Jasmine Cookbook

Jasmine provides a rich set of libraries to design and develop tests for JavaScript applications. Its tests aim to run on any JavaScript-enabled platform and have easy-to-read syntax.

This book is a comprehensive guide to designing and developing Jasmine tests to produce world-class software. The recipes are elaborated using examples from a real-world application, which involves various aspects of designing and developing tests from simple to complex level. You will learn how to create tests for jQuery and Ajax, HTML and JSON fixtures, CoffeeScript, and AngularJS. By learning and applying the best practices throughout this book, you can deliver your applications with zero defects and ensure success for you and your clients.

What this book will do for you...

- Develop JavaScript applications with Jasmine using the behavior-driven development (BDD) process
- Apply custom matchers by enhancing your Jasmine tests to test specific features or functionality
- Design Jasmine specs for jQuery and Ajax with HTML and JSON fixtures
- Generate code coverage analysis for JavaScript code using Karma and Istanbul
- Create Jasmine tests for CoffeeScript and AngularJS
- Implement E2E (end-to-end) Jasmine specs for JavaScript applications to simulate a real-time scenario

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Jasmine Cookbook

Over 35 recipes to design and develop Jasmine tests to produce world-class JavaScript applications

Munish Sethi

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Customizing Matchers and Jasmine Functions'
- A synopsis of the book’s content
- More information on Jasmine Cookbook

About the Author

**Munish Sethi** is a postgraduate in computer science and MBA in IT from SCDL. He has 18 years of industry experience and has a sound knowledge of test automation, test- and behavior-driven development, white box testing, big data testing, review techniques, test estimation, and requirement analysis. Currently, he is working with Impetus Infotech Pvt. Ltd. where he plays the role of test craftsman and automation architect. With his diverse experience, he helps engineering teams with deliveries and also with designing test solutions and formulating test strategies for his clients. He has also designed and implemented many automation frameworks (such as modular, hybrid, data-driven, keyword-driven, and page-object models) using Jasmine/Protractor, QTP, Selenium WebDriver, Junit, TestNG, SOAPUI, Sikuli, Test Complete, and so on. He's very keen to explore new and upcoming tools and technologies.

He has worked with both product- and services-based organizations. Munish's Twitter handle is @munishksethi.
Jasmine Cookbook

Today, application quality is a major concern among organizations. Lack of quality not only affects one's credibility but is also detrimental for end users and business operations. It is very important to develop applications by applying best practices and effective processes, and exercising the entire code base before deploying it on the production.

Jasmine is an open source Behavior-Driven Development (BDD) framework for testing JavaScript applications. It encourages good testing practices and application development using a BDD approach. Jasmine provides a rich set of libraries to design and develop tests for JavaScript (or JavaScript-enabled platform) applications. The entire application code can be exercised to ensure the quality and correctness of functional behavior. Jasmine also provides the mechanism to define custom functions and spies to test end-to-end (E2E) scenarios.

This book is written to get you up and running with Jasmine faster than any other book. Each chapter focuses on a specific aspect of Jasmine and is broken down into easy-to-follow recipes. While most chapters make use of previous skills that you'll have learnt from earlier the text, you don't necessarily need to read each chapter in order. Each recipe begins with the explicit steps that you need to accomplish the goal of the recipe, followed by an explanation as well as some other ideas you may wish to explore on your own.

Lastly, let's focus on building high-quality apps with Jasmine to minimize defect leakage in production. All you need is this book, a computer with Jasmine, and a constructive mind.

What This Book Covers

Chapter 1, Getting Started with Jasmine Framework, introduces the basics of Jasmine, demonstrates its usage, and explains how unit tests will be designed for JavaScript. This also discusses the role of matchers for writing Jasmine specs.

Chapter 2, Jasmine with TDD and BDD Processes, describes how Jasmine tests will be designed along with application code using TDD and BDD processes, and also explains how to write Jasmine specs for new and existing code by applying TDD and BDD processes.

Chapter 3, Customizing Matchers and Jasmine Functions, elaborates Jasmine functions and the need of custom matchers and their usage. This also examines the concept of pending specs and designing Jasmine specs for exceptions.

Chapter 4, Designing Specs from Requirement, explains how meaningful specs can be designed by determining test requirements.
Chapter 5, Jasmine Spies, illustrates the concept of mocking and how to create Jasmine tests for methods or objects which are dependent on other methods or objects. This also demonstrates how to develop custom or user-defined mock functions.

Chapter 6, Jasmine with AJAX, jQuery, and Fixtures, elaborates how to design and develop a Jasmine spec for AJAX calls, jQuery, and asynchronous operations. This also examines the data-driven approach and how to design Jasmine specs with fixtures to manipulate DOM.

Chapter 7, Code Coverage with Jasmine Tests, covers how to configure code coverage tools and generate coverage reports using karma and istanbul, and it also explores how to increase code coverage for the existing or legacy code.

Chapter 8, Jasmine with Other Tools, describes how to implement Jasmine specs for AngularJS, CoffeeScript, and Node.js.

Chapter 9, Developing JavaScript Apps Using Jasmine - A Real-time Scenario, elaborates a real-time scenario to develop JavaScript applications using Jasmine and explains how to design specs with HTML using BDD approach, and also examines how to develop specs for a Change Request and validate the output of a method, object, or unit using the data-driven approach.
### Customizing Matchers and Jasmine Functions

In this chapter, we will cover:

- Writing a Jasmine test using a custom equality tester function
- Writing Jasmine tests with Custom Matchers
- Writing Jasmine tests for exceptions
- Declaring Pending specs with Jasmine tests

## Introduction

A matcher is used to evaluate an assertion of an object. If the expected and actual values match, the assertion passes, or else, the assertion fails. Jasmine provides a rich set of built-in matchers to test JavaScript applications. To know more about Jasmine's built-in matchers, refer to the Applying different matchers to the Jasmine test recipe in Chapter 1, Getting Started with Jasmine Framework. In addition to the built-in matchers, Jasmine also provides various functions and enables you to create custom matchers to test JavaScript code. Custom matchers help to document the intent of the spec and can also help to remove code duplication across specs. In this chapter, we will investigate the need of such custom matchers and find out how to develop these in order to validate a specific test condition or scenario. In addition, you will learn how to test the scenarios that expect JavaScript exceptions to be thrown.
**Writing a Jasmine test using a custom equality tester function**

You can customize how Jasmine determines whether two objects are equal by defining your own custom equality tester. A custom equality tester is a function, which takes two arguments. This function returns either `true` or `false` based on the arguments. Also, if it is unable to compare the argument values, then the custom equality tester function will return `undefined`.

In this recipe, you will learn how to compare the equality of two objects using such custom equality tester functions.

Let's consider a scenario where the equality of two strings should be decided using a custom equality tester function.

"As a JavaScript developer, I want to develop a mechanism to check the equality of two strings so that I can compare and categorize objects."

**Scenario-1**: Two strings should be equal if the first word of both strings is `Packt`

**How to do it...**

You need to perform the following steps:

1. First, you need to create a spec file (`custom_equality_spec.js`) under the `/spec` folder and code the following lines to define and implement the spec for scenario 1:
   ```javascript
   describe("Custom Equality Function: ", function() {
       describe("Strings: ", function(){
           it("should be custom equal if first word of both the strings is Packt", function() {
               expect("Packt Jasmine Book").toEqual("Packt Java Cookbook");
           });
       });
   });
   ```
2. Run the spec file custom_equality_spec.js. You will see that the Jasmine test fails, as shown in the following screenshot:

![Jasmine Test Failure](image)

3. Now, let's define the `MyCustomEquality` custom equality tester function within the spec file `custom_equality_spec.js` to determine the equality of two strings using the following code:

```javascript
var MyCustomEquality = function(strFirstString, strSecondString) {
  if (typeof strFirstString == "string" && typeof strSecondString == "string") {
    return strFirstString.substr(0,5) == strSecondString.substr(0,5);
  }
};
```

The `MyCustomEquality` function receives the first argument (that is, **Actual Value**) as the value passed to `expect()` and the value passed to the `toEqual` matcher itself as the second argument or **Expected Value**.

4. Next, use the following code to implement scenario 1 using a custom equality tester function:

```javascript
describe("Custom Equality Function: ", function() {
  beforeEach(function() {
    jasmine.addCustomEqualityTester(MyCustomEquality);
  });
  describe("Strings: ", function(){
    it("should be custom equal if first word of both the strings is Packt", function() {
      expect("Packt Jasmine Book").toEqual("Packt Java Cookbook");
    });
  });
});
```
Customizing Matchers and Jasmine Functions

Here, note how we registered the `MyCustomEquality` custom equality tester function inside the `beforeEach` function.

Once a custom equality tester function is registered in the `beforeEach` function, it will first be checked before the default equality logic.

5. Finally, run the spec file `custom_equality_spec.js` with the Jasmine runner (that is, `SpecRunner.html`). You will see that the Jasmine test passes:

```
Jasmine 2.0.1  finished in 0.001s

1 spec, 0 failures  raise exceptions

Custom Equality Function:
Strings:
  should be custom equal if first word of both the strings is 'Packt'
```

How it works...

In steps 1 and 2, we implemented scenario 1 using the `toEqual` matcher.

In step 3, we defined the `MyCustomEquality` custom equality tester function to determine the equality of two strings.

In steps 4 and 5, we implemented scenario 1 using the custom equality tester function.

Writing Jasmine tests with custom matchers

Jasmine's built-in matchers are very useful and work well in most scenarios. However, sometimes we need to develop custom matchers to implement specific scenarios. For example, we need to create the custom matchers to override the failure message or to define a new message (or custom error message). Also, a custom matcher can be created when you find a piece of code that tests the same set of properties over and over again across the different specs, and you want to bundle the fragment into a single assertion. By writing your own matcher, you eliminate code duplication and make tests more readable. In other words, by defining your own matchers, you develop a Domain-Specific Language (DSL) to describe the intent of tests in a more readable and expressive way.
To write a Jasmine test using custom matchers, let’s assume that the task is to develop an online ordering module for <XYZ> company’s website. There are some business rules to fill the online order.

"As an administrator, I want to validate all the business rules on submission of an online order so that I can process orders"

Let's consider some scenarios, that is, all business rules should be validated before accepting the online order:

- **Scenario-1**: Person's age should be greater than or equal to 21
- **Scenario-2**: Person’s first and last names are mandatory to process the online order

Now, to understand the concept of custom matchers, consider that whenever a Jasmine test fails, a user-defined error message should be displayed for both scenarios, as follows:

- **Error message for Scenario-1**: Minimum person’s age should be 21 years to place the order online
- **Error message for Scenario-2**: Person's first name and last name are mandatory to process the online order

### How to do it...

You need to perform the following steps to write Jasmine tests using custom matchers:

1. Create a spec file Custom_Matcher_spec.js under the /spec folder and code the following lines to define a spec for scenario 1:

   ```javascript
   describe("<XYZ> Company: Online Order Module", function() {
       describe("When to place the online order: " , function() {
           it("Age should be greater than " + "or equal to 21 years", function() {
               ...
           });
       });
   });
   ```
2. Now, run the spec file Custom_Matcher_spec.js with the Jasmine runner (that is, SpecRunner.html). You will see that an empty spec (that is, a spec without any expectation) passes:

```javascript
describe("<XYZ> Company: Online Order Module", function() {
    describe("When to place the online order: ", function(){
        beforeEach(function() {
            jasmine.addMatchers(personAgeValidationMatcher);
        });
        it("Age should be greater than or equal to 21 years", function() {
            // Add scenario 1 code here
        });
    });
});
```

3. Next, let's add the reference of the custom matcher to display the error message for scenario 1 using the following code:

```javascript
describe("<XYZ> Company: Online Order Module", function() {
    describe("When to place the online order: ", function(){
        beforeEach(function() {
            jasmine.addMatchers(personAgeValidationMatcher);
        });
        it("Age should be greater than or equal to 21 years", function() {
            // Add scenario 1 code here
        });
    });
});
```

Note, how we added the reference of a custom matcher (personAgeValidationMatcher) to the beforeEach function.

Before applying a custom matcher to any expectation, its reference needs to be added in the beforeEach function.
4. Run the spec file Custom_Matcher_spec.js. You will see that the spec fails in the absence of the `personAgeValidationMatcher` custom matcher function:

```
Jasmine 2.0.1

1 spec, 1 failure

Spec List | Failures

<XYZ> Company: Online Order Module when to place the online order: Age should be greater than or equal to 21 years

ReferenceError: 'personAgeValidationMatcher' is undefined
```

5. As indicated in the previous step, let's define the custom matcher `personAgeValidationMatcher` within the spec file `Custom_Matcher_spec.js` using the following code:

```javascript
var personAgeValidationMatcher = {
  toBeOlderThan: function() {
    return {
      compare: function(actualAge, expectedAge) {
        if (expectedAge === undefined) {
          throw "Expected value is required";
        }
        if (actualAge>=expectedAge) {
          return {
            pass: true
          };
        } else {
          return {
            pass: false
          };
        }
      };
    };
  }
};
```

Here, note that we created the `toBeOlderThan` method to validate a person's age within the `personAgeValidationMatcher` custom matcher. Also, a `compare` function is created with two parameters.
The compare function receives the first argument as **Actual Value**, which is passed to `expect()` and it receives the second argument as **Expected Value**, which is passed to the matcher itself.

The compare function returns a result object with a property called `pass` that is the Boolean result of matcher. The `pass` property tells the expectation whether the matcher is successful (`true`) or unsuccessful (`false`).

6. Now, let's create the `Person.js` file under the `/src` folder and define the `Person()` object to get one's age and name. Use the following code to define the `Person()` object:

   ```javascript
   var Person = function (age, firstName, lastName) {
     this.age = age;
     this.firstName = firstName;
     this.lastName = lastName;
   };
   ```

7. Next, use the following code to implement scenario 1:

   ```javascript
   describe("<XYZ> Company: Online Order Module", function() {
     describe("When to place the online order: ", function() {
       beforeEach(function() {
         jasmine.addMatchers(personAgeValidationMatcher);
       });
       it("Age should be greater than " + "or equal to 21 years", function() {
         var myPerson = new Person(25, "James", "Smith");
         expect(myPerson.age).toBeOlderThan(20);
       });
     });
   });
   ```

Here, you can observe that we are using the `toBeOlderThan()` function to validate a person's age, which is defined within the custom matcher (personAgeValidationMatcher).
8. Now, add the reference of the `Person.js` file to the Jasmine runner and run the spec file `Custom_Matcher_spec.js`. You will see that the spec passes:

```
jasmine.addMatchers(personAgeValidationMatcher);
```

9. Before implementing the custom failure message for scenario 1, let's use the following code to fail it:

```javascript
describe("<XYZ> Company: Online Order Module", function() {
    describe("When to place the online order: ", function()
    {
        beforeEach(function()
        {
            jasmine.addMatchers(personAgeValidationMatcher);
        });
        it("Age should be greater than " + "or equal to 21 years", function()
        {
            var myPerson = new Person(18, "James", "Smith");
            expect(myPerson.age).toBeOlderThan(20);
        });
    });
});
```

10. Run the spec file `Custom_Matcher_spec.js` with the Jasmine runner. You should see the following failure message:

```
Expected 18 to be older than 20.
```

In the preceding screenshot, you can see that the expectation generated an error message for the matcher: **Expected 18 to be older than 20.**
In the absence of a user-defined error message, the expectation will generate a default error message for the matcher.

11. Now, let's implement a user-defined error message for scenario 1 using the following code:

```javascript
var personAgeValidationMatcher = {
  toBeOlderThan: function() {
    return {
      compare: function(actualAge, expectedAge) {
        if (expectedAge === undefined) {
          throw "Expected value is required";
        }
        if (actualAge>=expectedAge) {
          return {
            pass: true,
            message: "Person is eligible to place online order"
          };
        } else {
          return {
            pass: false,
            message: "Minimum person's age should be 21 years to place the order online"
          };
        }
      }
    }
  }
};
```

In the preceding code snapshot, notice how we defined a custom message using the `message` property.
12. Run the spec file `Custom_Matcher_spec.js`. You will see a user-defined failure message, as shown in the following screenshot:

![Jasmine 2.0.1](image)

1 spec, 1 failure

<table>
<thead>
<tr>
<th>Spec List</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;XYZ&gt; Company: Online Order Module When to place the online order: Age should be greater than or equal to 21 years</td>
<td></td>
</tr>
<tr>
<td>Minimum person's age should be 21 years to place the order online</td>
<td></td>
</tr>
</tbody>
</table>

In the preceding screenshot, you can see that a user-defined message is displayed:

**Minimum person's age should be 21 years to place order online.**

13. Next, to implement scenario 2, let's create a custom matcher (`personNameValidationMatcher`) within the spec file `Custom_Matcher_spec.js` using the following code:

```javascript
var personNameValidationMatcher = {
  toContainFirstAndLastName: function() {
    return {
      compare: function(actual) {
        if (actual.firstName != undefined && actual.lastName != undefined) {
          return {
            pass: true,
            message: "Person is eligible to place the online order"
          };
        } else {
          return {
            pass: false,
            message: "First name and last name are mandatory to process the online order"
          };
        }
      }
    }
  }
};
```
Customizing Matchers and Jasmine Functions

Here, you can observe that we created the `toContainFirstAndLastName` method to validate the person’s full name. Also, we defined a custom message using the `message` property. This will be displayed if the Jasmine test fails.

14. Now, following the same pattern, let’s define and implement the spec for scenario 2 using the following code:

```javascript
describe("<XYZ> Company : Online Order Module ", function() {

describe("When to place the online order: ", function(){

    beforeEach(function() {
        jasmine.addMatchers(personAgeValidationMatcher);
        jasmine.addMatchers(personNameValidationMatcher);
    });

    //Scenario 1
    it("Age should be greater than " + "or equal to 21 years", function() {
        var myPerson = new Person(25, "James", "Smith");
        expect(myPerson.age).toBeOlderThan(20);
    });

    //Scenario 2
    it("First Name and Last Name are required " + "to place the online order", function() {
        var myPerson = new Person(23, "James", "Smith");
        expect(myPerson).toContainFirstAndLastName();
    });
});
```  

In the preceding code snapshot, you will observe that we added the reference to the custom matcher (that is, `personNameValidationMatcher`) inside the `beforeEach` function.

15. Finally, run the spec file `Custom_Matcher_spec.js`. You will see that the specs pass for both the scenarios:

```

```

2 specs, 0 failures

<XYZ> Company : Online Order Module

  When to place the online order:
    Age should be greater than or equal to 21 years
    First Name and Last Name are required to place the online order

```
How it works...

In steps 1 to 15, we looked at how Jasmine tests are developed using custom matchers. We also saw how the custom messages are generated when the test fails.

Let's take a look at what we did throughout this recipe.

In steps 1 to 8, we implemented scenario 1 using custom matchers. In step 3, we added the reference to the custom matcher (personAgeValidationMatcher) using the beforeEach function. In step 5, we created the personAgeValidationMatcher custom matcher in the spec file Custom_Matcher_spec.js. In step 6, to implement the functionality of scenario 1, we defined the Person object. In step 7, we implemented specs for scenario 1.

In steps 9 to 12, we implemented a user-defined failure message for scenario 1 using a custom matcher.

In steps 13 to 15, following the same methodology, we defined and implemented the specs for scenario 2.

Writing Jasmine tests for exceptions

In this recipe, you will learn to write Jasmine tests for JavaScript exceptions using the throwError() and throwErrorError() matchers. Also, you will learn how to write Jasmine tests to validate user-defined messages.

To write the Jasmine test for Exceptions, let's assume that you are developing an application for <ABC> company. Currently, you are developing an error handling module, which handles user-defined or system errors.

"As a JavaScript developer, I want to handle all the errors and validate error messages so that the end user should not get stuck during any process."

Let's consider some scenarios in the current context, that is, all the system and user-defined errors should be handled throughout the application:

- **Scenario-1:** Error should be thrown on any unexpected behavior or malfunction.
- **Scenario-2:** Error messages should be consistent throughout the application.
Customizing Matchers and Jasmine Functions

How to do it...

You need to perform the following steps to write Jasmine tests for exceptions:

1. Create a spec file Exception_spec.js under the /spec folder and code the following lines to define the spec for scenario 1:

   ```javascript
   describe("JavaScript Exceptions: ", function(){
       describe("Validate Errors:", function(){
           it("Error should be thrown on any " +
               "unexpected behavior or malfunctioning", function(){
               });
           });
       });
   });
   ```

2. Run the spec file Exception_spec.js. You will see that an empty spec passes, as shown in the following screenshot:

   ![Screenshot of Jasmine tests](image)

3. Create the Exception.js file and put it under the /src folder. Using the following code:

   ```javascript
   function addTwoNumbers(num1, num2){
       return num1 + num2;
   }
   function generateMaxValue(maxValue){
       var intVar=maxValue;
       if (intVar===undefined){
           throw new ParameterException("No value is assigned to variable intVar!");
       }
       intVar=intVar+1;
       if (maxValue<1 || maxValue>1000){
   ```
throw new ArgumentOutOfRangeException("Maximum value should be between 0 and 1000");
}
return intVar;
};

function ParameterException(message) {
    this.name = "ParameterException";
    this.message = message || "Default Message";
}
ParameterException.prototype = Object.create(Error.prototype);
ParameterException.prototype.constructor = ParameterException;

function ArgumentOutOfRangeException(message) {
    this.name = "ArgumentOutOfRangeException";
    this.message = message || "Default Message";
}
ArgumentOutOfRangeException.prototype = Object.create(Error.prototype);
ArgumentOutOfRangeException.prototype.constructor = ArgumentOutOfRangeException;

You can see in the preceding code snapshot that an error will be thrown along with a user-defined error message if the intVar variable is undefined or the value of the maxValue argument is not between 0 and 1000.

4. Now, use the following test code to implement scenario 1:

describe("JavaScript Exceptions: ", function(){
    describe("Validate Errors:", function(){
        it("Error should be thrown on any " +
            "unexpected behavior or malfunctioning", function(){
            expect(addTwonumbers).not.toThrow();
            expect(generateMaxValue).toThrow();
            expect(generateMaxValue).toThrowError();
        });
    });
});

In the preceding code snapshot, you can see that both functions addTwonumbers and generateMaxValue are used without parenthesis. It means that we are not passing any arguments to both functions.
5. Next, to test the functions’ arguments, let's implement the code for scenario 1 along with function parameters, using the following code:

```javascript
describe("JavaScript Exceptions: ", function(){
    describe("Validate Errors:", function(){
        //Scenario 1
        it("Error should be thrown on any " +
           "unexpected behavior or malfunctioning", function(){
            expect(addTwonumbers).not.toThrow();
            expect(generateMaxValue).toThrow();
            expect(generateMaxValue).toThrowError();
        });
        it("Error should be thrown on passing any " +
           "unexpected or wrong arguments", function(){
            /*Assertions to test parameters
             * of addTwonumbers() function*/
            expect(addTwonumbers.bind(null,1,2)).not.toThrow();
            expect(addTwonumbers.bind(null,1,2)).not.toThrowError();
            expect(function(){addTwonumbers(1,4);}).not.toThrow();
            /*Assertions to test parameters
             * of generateMaxValue() function*/
            expect(generateMaxValue.bind(null,5)).not.toThrow();
            expect(generateMaxValue.bind(this,5)).not.toThrow();
            expect(function() {generateMaxValue(2000);}).toThrow();
            expect(function() {generateMaxValue(2000);}).toThrowError();
        });
    });
});
```

Here, you can see that we passed the arguments in two ways – using the `bind` function and prefixed `function()` to test (AddTwonumbers and GenerateMaxValue).

To validate parameters of a function, you can pass the arguments with the `bind` function. This function was introduced in ECMAScript 5. The first parameter needs to be passed as `null` or `this`. Thereafter, you can pass the actual parameter(s) of the function to be tested.

Another way to pass the arguments is simply to prefix `function()` with the function to be tested.
6. Now, add the reference of the Exception.js file to the Jasmine runner (that is, SpecRunner.html) and run the spec file Exception_spec.js. You will see that both the specs pass for scenario 1:

```
Jasmine 2.0.1

2 specs, 0 failures

JavaScript Exceptions:
Validate Errors:
  Error should be thrown on any unexpected behavior or malfunctioning
  Error should be thrown on passing any unexpected or wrong arguments
```

7. Let's use the following code to implement scenario 2:

```javascript
describe("Validate Error Messages: ", function(){
  it("Error messages should be consistent " + 
    "throughout the application", function(){
      expect(generateMaxValue).toThrow(new Error("No value is 
        assigned to variable intVar!"));
      expect(generateMaxValue).toThrowError("No value is 
        assigned to variable intVar!");
      expect(generateMaxValue.bind(null,5000)).toThrow(new 
        Error("Maximum value should be between 0 and 1000");
      expect(generateMaxValue.bind(null,5000)).
       toThrowError("Maximum value should be between 0 and 1000");
     toThrowError("Maximum value should be between 0 and 1000");
      expect(function() {generateMaxValue(5000);}).toThrowError("Maximum value should be between 0 and 1000");
    });
});
```

In the preceding code snapshot, you will observe that we are using the toThrow and toThrowError matchers to validate user-defined error messages.
8. Now, to execute both scenarios, run the spec file `Exception_spec.js`. You will see that both scenarios are pass:

```
Jasmine 2.0.1  finished in 0.009s
...
3 specs, 0 failures
```

9. Next, let's fail scenario 2 by passing a message, which is different from the JavaScript code, using the following code:

```javascript
describe("Validate Error Messages: ", function(){
    it("Error messages should be consistent " +
        "throughout the application", function(){
            expect(generateMaxValue).toThrow(new Error("No value is assigned to variable intVar!"));
            expect(generateMaxValue).toThrowError("No value is assigned to variable intVar!");
            expect(generateMaxValue.bind(null,5000)).toThrow(new Error("Maximum value should be between 0 and 1000"));
            expect(generateMaxValue.bind(null,5000)).toThrowError("Maximum value should be between 0 and 1000");
            expect(function() {generateMaxValue(5000);}).toThrowError("Value should be between 0-1000");
        });
});
```

In the preceding test code, you can see that the error message has been changed. Now, the message is **Value should be between 0 and 1000** instead of **Maximum value should be between 0 and 1000**, which is defined with JavaScript code in the `Exception.js` file.
10. Run the spec file `Exception_spec.js` with the Jasmine runner. You will see that the spec fails for scenario 2:

![Screenshot](image)

You will observe in the preceding screenshot that the test fails because of inconsistent error messages.

**How it works...**

In steps 1 to 10, we saw how to write Jasmine tests to validate JavaScript exceptions/errors using the `toThrow` and `toThrowError` matchers. We also looked at how to write tests to validate error messages.

Let's take a look at the steps of this recipe.

In steps 1 to 2, we defined the empty spec for scenario 1.

In step 3, to write Jasmine tests for exception, we developed JavaScript code in the `Exception.js` file. Here, we defined two functions: `AddTwoNumbers()` and `GenerateMaxValue(maxValue)`. Also, to handle argument errors for the `GenerateMaxValue` function, we defined two functions (that is, `ParameterException` and `ArgumentOutOfRangeException`) along with user-defined error messages.

In steps 4 to 6, we implemented scenario 1 to test JavaScript exceptions using the `toThrow` and `toThrowError` matchers. In step 5, we implemented the code to test the function's arguments using the `bind` function. Also, we passed the parameter (that is, `maxValue`) value of more than 1000 to pass the test.

In steps 7 to 8, we implemented scenario 2 to validate the consistency of messages throughout the application.

In steps 9 to 10, we implemented test code for scenario 2 to make it fail. Here, we saw how a Jasmine test fails because of inconsistent messages defined across JavaScript code and Jasmine specs.
Furthermore, I would suggest that you change the error message defined in step 10 as per step 7 (that is, Maximum value should be between 0 and 1000) and run the spec file with the Jasmine runner to see the test results.

It's good practice to create Jasmine tests corresponding to exceptions to maintain the consistency of error messages.

### Declaring pending specs with Jasmine tests

Sometimes, the specific functionality of an application gets changed or becomes obsolete due to new business rules or change requests raised by stakeholders. In this situation, such legacy Jasmine tests also becomes obsolete and there is a need to remove them from the existing suites. In this recipe, you will learn the usage of pending specs and how to handle the specs that get changed or obsolete.

#### Getting ready

You will learn this recipe with the help of the second recipe in this chapter, *Writing a Jasmine test with Custom Matchers*.

To understand the concept of pending specs, consider that a few of the business rules have been changed for both the scenarios described in the second recipe. Now, let's consider the following scenarios as per the new business rules:

- **Scenario-1**: Age is not a criteria to place the online order
- **Scenario-2**: Only the first name is required to place the online order

#### How to do it...

You need to perform the following steps to declare pending specs with Jasmine tests:

1. First, you need to create a spec file (`Pending_spec.js`) under the `/spec` folder and get the following code from the spec file (`Custom_Matcher_spec.js`) created in the second recipe of this chapter, *Writing a Jasmine test with Custom Matchers*:

   ```javascript
   describe("<XYZ> Company : Online Order Module ", function() {
     describe("When to place the online order: ", function(){
       beforeEach(function() {
         jasmine.addMatchers(personAgeValidationMatcher );
         jasmine.addMatchers(personNameValidationMatcher);
       });
   });
   ```
//Scenario 1
it("Age should be greater than " +
   "or equal to 21 years", function() {
     var myPerson = new Person(25, "James", "Smith");
     expect(myPerson.age).toBeOlderThan(20);
   });

//Scenario 2
it("First Name and Last Name are required " +
   "to place the online order", function() {
     var myPerson = new Person(23, "James", "Smith");
     expect(myPerson).toContainFirstAndLastName();
   });

2. As per new business rules, a person's age is not a criteria and only the first name is required to process the online order. Let's use the following code to implement scenario 1 and scenario 2:

describe("<XYZ> Company : Online Order Module ", function() {
  describe("When to place the online order: ", function(){
    beforeEach(function() {
      jasmine.addMatchers(personAgeValidationMatcher );
      jasmine.addMatchers(personNameValidationMatcher);
    });

    //Scenario 1
    xdescribe("Given: Age is not a " +
           "criteria to place online order", function(){
      it("Age should be greater than " +
         "or equal to 21 years", function() {
             var myPerson = new Person(25, "James", "Smith");
             expect(myPerson.age).toBeOlderThan(20);
           });
    });

    //Scenario 2
    xit("First Name and Last Name are required " +
          "to place the online order", function() {
            var myPerson = new Person(23, "James", "Smith");
            expect(myPerson).toContainFirstAndLastName();
        });
  });
});
You can see in the highlighted code that we created a new `xdescribe` block to implement scenario 1.

In Jasmine, if any suite is declared with `xdescribe`, then specs created inside the suite are skipped. Also, when the suite is executed, it will not appear in the test results.

Furthermore, we assumed that JavaScript code is not developed corresponding to scenario 2 and marked the spec as pending using `xit`. Now, the spec will be ignored during the test execution.

If any spec is declared with `xit`, then it is marked as pending. However, it will appear in the test results as a pending spec. It is good practice to mark all the specs as pending until the code is not refactored as per a new business rule or change request.

3. Next, run the spec file `Pending_spec.js`. You will see that only one spec is executed instead of two and it is marked as pending:

```
3. Next, run the spec file Pending_spec.js. You will see that only one spec is executed instead of two and it is marked as pending:

4. Let's use the following code for scenario 2 to mark the spec as pending using different options provided by Jasmine:

```
it("First Name and Last Name are required " +
"to place the online order", function() {
  pending();
  var myPerson = new Person(23, "James", "Smith");
  expect(myPerson).toContainFirstAndLastName();
});

You will observe in the preceding code snapshot that we declared pending specs using different options.

In Jasmine, a spec is considered as pending in the following situations:

- If the spec is declared with `xit`
- If the `pending()` function is called within the spec
- If the `it` function is declared without a function body

5. Run the spec file `Pending_spec.js`. You will see that three specs ran out of four and all three specs are marked as pending:

```
Jasmine 2.0.1

Ran 3 of 4 specs - run all
3 specs, 0 failures, 3 pending specs
```

Now, you know how and why to declare the pending specs. Furthermore, I would suggest that you implement the code for scenario 2 as per the new business rule (that is, only a first name is required to place the online order).
Where to buy this book
You can buy Jasmine Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.