Chapter No. 2
"Your First Spec"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.2 "Your First Spec"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Paulo Ragonha is a software engineer. He loves web development for the opportunities that it carries; "to be able to craft a piece of software that can be instantly accessible by anyone" (with internet connection, of course).

In his early days of software development, he was mostly involved in game development and Java. But since his discovery of Ruby and JavaScript, he has worked uniquely on web applications.

His last three projects were big JavaScript applications, developed entirely driven by tests and with amazing tooling support. He has an amazing wife that he loves very much, lives in the beautiful Florianópolis, a coast city in the south of Brazil. He is a casual speaker, a biker, a runner, and a hobbyist photographer (he has earned an actual award taking pictures).

For More Information:
www.packtpub.com/jasmine-javascript-testing/book
I would like to thank Juliana, my wife, for supporting me during the writing of this book. It was not easy for her to have her husband almost unavailable for over two months. I would also like to thank my family for cheering me when I needed the most, and my friends, who were there for me, helping me boot up this book. I would specially like to thank Felipe Munhoz who was always available to hear me whinging about my texts, and Fabiano Soriani who replied to every single e-mail I've sent to him about the queries regarding the book. Thank you all!
Jasmine JavaScript Testing

This book is about being a better JavaScript developer. So throughout the chapters, the reader will not only learn about writing tests in the Jasmine “idiom”, but also about the best practices on writing a software in JavaScript language. It is about acknowledging JavaScript as a real platform for application development, and leveraging all its potential. It is about tooling and automation, and how to make your life easier and more productive.

Most importantly, this book is about craftsmanship in producing not only working software, but also well-crafted software.

The *Jasmine JavaScript Testing* book is a practical guide to writing and automating JavaScript testing for web applications. It uses technologies such as Jasmine, Sinon.JS, RequireJS, and Grunt.

Over the course of the chapters, the concept of test-driven development is explained through the development of a simple Stock Market Investment Tracker Application. It starts with the basics of testing through the development of the base domain classes (such as Stock and Investment), passing through the concepts of maintainable browser code, and concluding with a full refactoring to a Backbone.js application with RequireJS dependency management, and automated build.

**What This Book Covers**

*Chapter 1, Getting Started with Jasmine*, covers the motivations behind testing a JavaScript application. It presents the concept of BDD and how it helps you to write better tests. It also demonstrates how easy it is to download Jasmine and start coding your first test.

*Chapter 2, Your First Spec*, helps you learn the thought process behind thinking in test-driven development. You will code your very first JavaScript functionality driven by tests. You will also learn the basic functions of Jasmine, and how to structure your tests. It also demonstrates how Jasmine matchers work, and how you can create one of your own to improve your tests’ code readability.

*Chapter 3, Testing Frontend Code*, covers some patterns in writing maintainable browser code. You will learn about thinking in components, and how to use the Module Pattern to better organize your source files. You will also be presented with the concept of HTML fixtures, and how you can use it to test your JavaScript code without requiring your servers to render a HTML. You will also learn about a Jasmine plugin called Jasmine jQuery, and how it can help you write better tests with jQuery.

**For More Information:**
Chapter 4, *Asynchronous Testing—AJAX*, talks about the challenges in testing AJAX requests, and how you can use Jasmine to test any asynchronous code. You will learn about NodeJS, and how to create a very simple HTTP server to use as a fixture to your tests.

Chapter 5, *Jasmine Spies*, presents the concept of test doubles and how to use Spies to perform behavior checking.

Chapter 6, *Light Speed Unit Testing*, helps you learn about the issues with AJAX testing, and how you can make your tests run faster by using Stubs or Fakes (such as SinonJS Fake Server).

Chapter 7, *Testing Backbone.js Applications*, presents the four base abstractions of Backbone.js, and how you can write tests to a Backbone.js application. You will learn about what to test, and not to test, when using Backbone.js.

Chapter 8, *Build Automation*, presents you the power of automation. You will learn about RequireJS, and how it helps you organize the dependencies of large JavaScript applications. You will start to think in modules and their dependencies, and you will learn how to code your tests as modules. You will also learn about packing and minifying the code to production, and how to automate this process. Then you are going to use PhantomJS to run all your tests without a browser window, and even how to make them run automatically on any file change.

Chapter 9, *Conclusion*, talks about the future of testing and tooling in the JavaScript ecosystem.


For More Information:  
www.packtpub.com/jasmine-javascript-testing/book
Your First Spec

This chapter is about the basics. We are going to guide you through how to write your first spec, thinking in test first, and also show all the available global Jasmine functions. By the end of the chapter, you should know how Jasmine works and be ready to start doing your first tests by yourself.

The Investment Tracker application

To get you started, we need an example scenario: consider you are developing an application to track investments in the stock market.

The following form illustrates better how a user might create a new investment on this application:

<table>
<thead>
<tr>
<th>Symbol:</th>
<th>Shares:</th>
<th>Share price:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PETO</td>
<td>100</td>
<td>35</td>
</tr>
</tbody>
</table>

Form to add investments

This form will allow the input of three values that define an investment:

- A Symbol which represents what company (stock) the user is investing in
- How many Shares the user has bought (or invested in)
- How much the user has paid for each share (the Share price)

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

For More Information:

www.packtpub.com/jasmine-javascript-testing/book
Your First Spec

If you are unfamiliar with how the stock market works, imagine you are shopping for groceries. To make a purchase you must specify what you are buying, how many items you are buying, and how much you are going to pay. These concepts translate to an investment as:

- A stock which is defined by a symbol, such as PETO, can be understood to be a grocery type
- The number of shares is the quantity of items you have purchased
- The share price is the unit price of each item

Once the user has added an investment, it must be listed along with his or her other investments.

<table>
<thead>
<tr>
<th>Symbol:</th>
<th>Shares:</th>
<th>Share price:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOUE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PETO</td>
<td>8</td>
<td>-42.34%</td>
</tr>
</tbody>
</table>

Form and list of investments

The idea is to display how well his or her investments are going. Since the prices of the stocks fluctuate over time, the difference between the price the user has paid and the current price, indicate whether it is a good (profit) or a bad (loss) investment.

In the preceding figure, we can see that the user has two investments:

- One in the AOUE stock, which is scoring a profit of 101.80%
- And another in the PETO stock, which is scoring a loss of -42.34%

This is a very simple application and we will get a deeper understanding of its functionality as we go on with its development.

Jasmine basics and thinking in BDD

Based on the application presented previously, we can start writing acceptance criteria that define investment:

- Given an investment, it should be of a stock
- Given an investment, it should have the invested shares quantity

For More Information:  
www.packtpub.com/jasmine-javascript-testing/book
• Given an investment, it should have the share paid price
• Given an investment, it should have a cost

And to start coding these as a Jasmine spec, the first thing we need to do, is to create a new spec file. This file can be created anywhere, but it is a good idea to stick to a convention, and Jasmine already has a good one: specs should be in the /spec folder. Create a spec/InvestmentSpec.js file and add the following lines:

```javascript
describe("Investment", function() {
});
```

`describe` is a global Jasmine function used to define test contexts. When used as the first call in a spec, it creates a new test suite. It accepts two parameters:

• The name of the test suite, in this case, `Investment`
• A functions which will contain all its specs

Then to translate the first acceptance criterion (given an investment, it should be of a stock) into a Jasmine spec, we are going to use another global Jasmine function called `it`:

```javascript
describe("Investment", function() {
    it("should be of a stock", function() {

    });
});
```

It also accepts two parameters:

• The title of the spec, in this case, `should be of a stock`
• A function, which will contain the spec code

To run this spec, add it to the runner:

```html
<!-- include spec files here... -->
<script type="text/javascript" src="spec/InvestmentSpec.js"></script>
```

And execute it by opening the runner on the browser:

![First spec's passing result on the browser](image-url)
Your First Spec

This might sound strange to have an empty spec passing, but in Jasmine, as with other test frameworks, a failed assertion is required to make the spec fail.

An assertion is a comparison between two values that must result in a Boolean value. The assertion is only considered a success if the result of the comparison is true.

In Jasmine, assertions are written by using the global Jasmine function `expect`, along with a matcher that indicates what comparison must be made with the values.

Regarding the current spec (it is expected that the investment is of a stock), in Jasmine this translates into:

```javascript
describe("Investment", function() {
  it("should be of a stock", function() {
    expect(investment.stock).toBe(stock);
  });
});
```

The `expect` function takes only one parameter, which defines the actual value, or in other words, what is going to be tested: `investment.stock` and expects the chaining call to a matcher function: in this case `toBe`. That defines the expected value: `stock`, and the comparison method to be performed (to be the same).

Behind the scenes, Jasmine makes a comparison to check if the actual value (`investment.stock`) and expected value (`stock`) are the same, and if they are not, the test fails.

With the assertion written, the previously passing test is now failing:

```
Failing 1 spec  No try/catch
1 spec | 1 failing

  Investment should be of a stock.
  ReferenceError: investment is not defined
```

This spec is failing because, as the error message states: `investment is not defined`.
Chapter 2

The idea here is to do only what the error is indicating us to do, so although you might feel the urge to write something else, for now let's just create this investment variable with an Investment instance.

```javascript
describe("Investment", function() {
  it("should be of a stock", function() {
    var investment = new Investment();
    expect(investment.stock).toBe(stock);
  });
});
```

Don't worry that the Investment() function doesn't exist yet, the spec is about to ask for it on the next run:

![Failing 1 spec](image)

Spec asks for an Investment class

You can see that the error has changed to **Investment is not defined**. It now asks for the Investment function. So create a new file Investment.js in the src folder and add it to the runner:

```html
<!--[include source files here... -->
<script type="text/javascript" src="src/Investment.js"></script>
```

To define the Investment, write a constructor function inside the src/Investment.js file:

```javascript
function Investment () {};
```

This makes the error change. It now complains about the missing stock variable:

![Failing 1 spec](image)

Missing stock error

For More Information:  
www.packtpub.com/jasmine -javascript-testing/book
Your First Spec

One more time, we feed the code for which it is asking:

```javascript
describe("Investment", function() {
  it("should be of a stock", function() {
    var stock = new Stock();
    var investment = new Investment();
    expect(investment.stock).toBe(stock);
  });
});
```

And the error changes again, this time it is the missing `Stock` function:

Create a new file called `src/Stock.js`, and add it to the runner. Since the `Stock` function is going to be a dependency of `Investment`, we should add it just before `Investment`:

```html
<!-- include source files here... -->
<script type="text/javascript" src="src/Stock.js"></script>
<script type="text/javascript" src="src/Investment.js"></script>
```

And write the `Stock` constructor function:

```javascript
function Stock () {};
```

Finally the error is on the expectation:

```
Failing 1 spec No try/catch
1 spec | 1 failing
Investment should be of a stock.
ReferenceError: Stock is not defined
```

Expected undefined to be `Stock`
To fix this and complete this exercise, open the src/Investment.js file and add the reference to the stock parameter:

```javascript
function Investment (stock) {
    this.stock = stock;
};
```

And on the spec, pass stock as a parameter to Investment:

```javascript
describe("Investment", function() {
    it("should be of a stock", function() {
        var stock = new Stock();
        var investment = new Investment(stock);
        expect(investment.stock).toBe(stock);
    });
});
```

Finally you shall have a passing test:

![Passing Investment spec](image)

This exercise was meticulously conducted, to show how a developer works by feeding the spec with what it wants, when doing test-first development.

The drive to write code must come from a failing spec. You must not write code, unless its purpose is to fix a failing spec.

**Setup and teardown**

There are still three more acceptance criteria to be implemented. The next in the list is:

"Given an investment, it should have the invested shares quantity."
Writing it should be as simple as the previous spec was. In the spec/InvestmentSpec.js file, you can translate this new criterion into a new spec called should have the invested shares quantity:

```javascript
describe("Investment", function() {
  it("should be of a stock", function() {
    var stock = new Stock();
    var investment = new Investment({
      stock: stock,
      shares: 100
    });
    expect(investment.stock).toBe(stock);
  });

  it("should have the invested shares quantity", function() {
    var stock = new Stock();
    var investment = new Investment({
      stock: stock,
      shares: 100
    });
    expect(investment.shares).toEqual(100);
  });
});
```

You can see that apart from having written the new spec, we have also refactored the call to the Investment constructor, to support the new shares parameter.

To do so, we used an object as a single parameter in the constructor, to simulate named parameters, a feature JavaScript doesn't have natively.

To implement this in the Investment function is pretty simple: instead of having multiple parameters on the function declaration, it instead has only one, which is expected to be an object. Then, the function probes each of its expected parameters from this object, making the proper assignments:

```javascript
function Investment (params) {
  var params = params || {};
  this.stock = params.stock;
}
```

For More Information:
www.packtpub.com/jasmine-javascript-testing/book
The code is now refactored. We can run the tests to see that only the new spec should be failing:

```
function Investment (params) {
    var params = params || {}; 
    this.stock = params.stock;
    this.shares = params.shares;
};
```

And finally everything is green:

```
var stock = new Stock();
var investment = new Investment({
    stock: stock,
    shares: 100
});
```

But as you can see, the code that instantiates the Stock and the Investment is duplicated on both specs:

```
var stock = new Stock();
var investment = new Investment({
    stock: stock,
    shares: 100
});
```

To eliminate this duplication, Jasmine provides another global function called beforeEach that, as the name states, is executed once before each spec. So for these two specs, it will run twice—once before each spec.

For More Information:

www.packtpub.com/jasmine-javascript-testing/book
Refactor the previous specs, by extracting the setup code with the use of `beforeEach`:

```javascript
describe("Investment", function() {
  var stock, investment;

  beforeEach(function() {
    stock = new Stock();
    investment = new Investment({
      stock: stock,
      shares: 100
    });
  });

  it("should be of a stock", function() {
    expect(investment.stock).toBe(stock);
  });

  it("should have the invested shares quantity", function() {
    expect(investment.shares).toEqual(100);
  });
});
```

Much cleaner, we not only removed the code duplication, but also simplified the specs. They became much easier to read and maintain, since their only responsibility now is to perform the expectation.

There is also a teardown function (`afterEach`) that sets code to be executed after each spec. It is very useful in situations where a cleanup is required after each spec. We will see an example of its application in Chapter 6, Light Speed Unit Testing.

To finish the specification of `Investment`, add the remaining two specs to the `spec/InvestmentSpec.js` file:

```javascript
describe("Investment", function() {
  var stock;
  var investment;

  beforeEach(function() {
    stock = new Stock();
    investment = new Investment({
      stock: stock,
      shares: 100,
      sharePrice: 20
    });
  });

  it("should have the invested shares quantity", function() {
    expect(investment.shares).toEqual(100);
  });
});
```
Chapter 2

it("should have the share payed price", function() {
    expect(investment.sharePrice).toEqual(20);
});

it("should have a cost", function() {
    expect(investment.cost).toEqual(2000);
});

Run the specs to see them failing:

<table>
<thead>
<tr>
<th>Failing 2 specs</th>
<th>No try/catch</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 specs</td>
<td>2 failing</td>
</tr>
<tr>
<td>Investment should have the share payed price.</td>
<td>Expected undefined to equal 20.</td>
</tr>
<tr>
<td>Investment should have a cost.</td>
<td>Expected undefined to equal 2000.</td>
</tr>
</tbody>
</table>

Add the code to fix them in the src/Investment.js file:

function Investment (params) {
    var params = params || {};
    this.stock = params.stock;
    this.shares = params.shares;
    this.sharePrice = params.sharePrice;
    this.cost = this.shares * this.sharePrice;
};

Run the specs for the last time to see them green:

<table>
<thead>
<tr>
<th>Passing 4 specs</th>
<th>No try/catch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investment</td>
<td>should be of a stock</td>
</tr>
<tr>
<td>should have the invested shares quantity</td>
<td></td>
</tr>
<tr>
<td>should have the share payed price</td>
<td></td>
</tr>
<tr>
<td>should have a cost</td>
<td></td>
</tr>
</tbody>
</table>

Passing all four Investment specs

For More Information:
www.packtpub.com/jasmine-javascript-testing/book
It is important to always see a spec failing before writing the code to fix it, otherwise how would you know that you really fix it? Imagine this as a way to test the test.

**Nested describes**

**Nested describes** are useful when you want to describe similar behavior between specs. Suppose we want the following two new acceptance criteria:

- Given an investment, when its stock share price valorizes, it should have a positive return of investment (ROI);
- Given an investment, when its stock share price valorizes, it should be a good investment.

They both share the same behavior **when its stock share prices valorizes**.

To translate this into Jasmine you can nest a call to the `describe` function inside the existing one in the `spec/InvestmentSpec.js` file: (I removed the rest of the code for the purpose of demonstration; it is still there.)

```javascript
describe("Investment", function()
    describe("when its stock share price valorizes", function() {

    });
})
```

It should behave just like the outer one, so you can add specs (\textit{it}) and use setup and teardown functions (\textit{beforeEach, afterEach}).

**Setup and teardown**

When using setup and teardown functions, Jasmine respects the outer setup and teardown functions as well, so they are run as expected. For each spec (\textit{it}):

- Jasmine runs all setup functions (\textit{beforeEach}) from the outside in
- Runs a spec code (\textit{it})
- Runs all the teardown functions (\textit{afterEach}) from the inside out

For More Information:

So we can add a setup function to this new `describe` that changes the share price of the stock, so it’s greater than the share price of the investment:

```javascript
describe("Investment", function() {
    var stock;
    var investment;

    beforeEach(function() {
        stock = new Stock();
        investment = new Investment({
            stock: stock,
            shares: 100,
            sharePrice: 20
        });
    });

    describe("when its stock share price valorizes", function() {
        beforeEach(function() {
            stock.sharePrice = 40;
        });
    });
});
```

Coding a spec with shared behavior

Now that we have the shared behavior implemented, we can start coding the acceptance criteria described earlier. Each is, just as before, a call to the global Jasmine function `it`:

```javascript
describe("Investment", function() {
    describe("when its stock share price valorizes", function() {
        beforeEach(function() {
            stock.sharePrice = 40;
        });

        it("should have a positive return of investment", function() {
            expect(investment.roi()).toEqual(1);
        });

        it("should be a good investment", function() {
            expect(investment.isGood()).toBeTruthy();
        });
    });
});
```
Your First Spec

And after adding the missing functions to Investment in src/Investment.js:

```javascript
Investment.prototype.roi = function() {
    return (this.stock.sharePrice - this.sharePrice) / this.sharePrice;
};

Investment.prototype.isGood = function() {
    return this.roi() > 0;
};
```

You can run the specs, and see that they are passing:

![Passing specs](image)

Understanding matchers

By now, you've already seen plenty of usage examples for matchers, and probably can feel how they work.

You have seen how to use the `toBe`, the `toEqual`, and `toBeTruthy` matchers. These are a few of the built-in matchers available in Jasmine, but we can extend Jasmine by writing matchers of our own.

So, to really understand how Jasmine matchers work, we need to create one ourselves.

Custom matchers

Consider this expectation from the previous section:

```javascript
expect(investment.isGood()).toBeTruthy();
```

Although it works, it is not very expressive. Imagine if we could rewrite it instead, as:

```javascript
expect(investment).toBeAGoodInvestment();
```

For More Information:
www.packtpub.com/jasmine-javascript-testing/book
This makes a much better relation to the acceptance criterion:

"should be a good investment"→expect investment to be a good investment

And to implement it is quite simple. You do so by calling the `this.addMatcher` Jasmine function inside a setup (`beforeEach`) or a spec (`it`).

Although you can put this new matcher definition inside the `spec/InvestmentSpec.js` file, Jasmine already has a default place to add custom matchers, the file `spec/SpecHelper.js`. If you are using the Standalone Distribution, it already comes with a sample custom matcher; delete it and let's start from scratch.

The `addMatcher` function accepts a single parameter—an object where each attribute corresponds to a new matcher. So to add this new matcher, change the contents of the `spec/SpecHelper.js` file to:

```javascript
beforeEach(function() {
    this.addMatchers({
        toBeAGoodInvestment: function() {}
    });
});
```

A Jasmine matcher is simply a function that returns a Boolean value: `true`, to indicate that the expectation is passing and `false` if otherwise.

But to implement this matcher, we need access to the `investment` object, available via the `this.actual` property:

```javascript
toBeAGoodInvestment: function() {
    var investment = this.actual;
    return investment.isGood();
};
```

After getting access to the investment object, implementing the matcher was a simple return of the `isGood()` value.

By now, this matcher is ready to be used by the specs:

```javascript
it("should be a good investment", function() {
    expect(investment).toBeAGoodInvestment();
});
```

After the change, the specs should still be passing. But what happens if a spec fails? What is the error message that Jasmine reports?
We can see it by deliberately breaking the `Investment.isGood` implementation in `src/Investment.js`, to always return `false`:

```javascript
Investment.prototype.isGood = function() {
    return false;
};
```

When running the specs again, this is the error message that Jasmine generates:

```
Investment when its stock share price valorizes should be a good investment.
Expected { stock: { sharePrice: 40 }, shares: 100, sharePrice: 20, cost: 2000 } to be a good investment.
```

It is not so bad, but sure can be made better. Jasmine allows the customization of this message via the `this.message` property inside the matcher declaration. Jasmine expects this property to be a function that returns the error message:

```javascript
toBeAGoodInvestment: function() {
    var investment = this.actual;

    this.message = function() {
        return 'Expected investment to be a good investment';
    };

    return investment.isGood();
}
```

Run the specs again and the error message should change:

```
Investment when its stock share price valorizes should be a good investment.
Expected investment to be a good investment.
```

Now, let's consider another acceptance criterion:

"Given an investment, when its stock share price devalorizes, it should be a bad investment".
Although it is possible to create a new custom matcher (`toBeABadInvestment`), Jasmine allows the negation of any matcher by chaining `not` before the matcher call. So we can write that a "bad investment" is "not a good investment":

```
expect(investment).not.toBeAGoodInvestment();
```

Implement this new acceptance criterion in `spec/InvestmentSpec.js` by adding a new nested describe and spec:

```
describe("when its stock share price devalorizes", function() {
  beforeEach(function() {
    stock.sharePrice = 0;
  });

  it("should be a bad investment", function() {
    expect(investment).not.toBeAGoodInvestment();
  });
});
```

But there is a catch! Let's break the `Investment` code so that it is always a good investment:

```
Investment.prototype.isGood = function() {
  return true;
};
```

After running the specs again, you can see that this new spec is failing, but the error message is wrong: **Expected investment to be a good investment.**

That is the message that was hard coded inside the matcher. To fix this, you need to make the message dynamic, based on how the matcher was called.

Luckily there is a property available inside the matcher declaration that tells if the matcher was called with a chained `not`, `this.isNot`.

For More Information:

Your First Spec

Following is the fixed matcher with the dynamic message:

```javascript
toBeAGoodInvestment: function() {
  var investment = this.actual;
  var what = this.isNot ? 'bad' : 'good';

  this.message = function() {
    return 'Expected investment to be a ' + what + ' investment';
  };

  return investment.isGood();
}
```

This fixes the message:

```
Investment when its stock share price devalorizes should be a bad investment.

Expected investment to be a bad investment
```

And now this matcher can be used anywhere.

What lacked in this example was a way to show how to pass an expected value to a matcher like this:

```javascript
expect(investment.cost).toEqual(2000)
```

It turns out that a matcher can receive any number of expected values as parameters. So for instance, the preceding matcher could be implemented as:

```javascript
toEqual: function(expectedValue) {
  return this.actual == expectedValue;
};
```

Always before implementing any matcher, check first if there is one available that already does what you want.

## Built-in matchers

Jasmine comes with a bunch of default matchers, covering the basis of value checking in the JavaScript language. To understand how they work and where to use them properly is a journey on how JavaScript handles type.

For More Information:

www.packtpub.com/jasmine-javascript-testing/book
The toEqual built-in matcher
This is probably the most commonly used matcher, and you should use it whenever you want to check equality between two values.

It works for all primitive values (number, string, and Boolean) as well as any object (including arrays).

```javascript
describe("toEqual", function() {
  it("should pass equal numbers", function() {
    expect(1).toEqual(1);
  });

  it("should pass equal strings", function() {
    expect("testing").toEqual("testing");
  });

  it("should pass equal booleans", function() {
    expect(true).toEqual(true);
  });

  it("should pass equal objects", function() {
    expect({a: "testing"}).toEqual({a: "testing"});
  });

  it("should pass equal arrays", function() {
    expect([1, 2, 3]).toEqual([1, 2, 3]);
  });
});
```

The toBe built-in matcher
The toBe matcher has a very similar behavior to the toEqual matcher, in fact it gives the same result while comparing primitive values, but the similarities stop there.

While the toEqual matcher has a complex implementation (you should take a look at the Jasmine source code) that checks if all attributes of an object and all elements of an array are the same, here it is a simple use of the strict equals operator (===).

If you are unfamiliar with the strict equals operator, its main difference from the equals operator (==) is that the latter performs type coercion if the compared values aren't of the same type.
The strict equals operator always considers false any comparison between values of distinct types.

Here are some examples of how this matcher (and the strict equals operator) works:

describe("toBe", function() {
  it("should pass equal numbers", function() {
    expect(1).toBe(1);
  });

  it("should pass equal strings", function() {
    expect("testing").toBe("testing");
  });

  it("should pass equal booleans", function() {
    expect(true).toBe(true);
  });

  it("should pass same objects", function() {
    var object = {a: "testing"};
    expect(object).toBe(object);
  });

  it("should pass same arrays", function() {
    var array = [1, 2, 3];
    expect(array).toBe(array);
  });

  it("should not pass equal objects", function() {
    expect({a: "testing"}).not.toBe({a: "testing"});
  });

  it("should not pass equal arrays", function() {
    expect([1, 2, 3]).not.toBe([1, 2, 3]);
  });
});

It is advised to use the `toEqual` operator in most cases, and resort to the `toBe` matcher only when you want to check if two variables reference the same object.

For More Information:
www.packtpub.com/jasmine-javascript-testing/book
The `toBeTruthy` and `toBeFalsy` matchers

Besides its primitive Boolean type, everything else in the JavaScript language also has an inherent Boolean value, which is generally known as being either **truthy** or **falsy**.

Luckily in JavaScript, there are only a few values that are identified as falsy, as shown in the following examples for the `toBeFalsy` matcher:

```javascript
describe("toBeFalsy", function () {
  it("should pass undefined", function () {
    expect(undefined).toBeFalsy();
  });

  it("should pass null", function () {
    expect(null).toBeFalsy();
  });

  it("should pass NaN", function () {
    expect(NaN).toBeFalsy();
  });

  it("should pass the false boolean value", function () {
    expect(false).toBeFalsy();
  });

  it("should pass the number 0", function () {
    expect(0).toBeFalsy();
  });

  it("should pass an empty string", function () {
    expect("").toBeFalsy();
  });
});
```

Everything else is considered truthy, as demonstrated by these examples of the `toBeTruthy` matcher:

```javascript
describe("toBeTruthy", function () {
  it("should pass the true boolean value", function () {
    expect(true).toBeTruthy();
  });

  it("should pass any number different than 0", function () {
    expect(1).toBeTruthy();
  });
});
```

For More Information:
www.packtpub.com/jasmine-javascript-testing/book
Your First Spec

```
it("should pass any non empty string", function() {
    expect("a").toBeTruthy();
});

it("should pass any object (including an array)", function() {
    expect([]).toBeTruthy();
    expect({}).toBeTruthy();
});
```

But if you want to check if something is equal to an actual Boolean value, it might be a better idea to use the `toEqual` matcher.

The `toBeUndefined`, `toBeNull`, and `toBeNaN` built-in matchers

These matchers are pretty straightforward, and should be used to check for undefined, null, and NaN values.

```
describe("toBeNull", function() {
    it("should pass null", function() {
        expect(null).toBeNull();
    });
});

describe("toBeUndefined", function() {
    it("should pass undefined", function() {
        expect(undefined).toBeUndefined();
    });
});

describe("toBeNaN", function() {
    it("should pass NaN", function() {
        expect(NaN).toBeNaN();
    });
});
```

Both `toBeNull` and `toBeUndefined` can be written as `toBe(null)` and `toBe(undefined)` respectively, but that is not the case with `toBeNaN`.

In JavaScript, the NaN value is not equal to any value, not even NaN. So trying to compare it to itself is always false.

```
NaN === NaN // false
```

For More Information:
The Jasmine `toBeNaN` matcher is actually implemented taking into account that `NaN` is the only value that is not equal to itself. Here is how it is implemented:

```javascript
jasmine.Matchers.prototype.toBeNaN = function() {
  return (this.actual !== this.actual);
};
```

See how it checks if the actual is different from itself? So as a good practice, try to use these matchers whenever possible instead of their `toBe` counterparts.

### The `toBeDefined` built-in matcher

This matcher is useful if you want to check whether a variable is defined and you don't care about its value.

```javascript
describe("toBeDefined", function() {
  it("should pass any value other than undefined", function() {
      expect(null).toBeDefined();
  });
});
```

Anything except `undefined` will pass under this matcher, even `null`.

### The `toContain` built-in matcher

Sometimes it is desirable to check if an array contains an element or if a string can be found inside another string. For these use cases, you can use the `toContain` matcher.

```javascript
describe("toContain", function() {
  it("should pass if a string contains another string", function() {
      expect("My big string").toContain("big");
  });

  it("should pass if an array contains an element", function() {
      expect([1, 2, 3]).toContain(2);
  });
});
```
The toMatch built-in matcher

Although the toContain and the toEqual matchers can be used in most string comparisons, sometimes the only way to assert if a string value is correct, is through a regular expression. For these cases, you can use the toMatch matcher along with a regular expression.

```javascript
describe("toMatch", function() {
  it("should pass a matching string", function() {
    expect("My big matched string").toMatch(/My(.+)string/);
  });
});
```

The matcher works by testing the actual value ("My big matched string") against the expected regular expression (/My(.+)string/).

The toBeLessThan and toBeGreaterThan built-in matchers

The toBeLessThan and toBeGreaterThan are two simple matchers to perform numeric comparisons, as best described by the following examples:

```javascript
describe("toBeLessThan", function() {
  it("should pass when the actual is less than expected", function() {
    expect(1).toBeLessThan(2);
  });
});
```

```javascript
describe("toBeGreaterThan", function() {
  it("should pass when the actual is greater than expected", function() {
    expect(2).toBeGreaterThan(1);
  });
});
```

The toThrow built-in matcher

Exceptions are a language's way to demonstrate when something goes wrong.

So for example while coding an API, you might decide to throw an exception when a parameter is passed incorrectly. So how do you test this code?

Jasmine has the built-in toThrow matcher that can be used to verify that an exception has been thrown.

For More Information:
www.packtpub.com/jasmine-javascript-testing/book
The way it works is a little bit different from the other matchers. Since the matcher has to run a piece of code and check if it throws an exception, the matcher's actual value must be a function.

Here is an example of how it works:

```javascript
describe("toThrow", function() {
  it("should pass when the exception is thrown", function() {
    expect(function () {
      throw("Some exception");
    }).toThrow("Some exception");
  });
});
```

When the test is run, the anonymous function gets executed, and if it throws the Some exception exception, the test passes.

**Summary**

In this chapter, you have learned how to think in Behavior Driven Development (BDD) and drive your code from your specs. You have also become acquainted with the basic Jasmine global functions (describe, it, beforeEach, and afterEach) and got a good understanding of what is required to create a spec in Jasmine.

You have got familiar with Jasmine matchers, and know how they are powerful in describing a spec intent. You have even learned to create a matcher of your own.

By now you should be familiar with creating new specs and driving the development of your new application.

In the next chapter, we are going to take a look on how we can use the concepts learned in this chapter to start testing web applications: most commonly jQuery and HTML forms.
Where to buy this book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.