

17

TESTING YOUR CODE

OVERVIEW

By the end of this chapter, you will be able to demonstrate the **test-driven development (TDD)** methodology; identify different types of testing; write and execute testable code; develop an environment for unit testing; integrate testing into your build scripts; and write custom assertions and mock complex and effectual functions.

In this chapter, we will learn how to write bug-free and stable JavaScript code by writing test cases.

INTRODUCTION

In an earlier chapter, we looked at functional programming, including the differences between pure and effectual functions, and why effectual-free programming is so important. Throughout this course, an emphasis has been placed on creating code that is functionally oriented while utilizing JavaScript's built-in features to create code that can be reasoned with and, where appropriate, implements declarative abstractions. This chapter takes functional programming a step further by solidifying your understanding of the benefits of **functional programming** using tests.

In the early days of JavaScript programming, its implementation within the browser was often sparse. Coding within HTML pages often required ensuring that your page functioned well in both Netscape Navigator and Internet Explorer. At that time, the scripting language of choice for Internet Explorer was VBScript, which meant developers often needed to provide two script blocks: one for VBScript and one for JavaScript. Since browser scripts were much simpler, developers were typically satisfied with testing their applications physically by loading and using their web pages in each of the major browser types.

In more recent times, internet browsers have strived to provide JavaScript support that is universal, exposing functionality that attempts to execute identically in each browser and between browser versions. Even today, browsers are not 100% compatible with one another, but the extent of any differences is slowly reducing, allowing developers to create a near majority of their code for a single browser and then make minor adjustments for each additional browser flavor prior to the launch of their web application.

Since browser-based applications are getting larger and larger, with the required download of JavaScript files sometimes being several megabytes in size, ensuring that your application is free of bugs requires a validated collection of tactics, some of which may be provided by third-party libraries and tools and some of which may be implemented by yourself as a methodology of approach. One such methodology is known as **test-driven development (TDD)** and will be discussed throughout this chapter.

WHY IS TESTING IMPORTANT?

Testing, in a programming context, is primarily to do with using scripts to validate the functionality of an application. Since applications are written in code and are rigidly defined unless modified, it becomes advantageous to write your tests in code, thereby leveraging the following advantages:

- You can test applications at a greatly increased speed and frequency.
- You can automate the execution of tests to occur at important events in your development cycle.
- You can ensure that all features are tested extensively.
- You can test across numerous environments simultaneously.
- You can automate human interaction and service communication permutations.
- You can ensure that builds do not deploy unless **all** tests pass successfully, thereby reducing production downtime.
- You can calculate test coverage, ensuring that sufficient tests exist to prove the viability of new features.
- You can verify that new features do not break old functionality.

These days, many large browser-based applications, such as **single-page applications (SPAs)**, or even design-heavy websites, are created by groups of designers and developers in teams. Typically, these applications are written modularly, with individuals focusing on one or two modules at a time. However, while the application may be broken down into smaller chunks, those chunks still need to play well together amid the larger application. Since developers are more focused on smaller areas of the application code base, it is not feasible for any one person to maintain their attention to the broader application. As such, it is common for changes that have been made by one developer to impact the work of the other developers in their team.

Providing tests early on in an application's life cycle ensures that any changes within the application's code do not create bugs that remain unnoticed. If an area of an application becomes unworkable due to breakages elsewhere in the application, they can be immediately identified and assigned to the relevant developer for fixing.

TEST-DRIVEN DEVELOPMENT (TDD)

There are numerous methodologies that can be implemented by companies when designing and building applications, regardless of the technologies used. Large companies, consisting of many hundreds of employees, often utilize a documentation approach to application development, whereby applications are designed first on paper using accepted descriptive modeling, such as the **Universal Modeling Language (UML)**. This is colloquially known as **documentation-driven development (DDD)**. Approaching software development in this manner is often necessary as the design may need to pass through an approval process by different company departments, often with no coding knowledge, before actual development begins. DDD aims to ensure that code is written once and written well while minimizing any potentially wasted development time.

When designing applications in this manner, a substantial amount of research, planning, and documentation is necessary, often requiring diligence by one or more employees, known as software architects. These employees may write no code themselves but may be responsible for identifying each module, class, function, and property within the application, with considerable care afforded to the cohesion of the application's code and that of any third-party services and hardware interfaces. Software architects are experts in their field, sometimes commanding larger salaries than the engineers tasked with writing the code their documents describe.

When smaller companies approach software design, the notion of DDD can be impractical. Small companies often lack the financial foundation required to fund a long-term product without the guarantee of an early return. This means software must reach a practical use milestone as quickly as possible in order to fund the next stage of its development. This is often especially true of web application development, which can provide a more immediate time to market than other software and often evolves consistently well after the initial production release.

To facilitate the needs of smaller companies and shorter development times, a new methodology was devised, known as **Agile Development**. Historically, agile development has always existed, often referring to developers operating out of their own bedrooms or their family basement. Such developers coded "off the cuff," developing software that works, but in the shortest possible time frame. In doing so, the software can be released to one or more paying customers much more quickly, thereby generating needed capital. If, after the release date, the software is found to contain bugs, then the developer has the incentive to continue supporting and improving an already profitable product. Agile development has also become more practical with the advent of the internet, whereby new code deployments can reach all platform users with minimal effort and time.

The term agile development was coined, arguably, in 2001 by the authors of the Agile Manifesto. The notion of agile development is one that fosters adaptive planning, evolving development cycles, and early and frequent delivery of features. However, the implementation of each stage, with specific emphasis on the "adaptive planning" aspect of this methodology, is left to the implementor's discretion. As an addendum to this approach, a higher-level methodology has since been devised and widely accepted that fulfills the majority of the "adaptive planning" requirements using tests. This methodology is known as TDD.

DOCUMENTATION FROM TESTS

TDD is the idea of writing tests prior to writing the code that they are meant to test. By identifying a feature required for any given software, the developer must write tests that cover every possible aspect of that feature at the smallest possible level, even if all such tests initially fail due to the lack of any present implementation of that feature. Once all the tests have been created, the developer can then implement that feature and ensure that it is created to the satisfaction of all of the tests.

The purpose of writing tests in this manner is that such tests form the initial documentation, or "active planning," of each feature. The software is, therefore, immensurable and agile since the documentation is written on a feature-by-feature basis. Likewise, the software is fully testable with complete code coverage and meets the limited time-to-market factor necessitated by the organization.

TDD purists ascertain that tests must always come first, prior to any application code, albeit for a given function or feature. If a code base already exists for an application and a new feature is required, then tests must be written to identify the confines of that feature's functionality. For large, complex features, breaking the required implementation into smaller chunks is advised. Therefore, it is important to be able to identify the requirements of the feature in advance.

TESTS, DECLARATIVE CODING, AND FUNCTIONAL PROGRAMMING

As we mentioned in previous chapters, declarative coding is a means to identify and express the processes required to carry out a task rather than how they are implemented. For instance, imagine you are tasked with creating a function that finds the average word length in a sentence. Programming this imperatively may look like the following:

```
function averageWordLengthInSentence(sentence) {  
  if (typeof sentence !== "string")  
    return;
```

```
let words = sentence.split(" ");
let count = [];
let sum = 0;
for (let i = 0; i < words.length; i++) {
  if (!words[i].length || words[i].length == 0)
    continue;
  count.push(words[i].length);
  sum += words[i].length;
}
return Math.round(sum / words.length);
}
console.log(averageWordLengthInSentence("The big brown fox jumped over
the lazy cat"));
```

If you run this block, you will be presented with the result 4.

Now, functionally, the preceding code is overly complex. It can be considered pure since it is non-effectual; it does not affect the global scope. However, testing the code within the function means testing the function in its entirety. If a bug were present in the function, it would be hard to spot, especially if it was a seldom-encountered bug.

To fix this, the processes that are occurring within the function should be declared, thus identifying the processes undertaken as a series of steps. Looking at the function line by line, we can visualize the following required process:

1. Validate whether the passed parameter is a string.
2. Decompose the string into a list of words.
3. Reject words that have no length.
4. Get the length of each word.
5. Get the average of each word's length.

Now that the required processes to complete the task are known, it is possible to assume the required functions using the exact same process:

- **validateString(sentence)**
- **splitStringIntoList(str)**
- **rejectEmptyStrings(strList)**
- **mapStringLength(strList)**
- **averageStringLength(lengthList)**

The preceding list of functions represents a composable functional list of processes. Since each function is simple, requiring input and returning output, it is therefore just as simple to reason about their logic and to write tests for them.

NOTE

At this stage, it is not important to understand how the functions will be composed or whether a function should be created at all. It is simply necessary to identify the steps in their smallest possible form factor.

UNIT TESTING

Now that the elements of the task have been defined, tests can be devised that validate each step, identifying what should happen when the correct values are passed to a function as arguments and what should happen if incorrect values are passed as arguments. This is known as unit testing.

Unit testing pertains to the testing of individual units of code – the functions – that exist in your application. The preceding sequence of functions may each have one or more unit tests created to validate their functionality. Also, a function that wraps the implementation of all five functions may also be unit tested, as required.

In order to write unit tests, you must first decide the implementation possibilities surrounding the function. For pure functions, this may simply be the following:

- What is returned when an acceptable argument is passed to the function's invocation?
- What is returned when an unexpected argument is passed to the function's invocation?

This list may be further extended if different types of values are accepted and/or returned from the function.

When dealing with impure functions, the list may be bigger, with tests becoming increasingly more complex. This is because it may no longer be possible to correctly reason about the outcome of the function's response to different argument types. Will the function raise an exception? Will it return a value that is contrary to its intended purpose? Does the function exhibit race condition-like symptoms due to asynchronous code? This is why developing your applications using functional paradigms saves effort, time, and problematic surprises.

Let's take a look at an example using the first function in the preceding list, that is, **validateString**. From its name and intended purpose, we can assume the following:

- It should return a truthy value if the passed argument is a string.
- It should return a falsy value if the passed argument is not a string.

Since the implementation will be functional, there should be no surprises from its invocation, and no additional test requirements can be derived. What cannot be derived, however, is exactly what may be returned from this function. For instance, if a Boolean value is returned to identify the success of the function's invocation, then how can it be composed or piped to other functions? Returning a Boolean will mean that any appended function invocation will receive a value of **true** instead of the original string.

One way to fix this is by wrapping the values in an object container, known as a **higher-kinded type**. This way, if the value is contained in one type, it is considered valid, while an alternative type may be considered invalid. Higher-kinded types will be discussed a little later in this chapter.

EXERCISE 17.01: SIMPLE UNIT TEST

In this exercise, you will write tests. These will simply be functions that test a function. The functions must test how a function may successfully return and how it may fail, if possible. Let's get started:

1. To begin, you will need a function that needs to be tested. This function will be simple, but it's possible that it may fail:

```
function add(a, b) {  
  if (typeof a !== "number" || typeof b !== "number") {  
    return NaN;  
  }  
  return a + b;  
};
```

The function simply adds two numbers together. If either of the values is not a number, then **NaN** is returned.

2. Now that the function has been defined, a test can be created for it:

```
function passesWithTwoNumbers() {  
  return add(1, 2) !== NaN;  
}
```


3. With the passing test written, you can then write any failing function tests. These should be written with content that expects failure:

```
function failsWithNonNumber() {  
  return isNaN(add("1", 2));  
}
```

NOTE

The function is successful if the test fails as we expect it to. This is a passing test, despite the function being tested failing.

4. Finally, you need to run all the tests and expect them all to pass. In this case, you will expect them all to return **true**:

```
var allSuccessful = passesWithTwoNumbers() && failsWithNonNumber();  
console.log(allSuccessful);
```

You should be presented with a value of **true** in the console, since both tests will have passed.

A test run simply checks whether all the tests passed. If any failed, then it will log those failures so that the developer knows where errors occur in their code. Test runners do not have to be complex, but frameworks oriented around testing typically aim to be easy to read, rather than simple.

When testing, it is important, first and foremost, that tests cover all the possibilities of how the function may be called. It is not important, however, that all the tests pass. This is simply because your code can be updated to satisfy all tests, but the tests themselves should never need to change.

HIGHER-KINDED TYPES

Using higher-kinded types simply means working with an abstraction. In times past, developers, libraries, and book authors devised many ways to ascertain whether a function returned a success or failure result. These attempts included the following:

- Success and failure callback functions
- Stylized objects with properties denoting their validity
- Returning **null** or **-1**

The problem with each of these solutions, however, is that they are neither declarative nor simple, nor foolproof. If a function needed to return a **null** value to denote success when every other function returned that only when erroring, then how is the developer intended to handle that edge case?

To resolve this issue, other functional languages, such as Haskell, resorted to wrapping values in a simple construct. Otherwise known as a **Functor**, these containers are simply used to represent a specific **type**. The container typically provides at least two functions: one to wrap a value and one to extract the value:

```
let Identity = function(value) {
  this._val = value;
};
Identity.from = function(value) {
  return new Identity(value);
};
Identity.prototype.get = function() {
  return this._val;
};
let id = Identity.from("bob");
console.log(id.get());
console.log(id instanceof Identity);
```

The **console.log** calls in the preceding code will output the following:

```
"bob"
true
```

Languages that use **Functors** always denote a simple value type with the name **Identity** or **Id**. The value they represent may be anything – an integer, a string, or an object. The point is simply to be able to annotate the value in some fashion. Ergo, if you consider an **Identity** type as being a valid type, then perhaps a **Failure** type is its antithesis:

```
let Failure = function(value) {
  this._val = value;
};
Failure.from = function(value) {
  return new Failure(value);
};
```

```
Failure.prototype.get = function() {  
  return this._val;  
};  
let failed = Failure.from([1, 2, 3]);  
console.log(failed.get());  
console.log(failed instanceof Failure);  
console.log(failed instanceof Identity);
```

When running this block, you will see the following output:

```
1,2,3  
true  
false
```

As you can see, the **Failure** type is relatively identical to the **Identity** type.

By using types, passing values to and from functions becomes declarative, just as with their function counterparts, which makes testing their implementation much simpler.

FUNCTORS WITH FUNCTIONS

With this in mind, let's reexamine the previous problem. The **validateString** function should accept a string, but since you don't know whether the invocation of the function will occur within a compose or pipe chain, you must always expect it to receive a **Functor**. Since the function will only work with string values, you also need to check that the **Functor** contains a string value. Once processing has succeeded, the function will then return a **Functor**. Now, with these requirements in place, you can now deduce the following:

It should return **Failure** if it is not passed an **Identity** instance.

It should return **Failure** if the passed **Identity** doesn't contain a string value.

It should return **Identity** if the passed **Identity** contains a string.

The preceding list essentially prescribes the tests necessary to describe how your function will work. Note that you have not created your function yet; you have merely reasoned about how it will look from the outside. In the next section, we'll take a look at actually writing the tests for this function.

THE JEST TESTING FRAMEWORK

To write tests, you first need a test-oriented framework. There are many test frameworks in JavaScript that cover all manner of test types. However, for unit testing, none are simpler to implement than the Jest framework.

NOTE

For detailed information about Jest, including its documentation, source code, and community, visit <https://jestjs.io/>.

Jest is a complete testing framework. It consists of numerous functions and an opinionated implementation that allows you to write tests that sit alongside your greater application. By writing tests in separate modules, you can execute those tests against your code to ensure that they are correct while ensuring that those same tests do not get compiled into the final release build.

The Jest testing framework was developed by Facebook in 2014 and is compatible with all the leading development frameworks, including React, AngularJS, and Vue.js, as well as with regular JavaScript and Babel.

EXERCISE 17.02: SETTING UP TESTS USING BABEL

To get started with Jest, we'll walk through setting up a new JavaScript project using Babel and utilizing the bare minimum libraries required to work with Jest. This way, you can see just how powerful Jest is, right out of the box. Let's get started:

1. To begin, create a new working directory and navigate to it with your command console or Bash terminal. Then, initialize it for **npm** with the following code:

```
npm init
```

2. Feel free to use the default values where necessary for each of the questions that npm presents.
3. Next, run the following **npm** command to install the necessary libraries:

```
npm install --save-dev @babel/preset-env jest babel-jest
```

This will install the development-time dependencies. There will be no need for production-time dependencies in this chapter.

4. With the dependencies installed, you'll need to configure the Jest installation. To do this, open the **package.json** file that was added to the project directory and add the following information:

```
"jest": {
  "clearMocks": true,
  "rootDir": "src"
},
"babel": {
  "presets": [
    [
      "@babel/env",
      {
        "targets": {
          "browsers": [
            "last 2 versions"
          ]
        },
        "useBuiltIns": "usage"
      }
    ]
  ]
}
```

The entries should be placed in the root object as siblings to the scripts and dependencies objects. The preceding entries ensure that Babel compiles to the last two versions of each of the major browsers and tells the Jest framework where to find your tests.

5. Finally, update the **scripts** parameter in **package.json** to the following:

```
"scripts": {
  "test": "jest"
},
```

6. With your **package.json** file configured, go ahead and save it. Then, create a directory called **src** in the root of the project directory and, within that, create a blank file called **sentence.test.js**. You can now run the tests by executing the following code:

```
npm run test
```

7. If everything went according to plan, you should see the following output displayed in the console:

```
$ npm run test

> examples@1.0.0 test /home/jahred/Documents/writing/C14377 - Comprehensive Java
Script/examples
> jest

FAIL src/_.test.js
  ● Test suite failed to run

    Your test suite must contain at least one test.

      at ../../node_modules/@jest/core/build/TestScheduler.js:242:24
      at asyncGeneratorStep (../../node_modules/@jest/core/build/TestScheduler.js:1
31:24)
      at _next (../../node_modules/@jest/core/build/TestScheduler.js:151:9)
      at ../../node_modules/@jest/core/build/TestScheduler.js:156:7
      at ../../node_modules/@jest/core/build/TestScheduler.js:148:12
      at onResult (../../node_modules/@jest/core/build/TestScheduler.js:271:25)

Test Suites: 1 failed, 1 total
Tests:       0 total
Snapshots:   0 total
Time:        0.597s
Ran all test suites.
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! examples@1.0.0 test: `jest`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the examples@1.0.0 test script.
npm ERR! This is probably not a problem with npm. There is likely additional log
ging output above.

npm ERR! A complete log of this run can be found in:
npm ERR!     /home/jahred/.npm/_logs/2019-08-08T18_55_53_574Z-debug.log
```

Figure 17.1: Failing test run with no tests

As you can see, the test suite executed fine, but the result shows that it failed. This is simply because there are no tests yet.

WRITING TESTS

Now that Jest is installed, it's time to write the tests. Test scripts that are created using Jest are simply composed of a series of function calls in the following form:

```
test(<test label>, <test callback>);
```

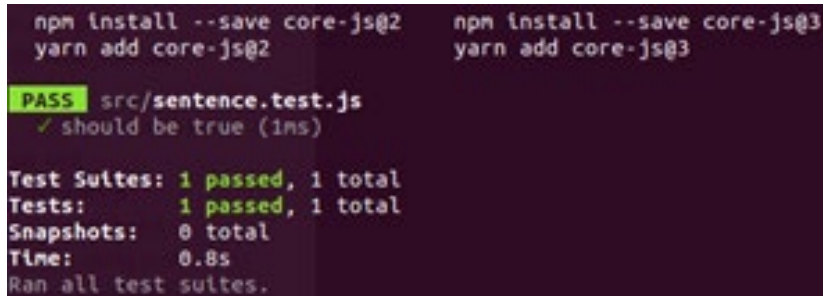
As we stated previously, a test tests a unit of functionality, preferably as pure and as singular a functionality as possible. Each test should test one aspect of that functionality. If more aspects are required to be tested, then more tests should be created.

Tests work by executing a piece of code, then verifying what it should or shouldn't have done. In Jest, validating that information is typically carried out using the **expect** function call.

Let's start by creating a simple test. Enter the following in **sentence.test.js**:

```
test('should be true', () => {  
  expect(true);  
});
```

Once saved, run the tests as before from the command line using **npm run test**. You should see something like the following:

A terminal window with a dark background showing the execution of a Jest test. The top two lines show the installation of dependencies: 'npm install --save core-js@2' and 'yarn add core-js@2' on the left, and 'npm install --save core-js@3' and 'yarn add core-js@3' on the right. Below this, the test file 'src/sentence.test.js' is highlighted in green, followed by a green 'PASS' label and the test description '✓ should be true (1ms)'. A summary of test results follows: 'Test Suites: 1 passed, 1 total', 'Tests: 1 passed, 1 total', 'Snapshots: 0 total', 'Time: 0.8s', and 'Ran all test suites.'

```
npm install --save core-js@2    npm install --save core-js@3  
yarn add core-js@2             yarn add core-js@3  
  
PASS src/sentence.test.js  
✓ should be true (1ms)  
  
Test Suites: 1 passed, 1 total  
Tests: 1 passed, 1 total  
Snapshots: 0 total  
Time: 0.8s  
Ran all test suites.
```

Figure 17.2: Passing a test with expect

You have successfully created a passing test.

What you just accomplished was simple. The test executes the passed callback function and associates it with the test label. The body of the function should make calls to the Jest assertion functions in order to describe a passing test. If all of the Jest function call conditions are truthy, then the test passes. If at least one of the Jest assertion function calls fails, then the test will fail. Let's try that now so that you know what a failing test looks like.

Update the function you added to **sentence.test.js** to the following:

```
test('should be true', () => {  
  expect(true).toBe(false);  
});
```

This is a failing condition. Notice that the failing function call isn't simply like so:

```
expect(false);
```

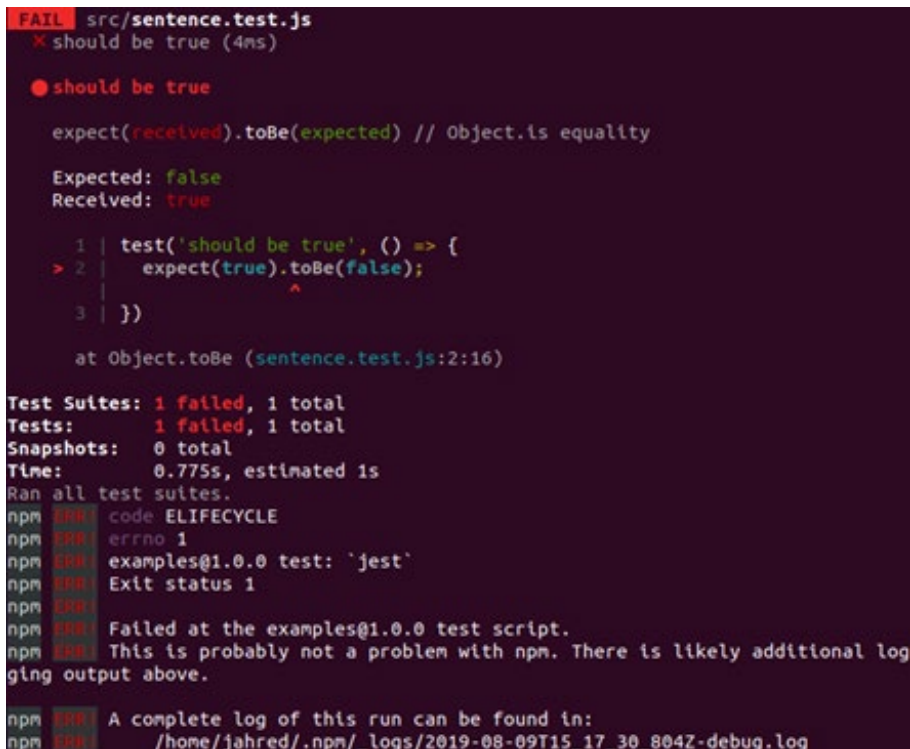
This is because simply passing **false** to **expect** is a passing result. The **expect** function is like the constructor of a Jest assertion. Therefore, any value that's passed to **expect**, without assertion parameters being applied, will appear as a passing assertion.

The **toBe** function is a common and simple chained assertion function that's used to compare the value within the **expect** parenthesis. It simply means the value that's passed to **expect** should be equal in value to the value in **toBe**. It is similar to using the **===** strict equality operator, so it will not match objects unless they reference the very same instance of an object.

If you need to test whether a value is an instance of a specific type, then this can be accomplished using **toBeInstanceOf**:

```
expect(value).toBeInstanceOf(Identity);
```

Now, run the test again. You should now be presented with the following output:



```
FAIL src/sentence.test.js
  ✕ should be true (4ms)

● should be true

  expect(received).toBe(expected) // Object.is equality

  Expected: false
  Received: true

   1 | test('should be true', () => {
>  2 |   expect(true).toBe(false);
     |                 ^
   3 | })
    at Object.toBe (sentence.test.js:2:16)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:  0 total
Time:        0.775s, estimated 1s
Ran all test suites.
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! examples@1.0.0 test: `jest`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the examples@1.0.0 test script.
npm ERR! This is probably not a problem with npm. There is likely additional log
ging output above.

npm ERR! A complete log of this run can be found in:
npm ERR! /home/jahred/.npm/_logs/2019-08-09T15_17_30_804Z-debug.log
```

Figure 17.3: Failing Test with expect

As you can see, when tests fail, the Jest test runner outputs a whole bunch of information pertaining to each failing test. This includes the following:

- The label assigned to the test
- The failing assertion
- The line number and character position of where the first failing assertion call begins
- The total number of tests run and the total number of those tests that failed

If multiple assertions exist within a test and all of them are erroneous, only the first failing assertion will be flagged as failing. This is because any further assertions within the test are skipped as soon as an assertion has failed, allowing the next test to be run.

TESTING THE `VALIDATESTRING` FUNCTION

Now that you know how tests are written, let's examine how the tests need to be written for the `validateString` function. In the *Functors with Functions* section, we highlighted three tests that are required to validate the working of the function. Each test will provide a possible argument to the function and then assert the result that's returned from the function's invocation. The tests need to be succinct. Since `validateString` will be a pure function, the tests can be simple, requiring no knowledge of any properties, functions, or environmental factors outside of the function call.

NON-IDENTITY ARGUMENT

The first test to attempt against the `validateString` function is as follows:

```
it should return Failure if it is not passed an Identity instance
```

This test will simply invoke the `validateString` function and pass it a string value. The purpose of this test is to signify specifically that a `non-Identity` instance will cause the invocation to fail. Therefore, passing a string, which may ordinarily be considered valid, is sufficient to prove this. The returned value from the invocation will then be asserted.

IDENTITY ARGUMENT WITH A NON-STRING VALUE

The next test to attempt against the **validateString** function is as follows:

```
it should return Failure if the passed Identity doesn't contain a string value
```

This test will pass an **Identity** function, but it will not contain a string value. Remember that it is only important to assert the specific test case. Therefore, writing tests for every possible type other than string is not practical. For the purposes of this test, use an integer value.

IDENTITY ARGUMENT WITH A STRING VALUE

The final test in the list will involve an **Identity** parameter containing a string value, regardless of whether or not it contains characters. This is the ideal argument for the invocation of **validateString**:

```
it should return Identity containing an Array if the passed Identity contains a string
```

The resulting value from the invocation should be an **Identity** instance that contains the original string value. It could even be the original value that was passed as the argument to the array since the object will be unchanged.

EXERCISE 17.03: POPULATING THE TESTS

This exercise will proceed with populating the tests in your Jest project. It will not, however, create the actual working function. Remember, TDD is all about writing the tests first. Once the tests have been fully realized, the code of the modules these tests describe can be realized. Let's get started:

1. Open the **sentence.test.js** file and remove all of its content. You'll start fresh for this exercise. Then, create a test call for each of the aforementioned criteria, using the description as its label:

```
test('it should return Failure if it is not passed an Identity instance', () => {
  });

test('it should return Failure if the passed Identity doesn\'t contain a string value', () => {
  });

test('it should return Identity containing passed string if the passed Identity contains a string', () => {
  });
```

2. Now, populate the first test with the following:

```
test('it should return Failure if it is not passed an Identity instance', () => {
  let id = 'this is a string';
  let res = validateString(id);
  expect(res).toBeInstanceOf(Failure);
});
```

Here, the **Identity** object is constructed, but the **Failure Functor** instance is expected to return.

3. The next test is almost identical, but with an **Identity** argument:

```
test('it should return Failure if the passed Identity doesn\'t contain a string value', () => {
  let id = Identity.from(1234);
  let res = validateString(id);
  expect(res).toBeInstanceOf(Failure);
});
```

The value that's passed to the **Identity** factory function is an integer, though it could have been any non-string value.

4. Finally, the third test will contain the passing criteria:

```
test('it should return Identity containing passed string if the passed Identity contains a string', () => {
  let id = Identity.from('this is a string');
  let res = validateString(id);
  expect(res).toBeInstanceOf(Identity);
  expect(res.get()).toBe(id.get());
});
```

Since the result is a **Functor**, the test asserts both the returned values type and the value it contains.

5. Now, run the tests with the following code:

```
npm run test
```

6. The resulting test should output failures for all three tests:

```

FAIL src/sentence.test.js
  ✕ it should return Failure if it is not passed an Identity instance (ms)
  ✕ it should return Failure if the passed Identity doesn't contain a string value (ms)
  ✕ it should return Identity containing an Array if the passed Identity contains a string

  ● it should return Failure if it is not passed an Identity instance

    ReferenceError: validateString is not defined

       1 | test('it should return Failure if it is not passed an Identity instance', () => {
       2 |   let id = "this is a string";
    >    3 |   let res = validateString(id);
         |           ^
       4 |   expect(res).toBeInstanceOf(Failure);
       5 | });
       6 | test('it should return Failure if the passed Identity doesn't contain a string value', () => {
    at Object.validateString (sentence.test.js:3:13)

  ● it should return Failure if the passed Identity doesn't contain a string value

    ReferenceError: Identity is not defined

       5 | });
       6 | test('it should return Failure if the passed Identity doesn't contain a string value', () => {
    >    7 |   let id = Identity.from(1234);
         |           ^
       8 |   let res = validateString(id);
       9 |   expect(res).toBeInstanceOf(Failure);
      10 | });
    at Object.Identity (sentence.test.js:7:12)

  ● it should return Identity containing an Array if the passed Identity contains a string

    ReferenceError: Identity is not defined

      10 | });
      11 | test('it should return Identity containing an Array if the passed Identity contains a string', () => {
    >    12 |   let id = Identity.from("this is a string");
         |           ^
      13 |   let res = validateString(id);
      14 |   expect(res).toBeInstanceOf(Identity);
      15 |   expect(res.get()).toBe(id.get());
    at Object.Identity (sentence.test.js:12:12)

Test Suites: 1 failed, 1 total
Tests:       3 failed, 3 total
Snapshots:   0 total
Time:        0.739s, estimated 1s
Ran all test suites.

```

Figure 17.4: Populated tests in the Jest project

Note that each test fails because of either the unknown function reference or references.

TEST FULFILMENT

Once your tests have been written, your function should be sufficiently documented and described, even if it does not exist yet. What's more, once you have run your tests and they have failed, you also have a manifest for the construction of your function.

A premise of TDD is that just enough code must be written to pass each test, but no more. Since the tests outline how the subject matter should work, both in expected and unexpected scenarios, those same tests therefore facilitate the scaffold for the subject matter's functionality. This means you simply need to respond to each error as it arises, rerunning your tests each time. Once all your tests pass, the subject matter of those tests is complete.

FULFILLING REFERENCE ERRORS

Each of the errors that are visible, when running the tests as they are, are referential errors. They exist because the tests cannot find the **Functors**, nor the **validateString** function, defined within the test scope. The reason for this is simply because those values have not been defined yet.

Tests exist to document and verify functionality in your core application. As such, the references to these items will not be declared within the test modules themselves. Instead, they will be imported into the test module from your main application modules.

In the current project, no application modules exist. Let's remedy that situation now.

EXERCISE 17.04: BUILDING FUNCTOR FUNCTIONALITY

FunctorsFailure and **Identity** have already been detailed. This exercise will place those objects into their own modules and the **validateString** function into its own module, and will reference them within the test module so as to resolve the current errors from the last test run. Let's get started:

1. Create a new file in the **src** directory called **functors.js** and paste the Functor object definitions below it:

functors.js

```
1 let Identity = function(value) {
2   this._val = value
3 };
4 Identity.from = function(value) {
5   return new Identity(value)
6 };
7 Identity.prototype.get = function() {
8   return this._val
9 };
10 let Failure = function(value) {
11   this._val = value
12 };
```

<https://packt.live/2NVfm7N>

2. As we discussed in the previous chapters, the last line allows both Functors to be imported into other modules by exposing them externally to their own module. Ensure you save this file.
3. Next, create a new file in **src** called **sentence.js** and populate it with the following simple function signature:

```
let validateString = function(value) {  
  };  
export {validateString};
```

This is the primary application module for this chapter. You'll notice that it has a similar name to the test module but is missing the **test** segment. This is typical of applications, wherein each module provided by the application has matching test modules that test the functions defined within it. Using a naming convention allows for the instant recognition of application code and their associated tests.

4. With the two modules complete, open up **sentence.test.js** and add the following **import** references to the top of the file:

```
import {Identity, Failure} from './functors';  
import {validateString} from './sentence';
```

This will import each of the missing references into the test scope.

5. Now, rerun the tests. The referential errors that were described in the last run should now be gone. You should instead be presented with new errors, detailing the mismatch of data types:

```

FAIL src/sentence.test.js
  ✖ it should return Failure if it is not passed an Identity instance (6ms)
  ✖ it should return Failure if the passed Identity doesn't contain a string value (1ms)
  ✖ it should return Identity containing an Array if the passed Identity contains a string

  ● it should return Failure if it is not passed an Identity instance

    expect(received).toBeInstanceOf(expected)

    Expected constructor: Failure

    Received value has no prototype
    Received value: undefined

       6 |     let id = "this is a string";
       7 |     let res = validateString(id);
    >    8 |     expect(res).toBeInstanceOf(Failure);
         |                   ^
       9 |   });
      10 |   test('it should return Failure if the passed Identity doesn't contain a string value', () => {
      11 |     let id = Identity.from(1234);
        at Object.toBeInstanceOf (sentence.test.js:8:15)

  ● it should return Failure if the passed Identity doesn't contain a string value

    expect(received).toBeInstanceOf(expected)

    Expected constructor: Failure

    Received value has no prototype
    Received value: undefined

      11 |     let id = Identity.from(1234);
      12 |     let res = validateString(id);
    >    13 |     expect(res).toBeInstanceOf(Failure);
         |                   ^
      14 |   });
      15 |   test('it should return Identity containing an Array if the passed Identity contains a string', () => {
      16 |     let id = Identity.from("this is a string");
        at Object.toBeInstanceOf (sentence.test.js:13:15)

  ● it should return Identity containing an Array if the passed Identity contains a string

    expect(received).toBeInstanceOf(expected)

    Expected constructor: Identity

    Received value has no prototype
    Received value: undefined

      16 |     let id = Identity.from("this is a string");
      17 |     let res = validateString(id);
    >    18 |     expect(res).toBeInstanceOf(Identity);
         |                   ^
      19 |     expect(res.get()).toBe(id.get());
      20 |   });
        at Object.toBeInstanceOf (sentence.test.js:18:15)

console.log src/sentence.test.js:5
[Function: Identity] { from: [Function] }

```

Figure 17.5: Type mismatch errors

TYPE MISMATCH ERRORS

At this point, you now have concrete values to work with. If you hadn't defined the **Functors** used by your function yet, it would be necessary to devise tests and construct those individually first. Since the **Functors** were already defined, those tests have been skipped.

NOTE

Note that when writing tests for your applications, you should attempt to ensure as much coverage of your code as possible, writing tests for each of the functions you write. However, since TDD advocates writing tests prior to the creation of application code, it doesn't make sense to write tests for existing code. Typically, tests for existing code are good to have as they validate that your application will run as expected. It is up to you to decide whether tests for the existing code should be created in retrospect.

With each of the actors in your tests present in spirit, it is time to bring them into the body by writing the code needed to fulfill the next iteration of test errors.

As shown in the preceding screenshot, the tests are failing due to slightly differing issues. The first two tests are expecting the **validateString** function invocation to return a **Failure Functor** instance, while the third test is expecting an **Identity Functor** instance. To resolve these issues, let's work on them individually.

EXERCISE 17.05: RESOLVING ERRORS FROM UNIT TESTS

In this exercise, you will modify the **validateString** function until it meets the criteria of each of the three tests. Let's get started:

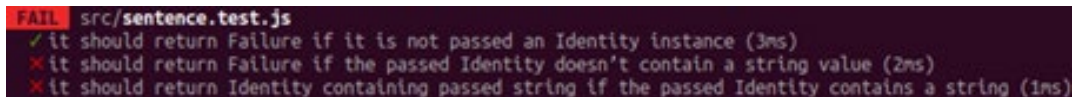
1. To fulfill the requirements of the function, the **sentence.js** module will need to utilize the **Identity** and **Failure Functor** types. Add the relevant import declaration at the top of the module:

```
import {Identity, Failure} from './functors';
```


2. To pass the first test, it is necessary to identify only those passed arguments that are not an instance of **Identity**. If this occurs, then an instance of **Failure** must be returned. Facilitating this will resolve the first test entirely and is as simple as it sounds. Update the **validateString** function to include this condition:

```
let validateString = function(value) {  
  if (!(value instanceof Identity))  
    return Failure.from("invalid argument");  
};
```

3. Run the tests. The first test should now pass:



```
FAIL src/sentence.test.js  
✓ it should return Failure if it is not passed an Identity instance (3ms)  
✗ it should return Failure if the passed Identity doesn't contain a string value (2ms)  
✗ it should return Identity containing passed string if the passed Identity contains a string (1ms)
```

Figure 17.6: First test pass output

NOTE


The first test presents a green tick, signifying that the test criteria have been met.

4. For the second test, if the argument is an **Identity** instance, then its contained value must also be a string value. Again, this is easily accomplished with a simple condition. Update the **validateString** function to include it:

```
let validateString = function(value) {  
  if (!(value instanceof Identity))  
    return Failure.from("invalid argument");  
  if (!(typeof value.get() === "string"))  
    return Failure.from("invalid argument");  
};
```

Since a string is a primitive type, it makes sense to compare it with the **typeof** keyword.

5. Run the tests a second time to see both the first and second tests passing successfully:



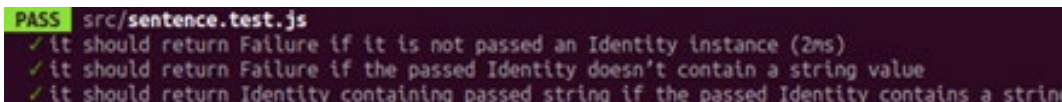
```
FAIL src/sentence.test.js
✓ it should return Failure if it is not passed an Identity instance (2ms)
✓ it should return Failure if the passed Identity doesn't contain a string value
✗ it should return Identity containing passed string if the passed Identity contains a string (2ms)
```

Figure 17.7: Second test pass output

6. The final test requires an Identity instance, with the same string value, to be returned if the full criteria are met. Since the first two tests already define erroneous argument passing, and since the value of the return type should be the same as the passed argument, the third test can be satisfied by simply returning the original argument value:

```
let validateString = function(value) {
  if (!(value instanceof Identity))
    return Failure.from("invalid argument");
  if (!(typeof value.get() === "string"))
    return Failure.from("invalid argument");
  return value;
};
```

7. If you run the tests for a third time, you should now see all of them pass successfully:



```
PASS src/sentence.test.js
✓ it should return Failure if it is not passed an Identity instance (2ms)
✓ it should return Failure if the passed Identity doesn't contain a string value
✓ it should return Identity containing passed string if the passed Identity contains a string
```

Figure 17.8: All tests pass output

You have completed your first TDD implementation.

ACTIVITY 17.01: TEST-DRIVEN DEVELOPMENT

Now that you have seen working tests with Jest, it is time to implement some test-driven development knowledge with an activity.

In this task, you need to describe four functions using tests: **add**, **subtract**, **multiply**, and **divide**. Each function should take two numerical arguments and return a single numerical argument. Any tests you write should define what happens when each function receives valid arguments and what should happen when invalid arguments are passed.

Steps:

1. Create the function signatures. The functions should have two arguments each but should have no actual code within them at this stage.
2. Since this is TDD, the tests should always be defined before the function's code.
3. Start optimistically. Write successful tests for each function first, then write failure tests after.
4. With the tests defined, run them and ensure that they fail.
5. Add code to each function and rerun the tests. Repeat until all tests pass.

This same methodology is how all TDD projects should be approached. This same process works identically, regardless of whether the function being tested is simple or complex. Simply define the scope of the function with tests, then flesh it out until all the tests pass.

JEST EXPECT METHODS

So far, you have seen only two of the methods provided by Jest's **expect** object. However, there are numerous methods, known as matchers, that can assert all kinds of conditions – even ones you write yourself. The Jest documentation lists a large number of immediately available methods for use in your tests. You can find this information here: <https://packt.live/2PZwuMk>.

We'll look at some of the more important methods throughout this chapter.

When working through the rest of this chapter, keep in mind a major philosophy behind writing tests. That is, tests should be succinct and easy to understand. There should be minimal setup required to run the test. All the assertions within a test should read like a sentence and should utilize a minimal form.

BOOLEAN ASSERTIONS

Previously, you were shown that simply calling **expect** with a value is not enough to determine a truthy assertion. However, calling **toBe(true)** against a value is verbose and considered ill form. What's more, **toBe(true)** will only assert whether the value it is being compared with is specifically **true**, not whether it is truthy. The **expect** methods list provides several alternatives that make such assertions easier to read and understand, including **toBeTruthy** and **toBeFalsy**:

```
expect(1 < 2).toBeTruthy();
expect(2 < 1).toBeFalsy();
expect("some string").toBeTruthy();
```

In situations where a Boolean operator would provide a valid condition in a test, such as in the first two examples in the preceding code, the **expect** object provides a number of methods that facilitate those same expressions, but in an English readable format:

```
expect(1).toBeLessThan(4);
expect(2).toBeLessThanOrEqual(3);
expect(3).toBeGreaterThan(2);
expect(4).toBeGreaterThanOrEqual(1);
```

FALSY ASSERTIONS

Sometimes, it is necessary to determine a specific type of falsy value, such as a **null**, **undefined**, or **NaN** response from a function. Jest provides specific matchers for these types using the expected format:

```
expect(null).toBeNull();
expect(undefined).toBeUndefined();
expect(NaN).toBeNaN();
```

REFUTING ASSERTIONS

If to assert means to ensure that something is **true** or absolute, then the opposite is to refute. Refuting an assertion can be required to reduce the number of calls needed to define a test case. For instance, if you wished to assert that a function returned any value other than **null**, then it would be impractical to test for all value types except **null**.

Jest provides an intermediary method call to change an assertion to a refute case, called the **not** method:

```
expect(1).not.toBeNull();  
expect(false).not.toBeTruthy();  
expect({}).not.toBeInstanceOf(Array);
```

OBJECT ASSERTIONS

You've already seen **toBeInstanceOf** when working with objects. Objects are complex types. So, it makes sense that asserting the properties of an object is more complex than that of a primitive type. **toBeInstanceOf** is a rather tidy way of determining the generality of an object, but it's not much use when working with raw objects or when the instance type is transient.

Jest provides several useful methods for working with types, with the most frequently used possibly being **toHaveProperty**.

The **toHaveProperty** method has a couple of purposes: to assert whether an object contains a property or to assert whether an object contains a property with a specific value:

```
expect({key: "value"}).toHaveProperty("key");  
expect({key: "value"}).toHaveProperty("key", "value");  
expect({key: "value"}).not.toHaveProperty("prop");
```

What makes **toHaveProperty** special, however, is its deep referencing capabilities.

Often, objects will contain other objects. Sifting through a tree of objects to assert a deeply nested property can be troublesome, particularly if any one of those object branches may not exist or may exist as a non-object type. When working with deeply nested objects, it may be tempting to write assertions like this:

```
let obj = {  
  first: {  
    second: {  
      third: 5  
    }  
  }  
}
```

```
expect(obj).toHaveProperty("first");  
let first = obj.first;  
expect(first).toHaveProperty("second");  
let second = first.second;  
expect(second).toHaveProperty("third", 5);
```

The **toHaveProperty** method alleviates such complexities by supporting deeply nested property assertion. There are two approaches to this, with the first approach utilizing dot notation, much like known object property access in JavaScript itself:

```
expect(obj).toHaveProperty("first.second.third", 5);
```

The second approach utilizes an array of properties, in hierarchical order:

```
expect(obj).toHaveProperty(["first", "second", "third"], 5);
```

Using an array is especially useful if the property access list is a dynamic list of values. However, it also poses another benefit.

Since objects in JavaScript can utilize any string form you wish, it is also possible to use a property name that follows the same dot notation, like so:

```
let alt = {"first.second": 10};
```

When referencing this value with **toHaveProperty**, doing the following will not work:

```
expect(alt).toHaveProperty("first.second", 10); // fails
```

This is because the underlying **toHaveProperty** handler expects the property name to be a deeply nested reference and will have it no other way. To resolve this, utilize the **array** option instead:

```
expect(alt).toHaveProperty(["first.second"], 10); // succeeds
```

ARRAY ASSERTIONS

Like objects, arrays also have methods for asserting values. The simplest of these is **toHaveLength**, which simply asserts the length of an object:

```
let arr = [1, 2, 3];  
expect(arr.length).toBe(3); // incorrect form  
expect(arr).toHaveLength(3); // accepted form
```

The other rather powerful assertion is **toContain**, which asserts that at least one of an array's values is equal to another value:

```
expect(arr).toContain(2);
```

Note, however, that this may not be what you want when dealing with arrays of objects. For instance, you may want to check that an array contains an object of a certain structure, but don't have a specific reference to an object that is contained therein. Using **toContain** is useless in this scenario because it works similarly to **toBe**, whereby only the object references are compared. To solve this particular problem, you would instead use **toContainEqual**.

toContainEqual is not similar to **toBeInstanceOf** as it does not care about inheritance. It only cares about the structure of the object in question:

```
let obj2 = {one: 1, two: {three: 3}};  
let arr2 = [obj2];  
expect(arr2).toContainEqual({one: 1, two: {three: 3}});
```

As you can see, the handler supports deep equality, so that nested objects are also matched.

toContainEqual matches objects exactly, meaning partial matches do not work. For example, the following will not pass:

```
let obj3 = {one: 1, two: {three: 3}, fourth: 4};  
let arr3 = [obj3];  
expect(arr3).toContainEqual({one: 1, two: {three: 3}}); // fails
```

FUNCTION ASSERTIONS

Since functions are objects too, it makes sense that there would be assertions for them. There are many reasons why you would want to assert a function and, indeed, many ways a function that can be asserted.

A common use case for function assertion is to assert that a function was called. If all you require is to know that a function was invoked, then **toHaveBeenCalled** will suffice:

```
expect(func).toHaveBeenCalled();
```

However, just calling the assertion will not work. For Jest to know of the function's existence and, therefore, to keep track of its usage, it must first be wrapped using the **jest.fn** helper:

```
let func = (x) => x + x;  
func(2);  
expect(func).toHaveBeenCalled();
```

If you need to know that the function was called, but you also want to know what the invocation returned, then you can use **toHaveBeenCalled**With:

```
expect(func).toHaveBeenCalled(4);
```

Finally, if you want to validate that the function was called and the arguments that were passed to it, you could use **toHaveBeenCalled**With:

```
expect(func).toHaveBeenCalled(2);
```

toHaveBeenCalledWith accepts an arbitrary number of arguments, which means it can match the values you expect your function to have received.

This was a slice of the function assertion methods provided by Jest. For a complete listing, visit the Jest documentation website.

TEST MODULE STRUCTURE

When creating your test modules, as we discussed previously, you will want to make your tests as succinct as possible. Each of the tests within a module should accurately, but concisely, describe the purpose of your tested units. In so doing, you will need to consider ways to reduce code within your tests and within the test module in general.

The Jest framework provides a number of ways to reduce clutter in your test modules and make your tests more readable.

THE DESCRIBED FUNCTION

If a module in your application has its sister test module, and each function within the application module has one or more unit tests, then it is likely that your test modules will grow very large indeed. Often, unit test modules can become quite difficult to traverse, especially when returning to update a test or add further tests for a specific function.

The Jest framework provides the **describe** function to help assuage this issue by acting as a wrapper around related tests:

```
describe('test wrapper', () => {
  test('this should fail', () => {
    expect(false).toBeTruthy();
  })
  test('this should pass', () => {
    expect(true).toBeTruthy();
  })
})
```

A **describe** call contains a text label, much like a **test** call. This label is prepended to **test** labels when errors occur so that erroring tests are easier to locate:

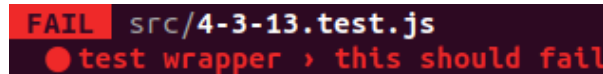


Figure 17.9: Described test failure

Each test module in your application can have one or more **describe** calls. With them in place, you can then easily locate tests by simply looking for the parent **describe** call first, and then for its nested **test** call.

TEST SETUP AND TEARDOWN FUNCTIONS

Often, related tests will have the same setup requirements. For instance, a complex object may need to be created with a given number of steps for multiple tests oriented around a single group of functions. Verbosity is the antithesis of good tests. Since tests should always be succinct, it makes sense to group such setup code into a singular location.

Jest provides handy methods for both the setting up and tearing down of tests, with each being available on a per-test basis or for all tests. These are **beforeEach**, **beforeAll**, **afterEach**, and **afterAll**:

```
let started;
beforeAll(() => {
  started = new Date();
  return started;
})
afterAll(() => {
  let finished = new Date();
  console.log("Tests took", finished.getTime() - started.getTime(),
```

```
"milliseconds");  
})  
test('some test', () => {  
  expect(true).toBeTruthy();  
})
```

Test setup and teardown are extremely useful when ensuring a clean environment. For instance, you may need to use the **before** functions to set up a data store and use the **after** functions to delete the store:

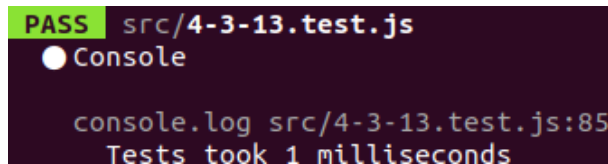


Figure 17.10: Setup and teardown result

Test setup and teardown can also be combined with the **describe** function call. If using one of the functions outside of **describe** blocks, the setup/teardown will run for each of the functions within all of the **describe** blocks. Using setup/teardown within a **describe** block will run for tests within that block only.

When calling asynchronous requests within any of the setup/teardown functions, returning the generator or promise of that request will delay the execution of the tests until that generator/promise resolves. This is essential for tests that execute as a result of loading data from an external source, ensuring that the tests do not execute before the data is ready.

ASYNCHRONOUS ASSERTIONS

So far, all of the tests you have run or experimented with have been synchronous; a function was invoked and its return value was asserted. Often, however, this won't be the case. Instead of the function return value being asserted, you will instead want to assert on the result of an asynchronous process that may have executed long after the test has ended.

To see how asynchronous tests may pass or fail, Jest provides the **assertions** method:

```
expect.hasAssertions();
```

hasAssertions simply asserts whether at least one other assertion succeeded within the test. This is particularly useful with asynchronous tests, whereby you are never truly sure whether an assertion executed and passed or simply never executed at all.

A more precise alternative to **hasAssertions** is the **assertions** method, which asserts only if a specific number of assertions were made:

```
expect.assertions(2);
```

With this in mind, let's look at a failing asynchronous test:

```
test('Failing async call', () => {  
  expect.hasAssertions();  
  setTimeout(() => {  
    expect(true).toBeTruthy();  
  }, 1000);  
})
```

When run, you should see an error that looks something like the following:



Figure 17.11: Failing asynchronous test

As you can see, while an assertion existed within the test, the test ended before the assertion was ever called. Since the **hasAssertions** call specified that an assertion must take place for the test to pass, it resulted in a failure.

In order to make such a test pass, you need to utilize either a generator or a promise. By returning the promise from the test, the test never exits until the promise resolves:

```
test('Passing async call', () => {
  expect.hasAssertions();
  function later(delay) {
    return new Promise(function(resolve) {
      setTimeout(resolve, delay);
    });
  }
  return later(1000).then(() => expect(true).toBeTruthy());
})
```

In the preceding example, the required assertion doesn't occur until after resolving the promise. If the promise is not returned from the test, then Jest never expects an asynchronous requirement and would fail the test. However, since the promise is returned, the test passes successfully.

CUSTOM ASSERTIONS

Earlier in this chapter, a pipeline was discussed that would convert a sentence into an average word length value. At that time, it was decided that **Functors** could be used to ensure a simple uniform means to determine a successful or failed execution:

```
let pipeline = pipe(
  validateString,
  splitStringIntoList,
  rejectEmptyStrings,
  mapStringLength,
  averageStringLength
);
let sentence = Identity.from("The quick brown fox jumped over the lazy dog");
pipeline(sentence);
```

This example uses the pipe function we described in *Chapter 15, Understanding Functional Programming*:

```
const pipe = (...fns) => input => fns.reduce((prev, fn) => fn(prev),
input);
```

If the execution is a success, then the resulting value would be an **Identity** instance containing the calculated value. If, however, an error had occurred, then a **Failure** instance would be returned containing the reason for the failure.

Those of you who are astute may have noticed an issue with the tests that were written to describe the `validateString` function and the function's implementation itself. Since the function is designed to form part of a pipeline or compose call, it's possible for a `Failure` instance to be passed in as an argument. This raises a couple of problems:

- Wrapping a `Failure` argument into a new `Failure` instance creates a deeply nested object.
- Creating a new `Failure` instance loses any error information that was sent from previous function invocations.

If you consider the goal of the data transformation, the sentence is input into the first function wrapped in an `IdentityFunctor` and an integer is hopefully expected to return from the last function wrapped in an `IdentityFunctor`.

The following diagram outlines the flow of information through the function pipeline:

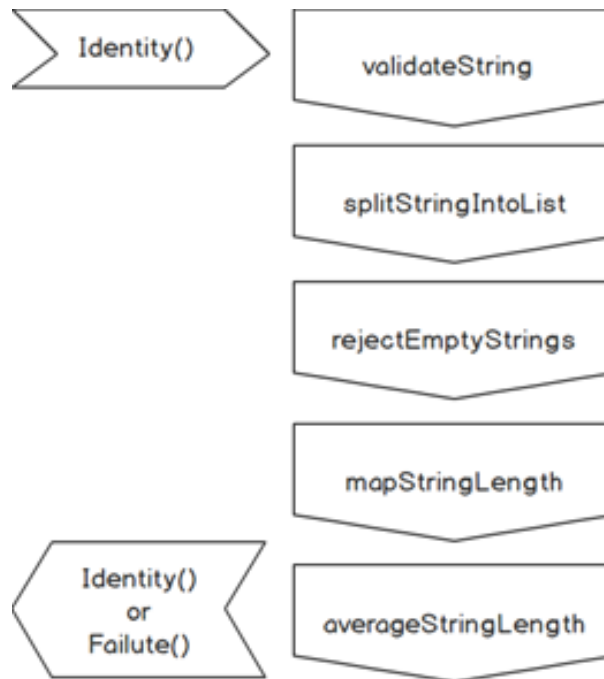


Figure 17.12: Function pipeline

Each function in between passes a **Functor** between them. This is a new **IdentityFunctor** if the previous function succeeded or a **FailureFunctor** representing the first error to occur, as shown in the following diagram:

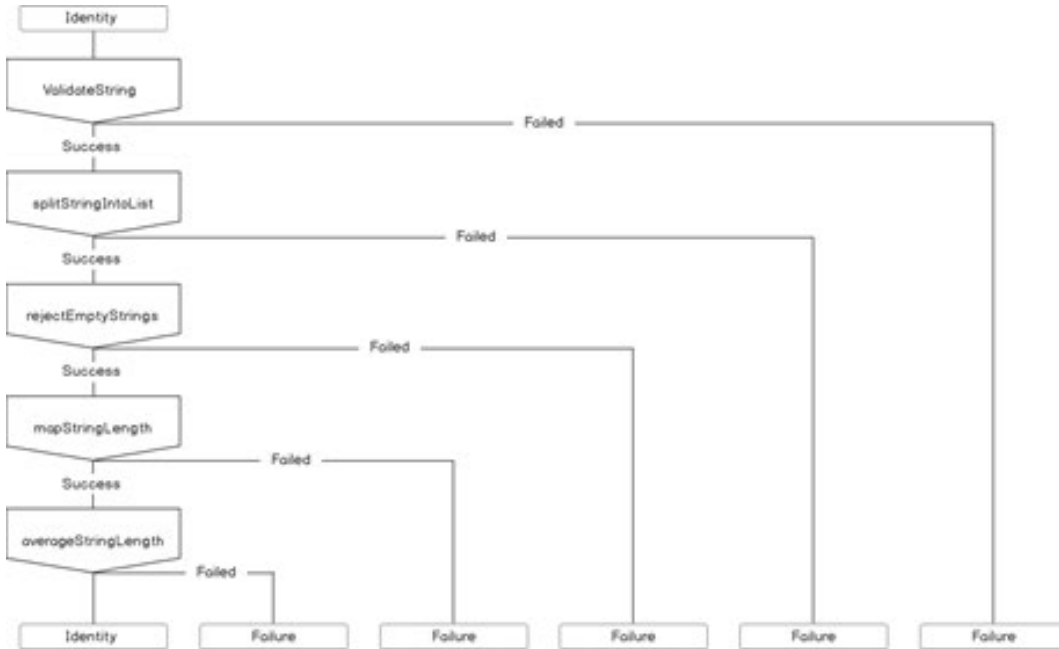


Figure 17.13: Functor flow

In order to handle this correctly, the function should return any **Failure** instance arguments immediately, before any further checks are made. In the case of **validateString**, this would look like the following:

```

let validateString = function(value) {
  if (value instanceof Failure)
    return value;
  if (!(value instanceof Identity))
    return Failure.from("invalid argument");
  if (!(typeof value.get() === "string"))
    return Failure.from("invalid argument");
  return value;
};

```

Then, with this in place, another test is required that ensures that any **Failure** instance arguments are passed cleanly and returned:

```
test('it should return original Failure instance passed in as an argument',  
  () => {  
    let msg = "this is a string";  
    let failure = Failure.from(msg);  
    let res = validateString(failure);  
    expect(res).toBe(failure);  
    expect(res.get()).toBe(msg);  
  });
```

Now, while the changes to the original function and the newly required test are both simple, you may have realized that these same changes will be necessary for all the functions utilizing **Identity** and **FailureFunctors**. If you then have many of these functions in your application, you will find yourself writing all manner of tests that perform the exact same task.

This kind of situation is commonplace when writing tests. However, since the mantra of unit testing is to keep them succinct and easy to read, the Jest framework provides some helpful features aimed at just this type of scenario.

THE MATCHER FORMAT

Jest matchers are simply functions. As an assertion, those functions decide whether data is valid or invalid using a contextual comparison of that data. The simplest matcher, **toBe**, could be described as follows:

```
Object.is(a, b);
```

If **a** is the same as **b**, then the assertion is valid. Otherwise, it is not.

When assertions fail in Jest, a message is output to the console describing the reason for the assertion's failure. It is this message and the notion of a pass or fail that form a minimalist matcher.

As an example, the **toBe** matcher, when reduced to its simplest form, will look as follows:

```
toBe(received, expected) {  
  const pass = Object.is(received, expected);  
  const message = () => `Expected: ${expected}\nReceived: ${received}`  
  return {actual: received, message, pass};  
}
```

The first parameter in the matcher handler is always the value that's passed to the **expect** call. The additional parameters are the arguments to the matcher. After the calculation against each of those values, the function should return an object containing the original value named **actual**, the **pass** state, and the associated **message** as a function. Additional parameters are also available but have been excluded from this example in order to keep it simple.

THE EXTEND FUNCTION

The Jest **expect** object provides a function called **extend**, which allows for new matcher methods to be passed. This function is not a method but is applied directly to **expect**. When called, you supply an object argument containing each of the matcher functions you wish to create. For instance, with regard to the previous **toBe** example, it can be applied like so:

```
expect.extend({
  toBe(received, expected) {
    const pass = Object.is(received, expected);
    const message = () => `Expected: ${expected}\nReceived: ${received}`
    return {actual: received, message, pass};
  }
});
```

EXERCISE 17.06: EXTENDING JEST WITH FUNCTOR MATCHERS

With this knowledge in hand, let's have a go at creating matchers to determine **Identity** and **FailureFunctor** instances. Let's get started:

1. Create a new file in the **src** directory and call it **functor_matchers.js**. Then, at the top of this file, add the following line:

```
import {Identity, Failure} from './functors';
```

This will include the **Functors** we created earlier in this chapter.

2. Next, define a new function for the **toBeIdentity** matcher, which will receive the **received** parameter, but also a value to compare against the Functor's contained value:

```
let toBeIdentity = function(received, expected) {
```


3. Within this function, you'll need to determine whether the assertion passes or not. This will be a two-step process. If the **received** value is an **Identity** instance and the **expected** value is **null**, then the assertion will pass. Also, if the **received** value is an **Identity** instance and also contains **expected**, which is not **null**, then the assertion will pass. Otherwise, it will fail:

```
const maybe_expected = (expected == null || Object.is(received.get(),
expected));
const pass = (received instanceof Identity) && maybe_expected;
```

The **null** and populated **expected** arguments have been covered here so that assertions can be made to validate an **Identity** even when the contained value is irrelevant. As you can see, **pass** is a logical **AND** condition of both cases.

4. Next, you will need to construct the message. This is a little complex as there are three potential outcomes: complete pass, **received** does not contain **expected**, and **received** is not an **Identity**. To decide this, a nested ternary pair will work:

```
const message = () => pass
  ? "Matched Failure with value"
  : (received instanceof Failure)
  ? `Expected Failure with ${expected}, but received Failure with
    ${received.get()}`
  : `Expected Failure but received ${typeof received}`;
```

NOTE

The value is a function. Each of the messages uses the interpolation string type for ease of concatenation.

5. Finally, return the correct object structure, as expected by Jest:

```
return {actual: received, message, pass};
```

6. For the **toBeFailure** matcher, the exact same process must be followed, except with exchanged terms so that it is oriented around the **FailureFunctor**:

```
let toBeFailure = function(received, expected) {
  const maybeExpected = (expected == null || Object.is(received.
get(), expected));
  const pass = (received instanceof Failure) && maybeExpected;
  const message = () => pass
```

```
    ? "Matched Failure with value"
    : (received instanceof Failure)
      ? `Expected Failure with ${expected}, but received Failure
with ${received.get()}`
      : `Expected Failure but received ${typeof received}`;
    return {actual: received, message, pass};
  }
}
```

7. Now, simply export both matchers from the module:

```
export {toBeIdentity, toBeFailure};
```

8. To test these matchers, you'll need to create a test. Open the **sentence.test.js** file and add the following **import** to the top of the file:

```
import {toBeIdentity, toBeFailure} from './functor_matchers';
```

9. Next, create a new test with a couple of sample **functor** instances:

```
test('Exercise 5', () => {
  let id = Identity.from("some string");
  let fail = Failure.from("some failure");
```

10. Now, before the matchers will work, you will need to install them using the **extend** function:

```
expect.extend({toBeIdentity, toBeFailure});
```

11. Then, add tests to validate the sample **functor** instances:

```
expect(id).toBeIdentity();
expect(id).toBeIdentity("some string");
expect(id).not.toBeFailure();
expect(id).not.toBeIdentity("other string");
expect(fail).toBeFailure();
expect(fail).toBeFailure("some failure");
expect(fail).not.toBeIdentity();
expect(fail).not.toBeFailure("other failure");
})
```

The tests simply match the correct Functor type and contents against the instances.

12. If you run these tests now, you should see them pass. You have successfully written your own matchers:

```
jahred@Jahreds-Blade:~/Documents/writing/The-JavaScript-Workshop/Lesson16$ npm run test
> examples@1.0.0 test /home/jahred/Documents/writing/The-JavaScript-Workshop/Lesson16
> jest

WARNING: We noticed you're using the 'useBuiltIns' option without declaring a core-js version. Currently, we assume version 2.x when no version is passed. Since this default version will likely change in future versions of Babel, we recommend explicitly setting the core-js version you are using via the 'corejs' option.

You should also be sure that the version you pass to the 'corejs' option matches the version specified in your 'package.json's 'dependencies' section. If it doesn't, you need to run one of the following commands:

    npm install --save core-js@2    npm install --save core-js@3
    yarn add core-js@2              yarn add core-js@3

PASS src/4-3-14.test.js
  4.3.14 - Custom Matchers
    ✓ Exercise 5 (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.75s, estimated 1s
Ran all test suites.
```

Figure 17.14: Matcher tests passing

MOCKING TESTS

Mocking is a term that's used when elements of your application are replaced with dummy data or functions. This is particularly useful when areas of your application are not pure and, therefore, cannot be cleanly tested. By replacing those effectual elements of your application, you can render them static and therefore accurately reason about the result of your function's processes.

Mocking is also very useful when testing elements of your application that respond to external criteria. For instance, your application may utilize a module that makes calls to a third-party service. Since that service may return different data each time, or even be completely unavailable when testing, providing mocks that simulate the remote service can ensure that your application is accurate.

Jest provides several different types of mocking mechanisms, which will be covered in the remainder of this chapter. These include the following:

- Module bypassing mocks
- ES6 class mocks
- Timer function mocks
- Manual mocking
- Function mocking

Typically, the Jest framework will attempt to automate much of the mocking process, so utilizing the correct mocking approach will save a lot of time and effort.

FUNCTION MOCKS

There are typically three ways to mock functionality in your tests. When creating mocks, you have a choice between ad hoc per-function mocking or automated module mocking and manual module mocking.

A **mocked function** is simply a function that Jest can reason about. For instance, if I were to create a callback, I could create one as a mock by passing it to `jest.fn()`:

```
const callbackFn = () => console.log("I am a callback");
const callbackMock = jest.fn(callbackFn);
```

When converting this into a mock function, Jest applies a **mock** property to it, which stores metadata about the function throughout its lifetime:

```
{calls: [], instances: [], invocationCallOrder: [], results: []}
```

This metadata is useful for analyzing how the function was called throughout a test. The following table describes these properties and their uses:

Properties	Uses
calls	This is an array that stores information about each of the invocations of the mock. Each invocation results in an array that contains the arguments that were sent in the invocation.
instances	This is an array that stores any instances of the function, resulting in objects where the function is used as a constructor.
invocationCallOrder	This is a list of all mock calls relative to the mock function invocation.
results	This is an array providing each of the function's return values for all of its invocations.

Figure 17.15: Mocked function Properties and their Uses

With this data, you can analyze how your application utilized the function in great detail.

Jest also provides a couple of useful functions that can be executed against the mock function that are designed to simulate your mocked function by returning different, but anticipated, values for each set number of invocations. **mockReturnValueOnce** and **mockReturnValue** are called on the mocked function itself prior to being invoked by your application:

```
callbackMock.mockReturnValueOnce('a');
callbackMock.mockReturnValueOnce('b');
callbackMock.mockReturnValue('c');
expect(callbackMock()).toBe('a');
expect(callbackMock()).toBe('b');
expect(callbackMock()).toBe('c');
expect(callbackMock()).toBe('c');
```

mockReturnValue ensures that all the remaining invocations return the same set value, and so should always be utilized last.

NOTE

mockReturnValueOnce is extremely useful when testing a function with a repertoire of possible parameter types. For instance, in the pipeline examples, **mockReturnValueOnce** could be used to test multiple variations of an intermediary pipeline function.

The aforementioned mock functions apply to mocked functions that work synchronously. If you're dealing with an asynchronous function, such as one that returns a **promise** or uses **async await**, then a different set of functions apply. Since an asynchronous function may result in a resolved or rejected promise, Jest provides counterparts for both that resemble those for synchronous functions:

- **mockResolvedValue**
- **mockResolvedValueOnce**
- **mockRejectedValue**
- **mockRejectedValueOnce**

AUTOMATED MODULE MOCKS

Automated mocks refer to the mocking of existing modules, whether these are your own modules or modules that you have included with your project via npm or Yarn.

As when using **jest.fn** to mock functions, module mocking occurs when calling **jest.mock** and passing it the location of the module you want to mock:

```
jest.mock('sentence');
```

Once mocked, all of the functions within that module become generically mocked functions. This means that they are easy to reason about in Jest but will not return the expected results when invoked without a little extra work.

Let's look at this in an example:

```
// async_fun.js
import {Identity, Failure} from './functors';
```

Once the module is imported, you use **asyncFun** as follows:

```
async function asyncFun() {
  const TWO_SECONDS = 2000;
  let oneTwoOrThree = () => Math.floor(Math.random() * 3) + 1; //
function
generates 1, 2 or 3

  let promise = new Promise((resolve, reject) => { // create promise
    setTimeout(() => { // dummy async request function
      let value = oneTwoOrThree(); // get random number
      if (value % 2 === 0) { // if number is even
        reject(Failure.from(value)); // resolve with Failure
      } else {
```

```

        resolve(Identity.from(value)); // otherwise, resolve with
    Identity
    }
    }, TWO_SECONDS); // that delays resolving for 2 seconds
  });

```

You will see that the return value will be a **promise**.

```

    let result = await promise;
    return result;
  }
  export {asyncFun};

```

This module presents an interesting challenge because of its use of **async await**. Now, this particular example is rather contrived, but it does nicely represent the common functions that make HTTP calls to remote services or databases, in particular because of the call to **Math.random()**. This function is both effectual and asynchronous.

asyncFun is problematic for two reasons:

It takes time, beyond computational time, to complete execution.

Its return value is unpredictable.

Writing unit tests for **asyncFun** is not only difficult, but it also increases the duration required to run all tests, thanks to the timer of **asyncFun**. In situations where many functions exist, and each makes delayed asynchronous calls, executing them in sequence could take a very long time indeed.

Thankfully, you don't need to suffer from such problems. By mocking **asyncFun**, you can convert its call into a quick, predictable affair that makes reasoning with its behavior much easier:

```

// async_fun.test.js
import {Identity, Failure} from './functors';
import {toBeIdentity, toBeFailure} from './functor_matchers';
import {asyncFun} from './async_fun';

```

Once the modules are imported, you can mock **asyncFun** as follows:

```

jest.mock('./async_fun');
expect.extend({toBeIdentity, toBeFailure});

test('Automated Module Mocks - Success', () => {
  let failed = Failure.from(0);

```

```
    asyncFun.mockResolvedValue(failed);  
    return asyncFun().then(result => expect(result).toBeFailure(0));  
  })  
  test('Automated Module Mocks - Failure', () => {  
    let success = Identity.from(2);  
    asyncFun.mockResolvedValue(success);  
    return asyncFun().then(result => expect(result).toBeIdentity(2));  
  })
```

Remember that, since **asyncFun** uses **async await**, its return value will be a **promise**. Therefore, the unit tests should deal with it asynchronously too.

MANUAL MODULE MOCKS

Up until now, each time a function needed mocking, it would be defined as such within the test modules themselves. This can prove excessive if the functions you're using should be mocked for all the tests in your application. Remember, TDD demands succinct and concise code. Less is indeed more.

To combat wasteful typing, Jest provides a means to define mocks for modules once, to be used throughout all of your test modules as needed, called manual mocking.

Manual mocks exist as counterpart modules that exist in a directory called **__mocks__** (all lower case). This counterpart directory must sit in the same directory as the module you are mocking. The mock module is then placed into the **__mocks__** directory and is given the same name as the module it mocks. Thus, if your module is called **products.js** and resides in **src/services**, your mocking module will also be called **products.js** and will reside in **src/services/__mocks__**.

If the mocking modules are installed with npm, the mocking modules can be placed in a **__mocks__** directory in the root of the source directory.

NOTE

Note that if a module uses scoping, such as **@babel/some_module**, then the module must sit in an additional directory with the scope name (**__mocks__/@babel/some_module.js**).

If a manual mock module exists for a given module, then calling `jest.mock` against the real module will direct Jest to automatically use the mock module. This saves large amounts of duplicate code from your tests, but also keeps your mocks nicely contained and easy to find.

TIMER FUNCTION MOCKS

Earlier in this chapter, you were shown how to handle asynchronous tests. The example test that was provided used a `setTimeout` call to simulate a callback that may take one second to resolve. Utilizing timers, such as timeouts and intervals, is commonplace in JavaScript applications, but what do you do if a timer is set to call back after a much larger period of time? Certainly, awaiting the response of a 30-minute timeout even for tests would be impractical.

Let's revisit the example:

```
const ONE_SECOND = 1000;
setTimeout(() => {
  expect(true).toBeTruthy();
}, ONE_SECOND);
```

This example becomes asynchronous thanks to the `setTimeout` function call. As we discussed in an earlier chapter, there are four timer-based functions in JavaScript:

- `setTimeout`
- `setInterval`
- `clearTimeout`
- `clearInterval`

Jest provides a means to mock each of these functions in a simple manner using a single static function call:

```
jest.useFakeTimers();
```

By calling this function, the timer functions are swapped out for versions that are immediately assertable. In order to demonstrate this with the preceding example, let's move the callback function into a separate variable:

```
const callback = jest.fn();
setTimeout(callback, 1000);
```

Now, when run, the **setTimeout** function will be executed immediately, but it is not the **setTimeout** function provided by the JavaScript runtime. Instead, it is a function provided by the Jest framework itself.

To query whether **setTimeout** was called, and indeed how it was called, you can simply use function-based matchers, like so:

```
expect(setTimeout).toHaveBeenCalledTimes(1);  
expect(setTimeout).toHaveBeenLastCalledWith(callback, 1000);
```

As well as asserting that **setTimeout** was called with the callback as an argument, you may also want to assert that the callback was then called:

```
expect(callback).toHaveBeenCalledTimes(1);
```

If you run the preceding assertion, however, it will fail. This is because the timer mocks do not execute their callbacks by default. In order to simulate this, you need to request that Jest run all the mock timers. Do this with the following statement:

```
jest.runAllTimers();
```

Now, if you run the same assertion, everything should pass correctly.

ACTIVITY 17.02: SENTENCE PARSING TEST SUITE

For this activity, imagine that you have been tasked with implementing the complete functional "average word length in a sentence" implementation, as described at the beginning of this chapter. This process must utilize the functional pipe and include the following functions:

- **validateString**
- **splitStringIntoList**
- **rejectEmptyStrings**
- **mapStringLength**
- **averageStringLength**

The aim of this task is to follow TDD methodologies. This means that you must reason about what tests each function requires, how they can be described, and how to correctly validate the function tersely using assertions. Try to imagine all the permutations for using each function, but then round those permutations down to their simplest forms.

To aid in this task, the following table describes what each function should take as a value (forgetting about the **functors** that wrap them for the moment) and what they should return:

Function	Arguments	Returns
<code>validateString</code>	String	String
<code>splitStringIntoList</code>	String	Array<String>
<code>rejectEmptyStrings</code>	Array<String>	Array<String>
<code>mapStringLength</code>	Array<String>	Array<Integer>
<code>averageStringLength</code>	Array<Integer>	Integer

Figure 17.16: Functions and their return value

Remember that each function should receive these values wrapped in a **Functor** and must return a value wrapped in a **Functor**. If you would like a tougher challenge, then consider how you could augment each function to accept a raw value as an alternative argument. Functions that can interchange between raw and **Functor** arguments must still return a **Functor**.

When writing tests for each function, consider which tests are common for all functions. Can they be combined into a custom matcher?

Finally, once all the functions have been tested and defined, consider unit tests that may facilitate a function that implements the entire pipeline. What additional tests are required? What similar tests are required? Try to cover every usage possibility for all the functions.

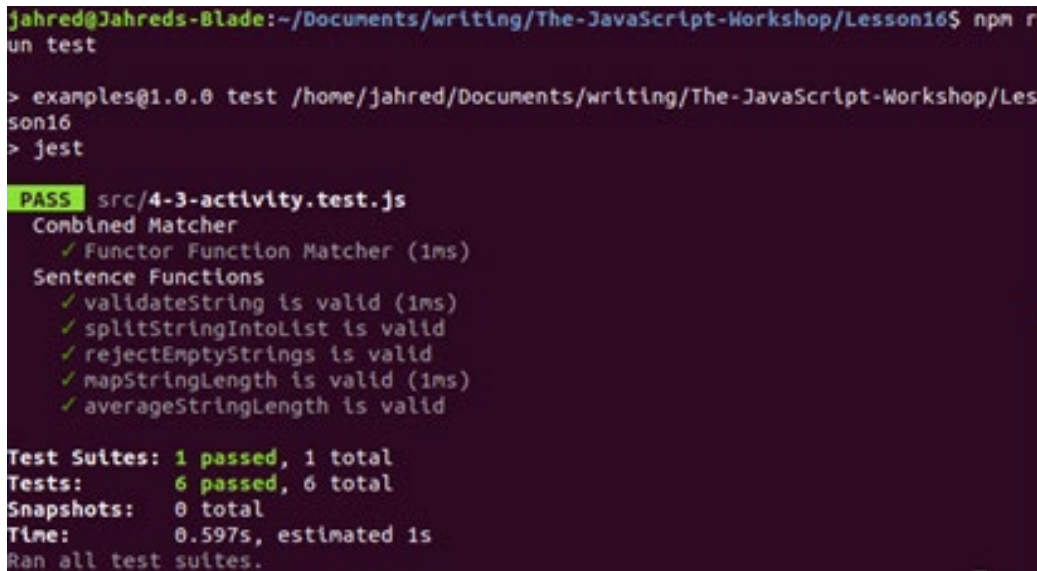
Once the tests have been completed, start writing the functions themselves. You should aim to get each test passing in sequence. Simply run the tests, read the first error presented, and resolve it. You can work on each function individually if it helps keep the task uncluttered.

Steps:

1. Identify the test requirements of each function. What would be a passing condition? What would constitute failing conditions?
2. Based on commonalities between the conditions, extend Jest with any required matchers to ensure brevity when writing the tests.
3. Begin writing the tests. Working with each pipeline function in turn, create any passing tests, then any failure-oriented tests. Try to be thorough but refrain from condition duplication.

4. With the tests written, run all the tests and see which ones fail.
5. Begin populating the pipeline functions with functionality that fulfills the test requirements only. No code should be written that does not have a test for it. The code should be as concise as it can be.
6. Repeat steps 4 and 5 until all the tests pass.

The expected solution for this activity is as follows:



```
jahred@Jahreds-Blade:~/Documents/writing/The-JavaScript-Workshop/Lesson16$ npm run test
> examples@1.0.0 test /home/jahred/Documents/writing/The-JavaScript-Workshop/Lesson16
> jest

PASS src/4-3-activity.test.js
  Combined Matcher
    ✓ Functor Function Matcher (1ms)
  Sentence Functions
    ✓ validateString is valid (1ms)
    ✓ splitStringIntoList is valid
    ✓ rejectEmptyStrings is valid
    ✓ mapStringLength is valid (1ms)
    ✓ averageStringLength is valid

Test Suites: 1 passed, 1 total
Tests: 6 passed, 6 total
Snapshots: 0 total
Time: 0.597s, estimated 1s
Ran all test suites.
```

Figure 17.17: Activity test run

SUMMARY

This has been a packed chapter. In a short space of time, you have understood why testing is important, but also have learned about philosophies that allow tests to underpin your software design. In turn, you have learned how to write more concise functions that facilitate a single specific need and nothing more.

Using the Jest framework, you have dissected a problem, defined its requirements, and leveraged assertions outlining the confines of the problem to realize pure, reasonable functions that are persistently predictable.

Later in the chapter, you discovered how to reduce testing complexity to its simplest form by keeping to concise test conditions that are easy to understand and maintain.

As the final bonus chapter, this completes your journey through modern JavaScript development, priming you with all you need to create great-looking software.

