it's just another way to make your tests more understandable and readable, in this case we are testing our method with different parameters, and thus in different contexts. (not to be confused with "context" in the javascript sense of the value of `this`)

```
describe("prototype.chainChaseTail()", function() {
  context("with an invalid parameter", function() {});
  context("with a valid number parameter", function() {});
});
```

We'll start by writing our test for when the method is invoked with invalid parameters. To do this we'll use Chai's `throw` method ensuring to wrap our error throwing function in another function:

```
context("with an invalid parameter", function() {
  it("should throw a TypeError when given an argument that is not a number", func
    expect(() => layla.chainChaseTail("3")).to.throw(TypeError);
  });
});
```

Note here we are passing the literal string `"3"` not the number 3.

Nice, now we can concentrate on our other context with a valid parameter - and how to go about testing this function. In order to test `chainChaseTail` properly we'll need to see *how many times* the `chaseTail` method is invoked. Which means we'll need to import another library that will add extra functionality to `Chai`. We'll import the Chai Spies library using `npm install chai-spies` in our top level directory.

Now we'll insert a few lines of code to the top of file to set up our shiny new Chai Spies:

```
// top of dog-spec.js
const chai = require("chai");
const expect = chai.expect;
const spies = require("chai-spies");
chai.use(spies);
```

We now have access to the `chai-spies` module in our tests. The `Chai Spies` library provides a lot of added functionality including the ability to determine if a function has been called and how many times that function has been called.

So let's get started spying! We'll setup our `it` function with an appropriate string:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {});
});
```

Now in order to spy on a function we first need to tell Chai which function we'd like to spy on using the `chai.spy.on` method. In this case we'd like to spy on the instance of a Dog that will be invoking the `chainChaseTail` method to determine how many times the `chaseTail` method is then invoked.

So we will set up our spy on the dog instance in question, as well as tell our chai spy which method to keep track of:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {
    // the first argument will be the instance we are spying on
    // the second argument will be the method we want to keep track of
    const chaseTailSpy = chai.spy.on(layla, "chaseTail");
  });
});
```

Now that our spy is set up we now need make sure our dog instance will actually call the `chainChaseTail` function! Otherwise our spy won't have anything to spy on:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {
    const chaseTailSpy = chai.spy.on(layla, "chaseTail");
    // we need to invoke chainChaseTail because that is the method that
    // will invoke chaseTail which is the method we are spying on
    layla.chainChaseTail(3);
  });
});
```

Finally, we need to add our actual test - otherwise this is all for naught! Chai has some really nice chaining methods when it comes to checking how many times a function has been invoked. Here we'll use the method chain of `expect(func).to.have.been.called.exactly(n)` to test that the method we are spying on - `chaseTail` was invoked a certain number of times:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {
    const chaseTailSpy = chai.spy.on(layla, "chaseTail");
    layla.chainChaseTail(3);
    // below is our actual test to see how many times our spy was invoked
    expect(chaseTailSpy).to.have.been.called.exactly(3);
  });
});
```

Testing static methods on classes

Sweet! We are almost done testing this class - just one more method to go. We'll now work on testing the class method `Dog.cleanDogs`. To denote that this is a class method, not an instance method, our `describe` string will not use the word `prototype`:

```
describe("cleanDogs()", function() {
  it("should return an array of each cleaned dog string", function() {});
});
```

Now the `Dog.cleanDogs` class method will intake an array of dogs and output an array where each element is a string noting that the passed in dog instance's paws are now clean. In order to properly test this function we'll probably want an array of more than one dog instance. Let's create a new dog and pass an array of two dog instances to the `Dog.cleanDogs` method:

```
describe("cleanDogs()", function() {
  it("should return an array of each cleaned dog string", function() {
    const zoey = new Dog("Zoey");
    let cleanDogsArray = Dog.cleanDogs([layla, zoey]);
  });
});
```

Then we'll create a variable for our expected output and compare the output we received from `Dog.cleanDogs`:

```
describe("cleanDogs()", function() {
  it("should return an array of each cleaned dog string", function() {
    const zoey = new Dog("Zoey");
    let cleanDogsArray = Dog.cleanDogs([layla, zoey]);
    let result = ["I cleaned Layla's paws.", "I cleaned Zoey's paws."];
    expect(cleanDogsArray).to.eql(result);
  });
});
```

Awesome! We have fully testing the `Dog`class's methods and learned a lot about testing along the way.

Here is our full testing file so you can ensure you got everything:

```
const chai = require("chai");
const expect = chai.expect;
const spies = require("chai-spies");
chai.use(spies);

// this is a relative path to the function location
const Dog = require("../problems/dog.js");

describe("Dog", function() {
  let layla;

  beforeEach("set up a dog instance", function() {
    layla = new Dog("Layla");
  });

  describe("Dog Constructor Function", function() {
    it('should have a "name" property', () => {
      expect(layla).to.have.property("name");
    });

    it('should set the "name" property when a new dog is created', () => {
      expect(layla.name).to.eql("Layla");
    });
  });

  describe("prototype.bark()", function() {
    it("should return a string with the name of the dog barking", () => {
      expect(layla.bark()).to.eql("Layla is barking");
    });
  });

  describe("prototype.chainChaseTail()", function() {
    context("with a valid number parameter", function() {
      it("should call the chaseTail method n times", function() {
        const chaseTailSpy = chai.spy.on(layla, "chaseTail");
        layla.chainChaseTail(3);

        expect(chaseTailSpy).to.have.been.called.exactly(3);
      });
    });

    context("with an invalid parameter", function() {
      it("should throw a TypeError when given an argument that is not a number",
        expect(() => layla.chainChaseTail("3")).to.throw(TypeError);
      });
    });
  });

  describe("cleanDogs()", function() {
    it("should return an array of each cleaned dog string", function() {
      const zoey = new Dog("Zoey");
      let cleanDogsArray = Dog.cleanDogs([layla, zoey]);

      let result = ["I cleaned Layla's paws.", "I cleaned Zoey's paws."];
      expect(cleanDogsArray).to.eql(result);
    });
  });
});
```

What you learned

In the upcoming project we'll be covering a lot more Chai syntax - but don't worry about memorizing this syntax! The point we are trying to make is that in the future you'll be using a variety of software testing frameworks and assertion libraries - the most important things are to know the basics of how to structure tests as well as being able to read and parse documentation to write tests.

In this series of readings we covered the basics of how to:

- properly format and denote your mocha tests using `describe`, `context` and `it` blocks
- write tests for individual functions as well as writing tests for classes
- use `chai-spies` to test how many times a function has been called
- test that functions will throw certain errors
- recognize and utilize the Mocha hooks: `before`, `beforeEach`, `after`, and `afterEach`