INSTANT

Short | Fast | Focused

Mock Testing with PowerMock

Discover unit testing using PowerMock

Deep Shah

[PACKT] PUBLISHING

Instant Mock Testing with PowerMock

Discover unit testing using PowerMock

Deep Shah

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Instant Mock Testing with PowerMock

First published: October 2013

Production Reference: 1241013

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-995-0

www.packtpub.com

Credits

Author

Deep Shah

Reviewer

Quentin Ambard

Acquisition Editor

Rubal Kaur

Commissioning Editor

Govindan K

Technical Editor

Aparna Kumari

Copy Editors

Alisha Aranha

Kirti Pai

Lavina Pereira

Project Coordinator

Sherin Padayatty

Proofreader

Simran Bhogal

Clyde Jenkins

Production Coordinator

Alwin Roy

Cover Work

Alwin Roy

About the Author

Deep Shah has been writing software for over a decade. As a child, he was always fascinated by computers. As time passed, his fascination grew into passion, and he decided to take up Software Development as a career choice. Deep has a degree in Computer Science and is a big fan of open source software. Being a Software Developer at heart, he likes exploring new programming languages, tools, and frameworks.

Deep strongly believes in writing unit tests. He thinks that any code (no matter when it's written) that does not have unit tests is legacy code. Deep has served stints with companies such as Amazon, SpiderLogic, and ThoughtWorks. He speaks a number of languages including Java, C#, JavaScript, Scala, and Haskell. In his free time, Deep likes to see the world, go to cool places, and click lots of pictures.

I would like to thank Jayway and the Java Community for creating a great mocking framework.

I cannot imagine finishing this book without the dedication and support of my loving family, who put up with long nights and working weekends for far longer than I had initially planned.

About the reviewer

Quentin Ambard uses PowerMock daily as a Java mock framework. He is currently the lead developer of the startup MyProcurement.fr, where he works with HBase and MongoDB, along with a bit of Scala on top of it.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

PowerMock is an open source mocking library for the Java world. It extends the existing mocking frameworks, such as EasyMocks (see `http://www.easymock.org/`) and Mockito (see `http://code.google.com/p/mockito/`), to add even more powerful features to them.

PowerMock was founded by Jayway (see `http://www.jayway.com/`) and is hosted on Google Code (see `http://code.google.com/p/powermock/`).

It has a vibrant community with a lot of contributors. Conscious efforts have been made to ensure that PowerMock does not reinvent the wheel. It only extends existing mocking frameworks and adds features that are missing from them. The end result is a mocking library that is powerful and is a pleasure to use.

Sometimes, a good design might have to be tweaked to enable testability. For example, use of final classes or methods should be avoided, private methods might need to open up a bit by making them package-visible or protected, and use of static methods should be avoided at all costs. Sometimes these decisions might be valid, but if they are taken only because of limitations in existing mocking frameworks, they are incorrect. PowerMock tries to solve this problem. It enables us to write unit tests for almost any situation.

What this book covers

Saying Hello World! (Simple) explains a basic mocking example using PowerMock. It will help us get familiarized with basic mocking and verification syntax.

Getting and installing PowerMock (Simple) demonstrates the steps for setting up PowerMock using IntelliJ IDEA and Eclipse. It also briefly describes other ways of setting up the PowerMock environment.

Mocking static methods (Simple) shows how effortlessly we can mock static methods with PowerMock. Most of the mocking frameworks have trouble mocking static methods. But for Power Mock, it's just another day at work.

Verifying method invocation (Simple) explains various ways in which we can verify a certain method invocation. Verification is an indispensable part of unit testing.

Mocking final classes or methods (Simple) covers how easily we can mock final classes or methods. Mocking final classes or methods is something that most mocking frameworks struggle with. Because of this restriction, sometimes a good design is sacrificed.

Mocking constructors (Medium) introduces the art of mocking constructors. Is a class doing too much in its constructor? With PowerMock, we can mock the constructor and peacefully write tests for our own code.

Understanding argument matchers (Medium) demonstrates how to write flexible unit tests using argument matchers. Only verifying that a certain method was invoked is a job half done. Asserting that it was invoked with correct parameters is equally important.

Understanding the Answer interface (Advanced) demonstrates the use of the `Answer` *interface, using which we can create some unusual mocking strategies. Sometimes mocking requirements are extremely complex, which makes it impractical to create mocks in the traditional way. The* `Answer` *interface can be used for such cases.*

Partial mocking with spies (Advanced) explains the steps to mock only a few methods of a given class while invoking the real implementation for all other methods. This is achieved in PowerMock by creating spies.

Mocking private methods (Medium) covers the steps to mock and verify private methods. Private methods are difficult to test with traditional mocking frameworks. But for PowerMock, it's a piece of cake.

Breaking the encapsulation (Advanced) shows how we can test the behavior of a private method and verifies the internal state of a class using the `Whitebox` *class. At times, a private method might be performing an important business operation, and we need to write unit tests for that method. The* `Whitebox` *class can be very handy in such situations.*

Suppressing unwanted behavior (Advanced) explains how we can suppress unwanted behavior, such as static initializers, constructors, methods, and fields.

Understanding Mock Policies (Advanced) demonstrates the use of Mock Policies to manage the repeated code needed to set up mocks for a complex object better.

Listening with listeners (Medium) demonstrates the steps to listen for events from the test framework. We might want to do some processing when the test method is invoked or create a report about how many tests were run, how many passed, how many failed, and so on. Listeners are a good fit for such requirements.

The *Understanding Mock Policies (Advanced)* and *Listening with listeners (Medium)* recipes are available for download from the Packt Publishing website, `http://www.packtpub.com/sites/default/files/downloads/Bonus_Recipes.pdf`

What you need for this book

For setting up PowerMock, we will need JDK 6 or a later version installed on the machine.

The detailed instructions to download and install JDK 6 or a later version can be found at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

In addition, we will need an Integrated Development Environment (IDE) to write unit tests with PowerMock effectively.

This book covers the steps to integrate PowerMock with Eclipse and IntelliJ IDEA. Download and install any one of your favorite IDEs from `http://www.jetbrains.com/idea/` for IntelliJ IDEA, and `http://www.eclipse.org/` for Eclipse.

Who this book is for

Written specifically for new users of PowerMock with little or no experience, this book will also help users of other mocking libraries to get familiar with PowerMock concepts. It also covers advanced PowerMock concepts for people with intermediate knowledge of PowerMock.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: “We can include other contexts through the use of the `include` directive.”

A block of code is set as follows:

```
public class EmployeeController {

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService
        employeeService) {
        this.employeeService = employeeService;
    }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public class EmployeeController {

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService
        employeeService) {
        this.employeeService = employeeService;
    }
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: “clicking on the **Next** button moves you to the next screen”.

Warnings or important notes appear in a box like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant Mock Testing with PowerMock

Welcome to *Instant Mock Testing with PowerMock*. This book will demonstrate the effective use of this versatile open source mocking framework. The recipes described in this book will introduce most of the concepts of PowerMock (see `http://code.google.com/p/powermock/`) that will enable us to use it effectively.

PowerMock enables us to write good unit tests for even the most untestable code. Most of the mocking frameworks in Java cannot mock static methods or final classes. But using PowerMock, we can mock almost any class.

PowerMock does not intend to reinvent the wheel. It extends existing frameworks such as EasyMocks (see `http://www.easymock.org/`) and Mockito (see `http://code.google.com/p/mockito/`) to enable us to do some of the things that these frameworks cannot. Because of this, PowerMock is extremely easy to learn and use.

The first part of the book will help us understand some of the basic mocking and verifying techniques using PowerMock. The later recipes will focus on writing unit tests for more complex scenarios.

PowerMock currently extends the EasyMock and Mockito mocking frameworks. Depending on which extension is preferred, the syntax to write any unit test differs slightly. All the features described in this book are supported using both these extensions, but in the interest of time and space, all the examples in this book will be developed using the PowerMock Mockito API.

Currently, PowerMock integrates with the JUnit (see `http://junit.org/`) and TestNG (see `http://testng.org/doc/index.html`) test frameworks. We will be using JUnit in all our examples.

Saying Hello World! (Simple)

Unit testing enables us to test small bits of code in isolation. This essentially enables us to test the behavior and functionality of our system in a very granular way.

Let's say that we want to write unit tests for a piece of code that converts an employee name to uppercase and writes it to a database. There are two ways in which we can test this:

- Assert that the code converts the employee name to uppercase and also verify that it gets written to the database. In this approach we verify two things:
  - The business logic of converting the employee name to uppercase
  - The fact that it gets written to the database correctly

  To verify the second statement, we need to make sure that the database is available when the test is executing. We are essentially testing the integration of our code with the database system. This approach of testing is different from unit testing and is called integration testing.

- Assert that the code converts the employee name to uppercase and verify that a call was made to the database to write this information. In this approach we verify two things (which are slightly different from what we verify in the first approach):
  - The business logic of converting the employee name to uppercase
  - The fact that we made a call to write this information to the database

  In this approach, we do not actually go to the database and check if the employee name was successfully written. We only verify that a call was made to write this information to the database.

  It's clear by now that we cannot work with the actual database in this approach. This is when mocking comes into the picture. Mocks are nothing but simulated objects that can mimic the behavior of real objects.

This recipe will demonstrate a very simple mocking example using PowerMock. It will show us the basic syntax for creating a mock and verifying a method invocation.

How to do it...

1. Let's say that we have a very simple `EmployeeController` class. As the name suggests, this class performs the **Create, Read, Update, and Delete** (**CRUD**) operations on the `Employee` class.
2. This class delegates the heavy lifting to the `EmployeeService` class to actually perform these operations.

3. Here's the code:

```
/**
 * This is a very simple employee controller
 * which will make use of the EmployeeService to
 * perform Create, Read, Update and Delete (CRUD)
 * of Employee objects.
 * It delegates the heavy lifting to the
 * EmployeeService class
 * @author Deep Shah
 */
public class EmployeeController {

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService
        employeeService) {
        this.employeeService = employeeService;
    }

    /**
     * This method is responsible to return the
     * projected count of employees in the system.
     * Let's say the company is growing by 20% every year,
     * then the project count of employees is 20% more than
     * the actual count of employees in the system.
     * We will also round it off to the ceiling value.
     * @return Total number of projected employees in the
        system.
     */
    public int getProjectedEmployeeCount() {
        final int actualEmployeeCount =
            employeeService.getEmployeeCount();

        return (int) Math.ceil(actualEmployeeCount * 1.2);
    }


/**
 * This class is responsible to handle the CRUD
 * operations on the Employee objects.
 * @author: Deep Shah
 */
public class EmployeeService {
```

```
    /**
     * This method is responsible to return
     * the count of employees in the system.
     * Currently this method does nothing but
     * simply throws an exception.
     * @return Total number of employees in the system.
     */
    public int getEmployeeCount() {
        throw new UnsupportedOperationException();
    }
}
```

4. Let's write the unit test for the `getProjectedEmployeeCount` method of `EmployeeController`.
5. Currently, the `getEmployeeCount` method of `EmployeeService` throws an exception. Hence, we will need to mock the `EmployeeService` instance used by `EmployeeController`.
6. Here is how the test code would look like:

```
/**
 * The class that holds all unit tests for
 * the EmployeeController class.
 * @author: Deep Shah
 */
public class EmployeeControllerTest {

    @Test
    public void shouldGetCountOfEmployees() {
        EmployeeController employeeController =
            new EmployeeController(new EmployeeService());
        Assert.assertEquals(10,
            employeeController.getEmployeeCount());
    }
}
```

7. The preceding test creates an instance of `EmployeeController` by passing the reference of `EmployeeService`. It then asserts that the count of employees is equal to `10`.
8. If we run the preceding test without any modification, the test case should fail with an instance of `UnsupportedOperationException`.
9. To fix the test, we will need to mock `EmployeeService`. We would also need to mock the `getEmployeeCount` method of `EmployeeService` to return the value `8`, which will then be bumped up by 20%, and the value `10` will be returned from the controller. This is what is asserted by our test.

10. Here's how we could do it using PowerMock:

```
@Test
public void
shouldReturnProjectedCountOfEmployeesFromTheService() {
    //Creating a mock using the PowerMockito.mock
    //method for the EmployeeService class.
    EmployeeService mock =
        PowerMockito.mock(EmployeeService.class);

    //Next statement essentially says that when
    //getProjectedEmployeeCount method
    //is called on the mocked EmployeeService instance,
    //return 8.
    PowerMockito.when(mock.getEmployeeCount())
        .thenReturn(8);

    EmployeeController employeeController = new
        EmployeeController(mock);
    Assert.assertEquals(10, employeeController
        .getProjectedEmployeeCount());
}
```

11. If we now run the above test, it would pass.
12. Let's look at one more method in `EmployeeController`:

```
/**
 * This method saves the employee instance.
 * It delegates this task to the employee service.
 * @param employee the instance to save.
 */
public void saveEmployee(Employee employee) {
    employeeService.save(employee);
}
```

13. This new method is called `saveEmployee`; its responsibility is to save the employee instance. It achieves this by delegating this responsibility to `EmployeeService`.
14. This method does not return any value. Hence, the unit test for this controller method has to simply assert that the `saveEmployee` method on the `EmployeeService` instance was called.
15. The test for this method is as follows:

```
@Test
public void
shouldInvokeSaveEmployeeOnTheServiceWhileSavingTheEmployee() {
    EmployeeService mock =
        PowerMockito.mock(EmployeeService.class);
```

```
        EmployeeController employeeController = new
            EmployeeController(mock);

        Employee employee = new Employee();
        employeeController.saveEmployee(employee);

        //Verifying that controller did call the
        //saveEmployee method on the mocked service instance.
        Mockito.verify(mock).saveEmployee(employee);
    }
```

16. This test is very similar to the previous test we saw. It creates a mock of `EmployeeSerivce`, and then constructs the instance of `EmployeeController`. It then invokes the `saveEmployee` method on the controller.
17. What is more interesting is the last statement. In the last statement, we are asserting that the `saveEmployee` method was called on the mocked instance of `EmployeeService`. If we run the test, it will pass.
18. Notice that to verify the `saveEmployee` method was invoked on the mocked instance of `EmployeeService`, we are using `Mockito.verify`. This shows how PowerMock does not reinvent the wheel. It essentially extends the functionality of the existing mocking frameworks.
19. To verify that the test case is actually asserting what it's supposed to assert, comment the call to the `saveEmployee` method of `EmployeeService` and rerun the test.

    `//employeeService.saveEmployee(employee);`

20. Running the test after this change hits us with a failure. The failure message is as follows:

```
Wanted but not invoked:
employeeService.saveEmployee(
    com.gitshah.powermock.Employee@276bab54);
-> at com.gitshah.powermock.EmployeeControllerTest
    .shouldInvokeSaveEmployeeOnTheServiceWhileSaving
    TheEmployee(EmployeeControllerTest.java:39)
Actually, there were zero interactions with this mock.
...
```

21. Even the failure message is extremely easy to read and understand. It tells us that we were expecting the `saveEmployee` method of the `EmployeeService` class will be called. However, there were zero interactions with that mocked instance of `EmployeeService`.

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.

How it works...

To create a mocked instance of any class, we can use `PowerMockito.mock`.

This mocked instance can be programmed to return dummy data on the occurrence of a certain event (such as method invocation). To set up the dummy data, we have to use the `PowerMockito.when` method.

The `Mockito.verify` method is used to verify that a certain method was invoked on the mocked instance.

There's more...

The `PowerMockito.mock` method has a couple of overloads. One of the overloads takes in `MockSettings` as an argument.

MockSettings

`MockSettings` is rarely used. Using `MockSettings` we can do the following:

1. **Naming the mocks**: This can be helpful while debugging.
2. **Creating mocks that implement extra interfaces**: This can be helpful to cover some corner cases; for more information, visit `https://groups.google.com/forum/?fromgroups=#!topic/mockito/YM5EF0x90_4`.
3. **Enabling verbose logging**: This can be helpful while debugging a test. It can be used to find incorrect interactions with the mock.
4. Register a listener for notifying method invocations on the mock.
5. Here's an example of using `MockSettings`:

```
@Test
public void shouldInvokeSaveEmployeeOnTheServiceWhile
    SavingTheEmployeeWithMockSettings() {
    EmployeeService mock =
        PowerMockito.mock(EmployeeService.class, Mockito
        .withSettings()
        .name("EmployeeServiceMock")
        .verboseLogging());
```

```
        EmployeeController employeeController = new
            EmployeeController(mock);

        Employee employee = new Employee();
        employeeController.saveEmployee(employee);

        //Verifying that controller did call the
        //saveEmployee method on the mocked service
        //instance.
        Mockito.verify(mock).saveEmployee(employee);
    }
```

6. The preceding code will set up the name of the mocked object as `EmployeeServiceMock`, and enable verbose logging.

Getting and installing PowerMock (Simple)

This recipe will describe the steps for setting up PowerMock in two major Integrated Development Environments (IDEs): IntelliJ IDEA (see `http://www.jetbrains.com/idea/`) and Eclipse (see `http://www.eclipse.org/`).

Getting ready

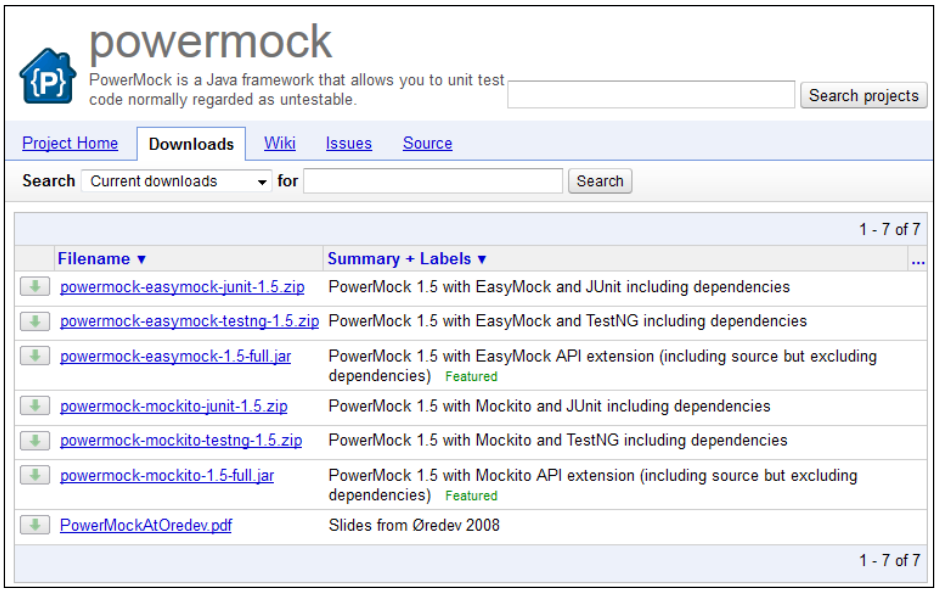
For setting up PowerMock, we will need JDK 6 or the later versions installed on the machine.

The detailed instructions on downloading and installing JDK 6 or a later version can be found at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

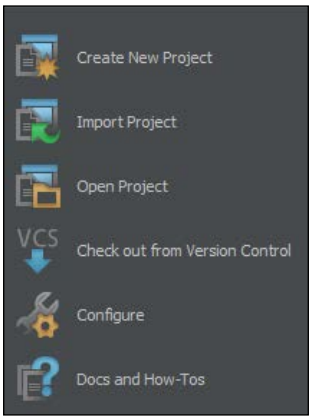
Download and install any one of your favourite IDEs from `http://www.jetbrains.com/idea/` for IntelliJ IDEA and `http://www.eclipse.org/` for Eclipse.

How to do it...

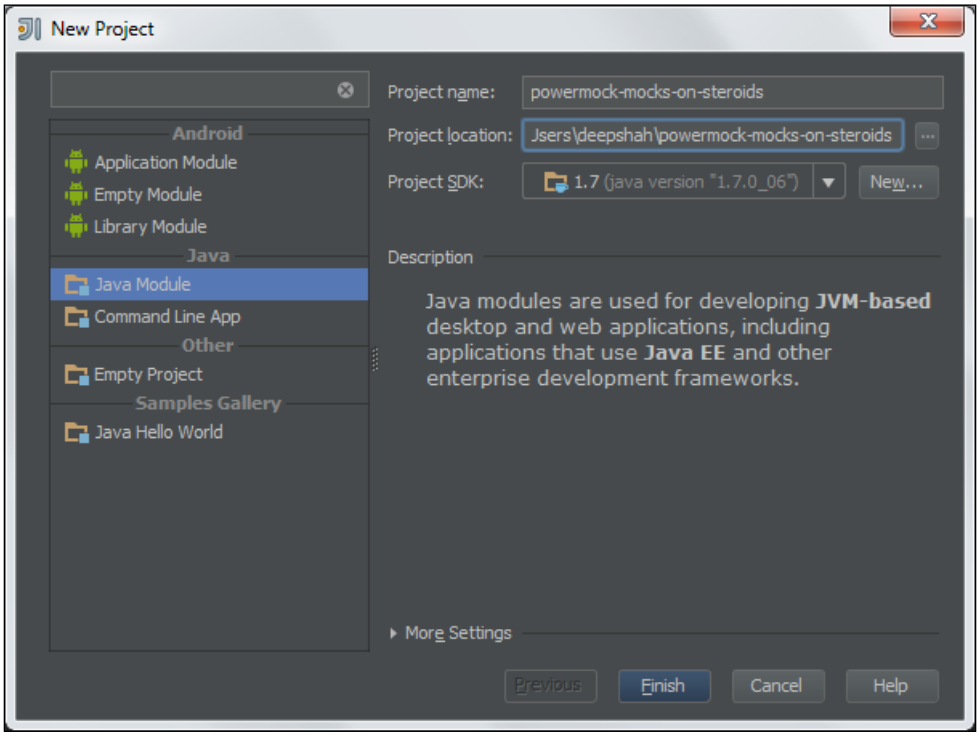
1. PowerMock is an open source mocking framework that is under active development. At the time of writing this book, PowerMock 1.5 was the most recent stable release.
2. Let's start by downloading PowerMock 1.5. Visit PowerMock home at `http://code.google.com/p/powermock/`.
3. PowerMock is hosted on Google Code. Click on the **Downloads** Tab on the page and you should see something as follows:



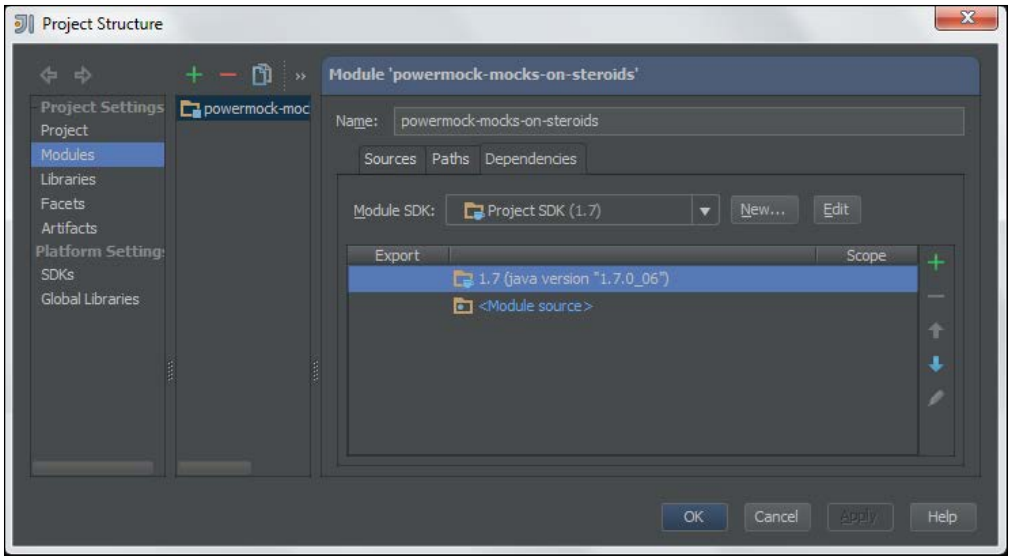
4. Since we are going to develop all our examples using the Mockito extension and JUnit test framework, download the `powermock-mockito-junit-1.5.zip` file from `http://code.google.com/p/powermock/downloads/detail?name=powermock-mockito-junit-1.5.zip&can=2&q=&sort=summary`.
5. Extract the ZIP file in the `$HOME/powermock-mocks-on-steroids/lib` directory. This ZIP file has all the dependent JAR files that we will need for writing unit tests with PowerMock.
6. Now let's set up PowerMock in IntelliJ IDEA.
7. Open IntelliJ IDEA. You should see a screen similar to the following screenshot:



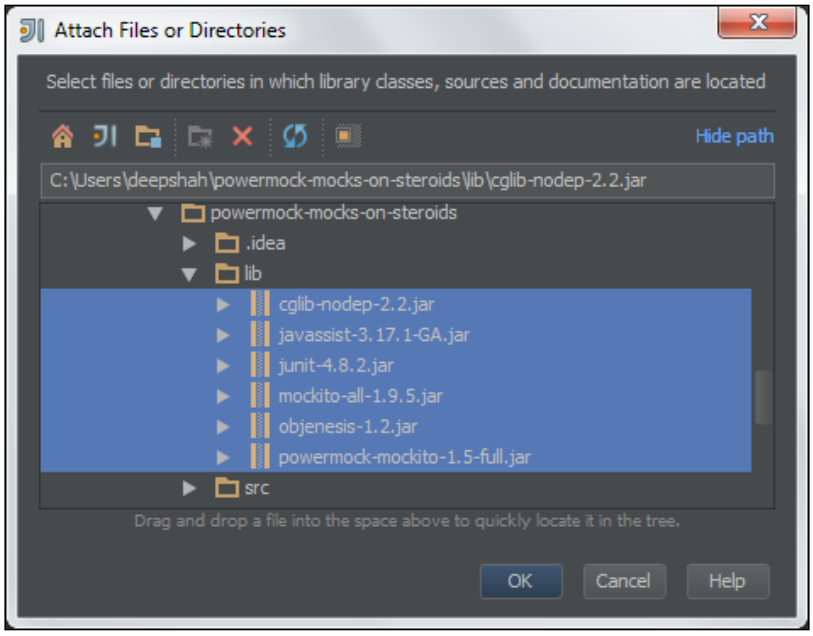
8. Start by clicking on **Create New Project**. It will ask us what type of project we want to create. Select **Java Module** from the left pane. Enter `powermock-mocks-on-steroids` in the **Project name** text field, and set the **Project location** as `$HOME/powermock-mocks-on-steroids`:



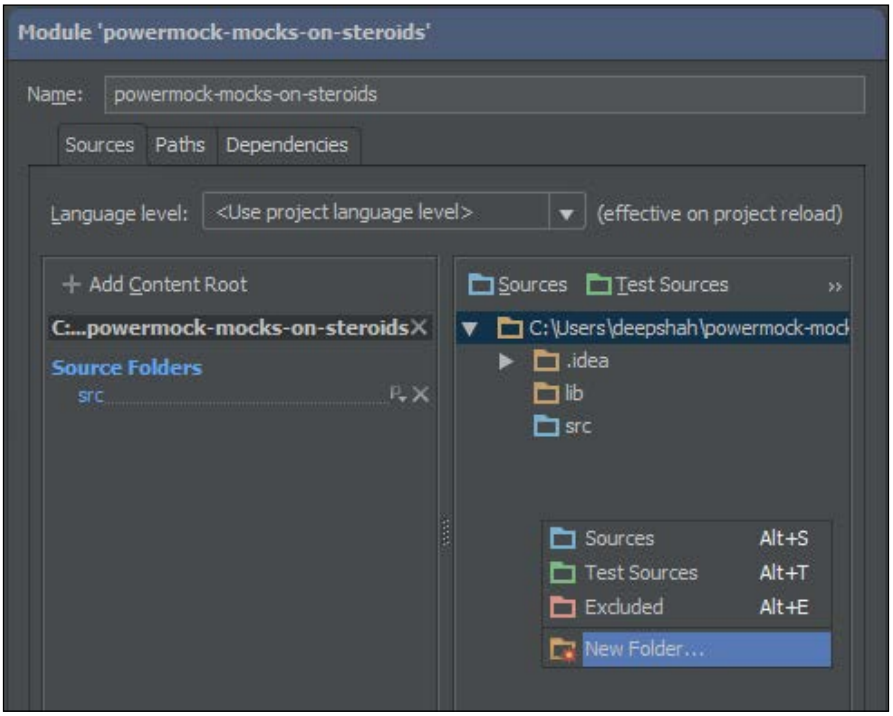
9. Clicking on **Finish** will open up the project with the project structure displayed on the left side. Notice that IntelliJ IDEA should have added the `src` folder automatically.
10. Next, let's add PowerMock and other dependent JARs to the project.
11. Right-click on the project and select **Open Module Settings** from the context menu.
12. This should open up a dialog window. In the dialog window, click on the **Dependencies** tab on the right side. This is how your screen should look like:



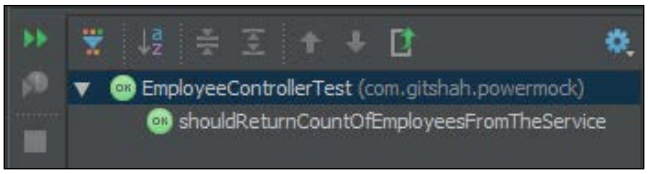
13. Press *ALT* + *Insert*. This should open up a small pop up; on the pop up select **Jars or directories...**.
14. This should open up a dialog that looks similar to the Windows Explorer. Navigate to the path `$HOME/powermock-mocks-on-steroids/lib`, select all the JAR files (which we extracted in Step 5), and click on **OK**. This is how your screen should look:



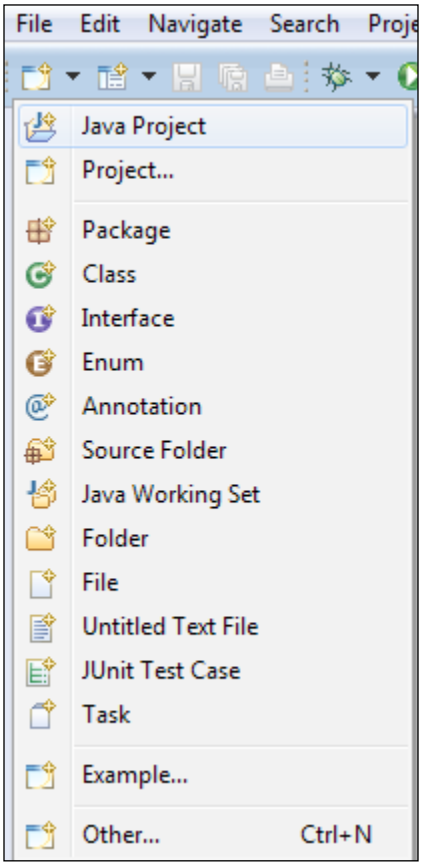
15. Click on the **Sources** tab in the module settings dialog. Right-click on the right-most bottom panel. This should open up a context menu, from this menu click on **New Folder...**. It should look very similar to the following screenshot:



16. This should open up a dialog to enter the folder name; enter `test`.
17. Click on the folder `test` in the module settings dialog, and mark it as **Test Sources**. Close the module settings dialog.
18. Create the source files `EmployeeController.java` and `EmployeeService.java` with only the `getEmployeeCount` method (from the *Saying Hello World! (Simple)* recipe) under the `src` folder in the `com.gitshah.powermock` package.
19. Create the test file `EmployeeControllerTest.java` with the method `shouldReturnCountOfEmployeesFromTheService` (from the *Saying Hello World! (Simple)* recipe.) under the `test` folder in the `com.gitshah.powermock` package.
20. Right-click on the `EmployeeControllerTest.java` file, and click on **Run 'EmployeeControllerTest'**. This should compile the project and run the unit test. If everything is set up correctly, we should see a screen that looks like the following screenshot, which indicates that the test passed.

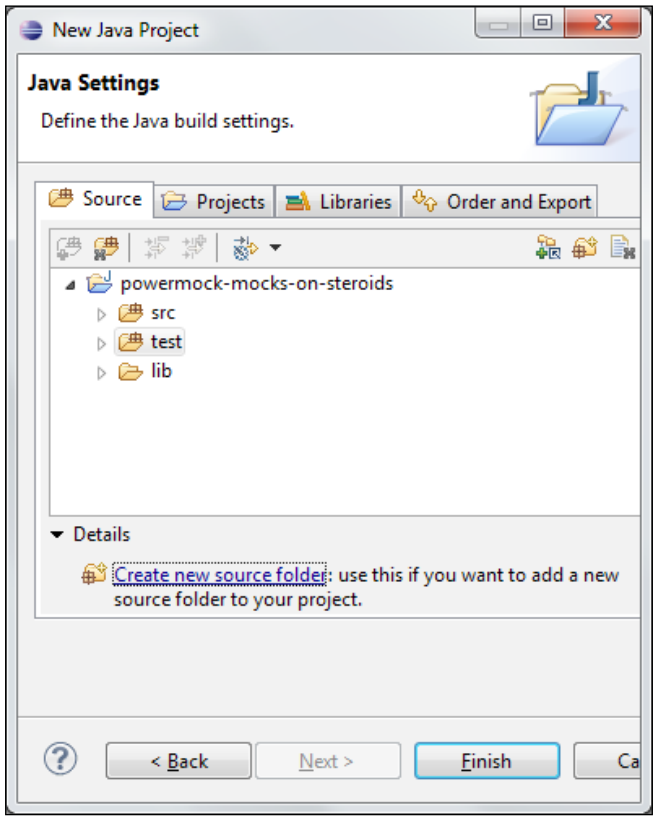


21. Now, let's set up PowerMock in Eclipse.
22. Open Eclipse and click on the small down arrow next to the new icon on the toolbar. This opens up a context menu. In the menu click on **Java Project**. The context menu should look as follows:

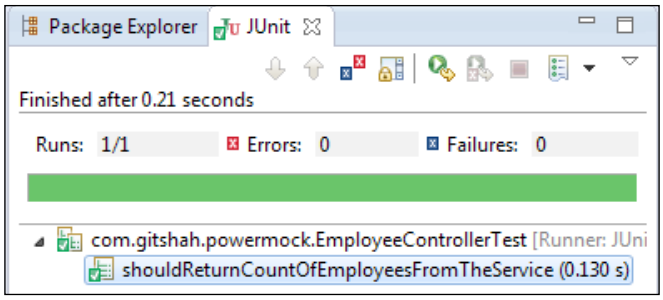


23. This opens up the Create Java Project wizard. In the first step, enter the project name as `powermock-mocks-on-steroids`, and click on **Next**. This should show the **Java Settings** dialog.

24. On the **Source** tab, click on the **Create new source folder** link and create the `src` and `test` folders. The end result should look something like:



25. Click on **Finish**, Eclipse should now open the project and you should see the **Project Explorer** on the left side.
26. Create the source files `EmployeeController.java` and `EmployeeService.java` with only the `getEmployeeCount` method (from the *Saying Hello World! (Simple)* recipe) under the `src` folder in the `com.gitshah.powermock` package.
27. Create the test file `EmployeeControllerTest.java` with the method `shouldReturnCountOfEmployeesFromTheService` (from the *Saying Hello World! (Simple)* recipe) under the `test` folder in the `com.gitshah.powermock` package.
28. Right-click on the `EmployeeControllerTest.java` file, and click on **Run As**. This opens up a smaller submenu, from the submenu select **JUnit Test**. This should compile the project and run the unit test. If everything is set up correctly, we should see that the test passes.



There's more...

PowerMock provides various other ways in which it can be installed.

Maven integration

Maven (see `http://maven.apache.org/`) is a software project management and build tool.

Add the following XML snippet to `pom.xml` for integrating PowerMock Mockito API with JUnit:

```
<properties>
   <powermock.version>1.5.1</powermock.version>
</properties>
<dependencies>
   <dependency>
      <groupId>org.powermock</groupId>
      <artifactId>powermock-module-junit4</artifactId>
      <version>${powermock.version}</version>
      <scope>test</scope>
   </dependency>

   <dependency>
      <groupId>org.powermock</groupId>
      <artifactId>powermock-api-mockito</artifactId>
      <version>${powermock.version}</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Follow the links on the `http://code.google.com/p/powermock/wiki/GettingStarted` page under the **Maven setup** heading to integrate PowerMock with Maven.

Other integration options

- PowerMock provides prebuild ZIP files for integrating the TestNG test framework and EasyMock extension API.
- These ZIP files can also be found at `http://code.google.com/p/powermock/downloads/list`.
- The installation process is very similar to the one described here.

Mocking static methods (Simple)

The real power of PowerMock is the ability to mock things that other frameworks can't. One such thing is mocking static methods.

In this recipe we will see how easily we can mock static methods.

Getting ready

The use of static methods is usually considered a bad Object Oriented Programming practice, but if we end up in a project that uses a pattern such as active record (see `http://en.wikipedia.org/wiki/Active_record_pattern`), we will end up having a lot of static methods.

In such situations, we will need to write some unit tests and PowerMock could be quite handy.

Start your favorite IDE (which we set up in the *Getting and installing PowerMock (Simple)* recipe), and let's fire away.

How to do it...

1. We will start where we left off. In the `EmployeeService.java` file, we need to implement the `getEmployeeCount` method; currently it throws an instance of `UnsupportedOperationException`.
2. Let's implement the method in the `EmployeeService` class; the updated classes are as follows:

```
/**
 * This class is responsible to handle the CRUD
 * operations on the Employee objects.
 * @author Deep Shah
 */
public class EmployeeService {
   /**
```

```
     * This method is responsible to return
     * the count of employees in the system.
     * It does it by calling the
     * static count method on the Employee class.
     * @return Total number of employees in the system.
     */
    public int getEmployeeCount() {
        return Employee.count();
    }
}
/**
 * This is a model class that will hold
 * properties specific to an employee in the system.
 * @author Deep Shah
 */
public class Employee {
    /**
     * The method that is responsible to return the
     * count of employees in the system.
     * @return The total number of employees in the system.
     * Currently this
     * method throws UnsupportedOperationException.
     */
    public static int count() {
        throw new UnsupportedOperationException();
    }
}
```

3. The `getEmployeeCount` method of `EmployeeService` calls the static method `count` of the `Employee` class. This method in turn throws an instance of `UnsupportedOperationException`.
4. To write a unit test of the `getEmployeeCount` method of `EmployeeService`, we will need to mock the static method `count` of the `Employee` class.
5. Let's create a file called `EmployeeServiceTest.java` in the `test` directory. This class is as follows:

```
/**
 * The class that holds all unit tests for
 * the EmployeeService class.
 * @author Deep Shah
```

```
 */
@RunWith(PowerMockRunner.class)
@PrepareForTest(Employee.class)
public class EmployeeServiceTest {

   @Test
   public void shouldReturnTheCountOfEmployees
      UsingTheDomainClass() {
      PowerMockito.mockStatic(Employee.class);
      PowerMockito.when(Employee.count()).thenReturn(900);

      EmployeeService employeeService = new
         EmployeeService();
      Assert.assertEquals(900,
         employeeService.getEmployeeCount());
   }
}
```

6. If we run the preceding test, it passes. The important things to notice are the two annotations (`@RunWith` and `@PrepareForTest`) at the top of the class, and the call to the `PowerMockito.mockStatic` method.
   - The `@RunWith(PowerMockRunner.class)` statement tells JUnit to execute the test using `PowerMockRunner`.
   - The `@PrepareForTest(Employee.class)` statement tells PowerMock to prepare the `Employee` class for tests. This annotation is required when we want to mock final classes or classes with final, private, static, or native methods.
   - The `PowerMockito.mockStatic(Employee.class)` statement tells PowerMock that we want to mock all the static methods of the `Employee` class.
   - The next statements in the code are pretty standard, and we have looked at them earlier in the *Saying Hello World! (Simple)* recipe. We are basically setting up the static `count` method of the `Employee` class to return `900`. Finally, we are asserting that when the `getEmployeeCount` method on the instance of `EmployeeService` is invoked, we do get `900` back.
7. Let's look at one more example of mocking a static method; but this time, let's mock a static method that returns `void`.
8. We want to add another method to the `EmployeeService` class that will increment the salary of all employees (wouldn't we love to have such a method in reality?).
9. Updated code is as follows:

```
/**
```

```
* This method is responsible to increment the salary
* of all employees in the system by the given percentage.
* It does this by calling the static giveIncrementOf method
* on the Employee class.
* @param percentage the percentage value by which
*   salaries would be increased
* @return true if the increment was successful.
* False if increment failed because of some exception
*   otherwise.
*/
public boolean giveIncrementToAllEmployeesOf(int
   percentage) {
   try{
      Employee.giveIncrementOf(percentage);
      return true;
   } catch(Exception e) {
      return false;
   }
}
```

10. The static method `Employee.giveIncrementOf` is as follows:

```
/**
* The method that is responsible to increment
* salaries of all employees by the given percentage.
* @param percentage the percentage value by which
*   salaries would be increased
* Currently this method throws
*   UnsupportedOperationException.
*/
public static void giveIncrementOf(int percentage) {
    throw new UnsupportedOperationException();
}
```

11. The earlier syntax would not work for mocking a **void static method**. The test case that mocks this method would look like the following:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(Employee.class)
public class EmployeeServiceTest {

   @Test
   public void shouldReturnTrueWhenIncrementOf10
      PercentageIsGivenSuccessfully() {
      PowerMockito.mockStatic(Employee.class);
      PowerMockito.doNothing().when(Employee.class);
      Employee.giveIncrementOf(10);
```

```
        EmployeeService employeeService = new
            EmployeeService();
        Assert.assertTrue(employeeService.giveIncrementTo
            AllEmployeesOf(10));
    }

    @Test
    public void shouldReturnFalseWhenIncrementOf10
        PercentageIsNotGivenSuccessfully() {
        PowerMockito.mockStatic(Employee.class);
        PowerMockito.doThrow(new
            IllegalStateException()).when(Employee.class);
        Employee.giveIncrementOf(10);

        EmployeeService employeeService = new
            EmployeeService();
        Assert.assertFalse(employeeService.giveIncrementTo
            AllEmployeesOf(10));
    }
}
```

12. Notice that we still need the two annotations `@RunWith` and `@PrepareForTest`, and we still need to inform PowerMock that we want to mock the static methods of the `Employee` class.
13. Notice the syntax for `PowerMockito.doNothing` and `PowerMockito.doThrow`:
    - The `PowerMockito.doNothing` method tells PowerMock to literally *do nothing* when a certain method is called. The next statement of the `doNothing` call sets up the mock method. In this case it's the `Employee.giveIncrementOf` method. This essentially means that PowerMock will do nothing when the `Employee.giveIncrementOf` method is called.
    - The `PowerMockito.doThrow` method tells PowerMock to throw an exception when a certain method is called. The next statement of the `doThrow` call tells PowerMock about the method that should throw an exception; in this case, it would again be `Employee.giveIncrementOf`. Hence, when the `Employee.giveIncrementOf` method is called, PowerMock will throw an instance of `IllegalStateException`.

How it works...

PowerMock uses **custom class loader** and **bytecode manipulation** to enable mocking of static methods. It does this by using the `@RunWith` and `@PrepareForTest` annotations.

The rule of thumb is whenever we want to mock any method that returns a **non-void value**, we should be using the `PowerMockito.when().thenReturn()` syntax. It's the same syntax for instance methods as well as static methods.

But for methods that return `void`, the preceding syntax cannot work. Hence, we have to use `PowerMockito.doNothing` and `PowerMockito.doThrow`. This syntax for static methods looks a bit like the record-playback style.

On a mocked instance created using PowerMock, we can choose to return canned values only for a few methods; however, PowerMock will provide defaults values for all the other methods. This means that if we did not provide any canned value for a method that returns an `int` value, PowerMock will mock such a method and return `0` (since `0` is the default value for the `int` datatype) when invoked.

There's more...

The syntax of `PowerMockito.doNothing` and `PowerMockito.doThrow` can be used on instance methods as well.

.doNothing and .doThrow on instance methods

The syntax on instance methods is simpler compared to the one used for static methods.

1. Let's say we want to mock the instance method `save` on the `Employee` class. The `save` method returns `void`, hence we have to use the `doNothing` and `doThrow` syntax. The test code to achieve is as follows:

```
/**
 * The class that holds all unit tests for
 * the Employee class.
 * @author Deep Shah
 */
public class EmployeeTest {

   @Test()
   public void shouldNotDoAnythingIfEmployeeWasSaved() {
      Employee employee =
         PowerMockito.mock(Employee.class);
   PowerMockito.doNothing().when(employee).save();

      try {
         employee.save();
      } catch(Exception e) {
         Assert.fail("Should not have thrown an
            exception");
      }
```

```
    }

    @Test(expected = IllegalStateException.class)
    public void shouldThrowAnExceptionIf
       EmployeeWasNotSaved() {
       Employee employee =
          PowerMockito.mock(Employee.class);
       PowerMockito.doThrow(new
          IllegalStateException()).when(employee).save();

       employee.save();
    }
}
```

2. To inform PowerMock about the method to mock, we just have to invoke it on the return value of the `when` method. The line `PowerMockito.doNothing().when(employee).save()` essentially means **do nothing** when the `save` method is invoked on the mocked `Employee` instance.
3. Similarly, `PowerMockito.doThrow(new IllegalStateException()).when(employee).save()` means throw `IllegalStateException` when the `save` method is invoked on the mocked `Employee` instance.
4. Notice that the syntax is more fluent when we want to mock `void` instance methods.

Verifying method invocation (Simple)

Verification is a process where we assert that a certain method was invoked by the code under test. PowerMock provides various ways in which we can perform the verification of a method call. This recipe will cover the steps required to verify a method invocation in various scenarios.

Getting ready

Let's start by verifying an instance method. The `EmployeeService` class had a method called `saveEmployee`; we are going to implement this method and write tests for it. Fire off your favorite IDE and let's begin.

How to do it...

1. The responsibility of the `saveEmployee` method is to save the employee's information to the database (DB). Here is how the code looks for this method:

```
/**
* The method that will save
* the employee instance to the DB.
```

```
* @param employee instance to save.
*/
public void saveEmployee(Employee employee) {
   if(employee.isNew()) {
      employee.create();
      return;
   }
   employee.update();
}
```

2. The corresponding code for the `Employee` class is as follows:

```
/**
 * The method that identifies if the employee
 * is not yet persisted in the DB.
 * @return true if employee is not yet
 * persisted in the DB, false otherwise.
 * Currently this method throws
 * UnsupportedOperationException
 */
public boolean isNew() {
   throw new UnsupportedOperationException();
}
/**
 * This method is responsible to update
 * an existing employee's information into the DB.
 * Currently this method throws
 * UnsupportedOperationException
 */
public void update() {
   throw new UnsupportedOperationException();
}
/**
 * This method is responsible to create
 * a new employee into the DB.
 * Currently this method throws
 * UnsupportedOperationException
 */
public void create() {
   throw new UnsupportedOperationException();
}
```

3. The `saveEmployee` method of `EmployeeService` checks whether the employee exists in the DB. Based on that, it does the following:
   - If it exists, `saveEmployee` updates the employee's information by invoking `employee.update()`
   - Else, it creates the employee's information by invoking `employee.create()`
4. The `Employee.java` methods currently throw exceptions.
5. Let's write the test case for the `EmployeeService.saveEmployee` method:

```
@Test
public void shouldCreateNewEmployeeIfEmployeeIsNew() {
   Employee mock = PowerMockito.mock(Employee.class);
   PowerMockito.when(mock.isNew()).thenReturn(true);

   EmployeeService employeeService = new EmployeeService();
   employeeService.saveEmployee(mock);

//Verifying that the create method was indeed invoked
//on the employee instance.
   Mockito.verify(mock).create();

//Verifying that while creating a new employee
//update was never invoked.
   Mockito.verify(mock, Mockito.never()).update();
}
```

6. The test method first creates a mock instance of the `Employee` class. It then mocks out the `isNew` method on the mocked instance and returns `true`.
7. After invoking the `saveEmployee` method on the instance of `EmployeeService`, we perform our verifications.
8. In this example, we are verifying that the `create` method was invoked on the `Employee` instance since the employee is a new employee. We are also verifying that we do not call the `update` method, since we are creating a new employee.
9. Let's see one more example of verifying a static method. In the *Mocking static methods (Simple)* recipe, we saw the `EmployeeService.giveIncrementToAllEmployeesOf` method. The tests of this method didn't verify that the `Employee.giveIncrementOf` static method was invoked.
10. To refresh our memory, here's the code for the `EmployeeService.giveIncrementToAllEmployeesOf` method:

```
/**
 * This method is responsible to increment the salary
 * of all employees in the system by the given percentage.
```

```
 * It does this by calling the static giveIncrementOf
 * method
 * on the Employee class.
 * @param percentage the percentage value by which
 * salaries would be increased
 * @return true if the increment was successful.
 * False if increment failed because of some exception
 * otherwise.
 */
public boolean giveIncrementToAllEmployeesOf(int
   percentage) {
   try{
      Employee.giveIncrementOf(percentage);
      return true;
   } catch(Exception e) {
   return false;
   }
}
```

11. The test case to verify that the `Employee.giveIncrementOf` method was actually invoked is as follows:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(Employee.class)
public class EmployeeServiceTest {

   @Test
   public void shouldInvoke_giveIncrementOfMethod
      OnEmployeeWhileGivingIncrement() {
      PowerMockito.mockStatic(Employee.class);
      PowerMockito.doNothing().when(Employee.class);
      Employee.giveIncrementOf(9);

      EmployeeService employeeService = new
         EmployeeService();
      employeeService.giveIncrementToAllEmployeesOf(9);

      //We first have to inform PowerMock that we will now
      //verify
      //the invocation of a static method by calling
      //verifyStatic.
      PowerMockito.verifyStatic();
      //Then we need to inform PowerMock
      //about the method we want to verify.
      //This is done by actually invoking the static
```

```
        //method.
        Employee.giveIncrementOf(9);
    }
}
```

12. Reiterating, to mock or verify a static method, we need the two annotations, `@RunWith` and `@PrepareForTest`, at the test class level.
13. The syntax to verify a static method is slightly different. We first need to inform PowerMock that we are now going to verify a static method by calling `PowerMockito.verifyStatic()`.
14. Next we need to perform the verification. This is done by actually invoking the desired static method; in this case, `Employee.giveIncrementOf`, and passing `9` as the argument.

How it works...

Mocks created using PowerMock will remember all method invocations made on them. To verify that a certain operation was invoked or never invoked, we have to use the `Mockito.verify` method. This method is to be used when we want to verify method invocation of instance methods.

This is an example of how PowerMockito does not try and reinvent the wheel. Instead, it reuses the functionality developed by the underlining mocking framework (in this case, Mockito).

To verify the invocation of static methods, we first need to inform PowerMock that we are going to verify the invocation of static methods by calling `PowerMockito.verifyStatic()`. Then we actually have to invoke the static method. This is not considered as an actual method invocation but as a static method verification.

There's more...

While writing tests for a decently complex method, we will end up with more than one test for testing the different aspects of the method. In such cases, we might have to write duplicate setup code in every test.

To get around this problem, we could create a method. Let's call it `initialize` and annotate it with the `@org.junit.Before` annotation. This method will automatically be invoked by JUnit before every test run. This is a common feature provided by unit testing frameworks to get rid of the repeated setup code across tests. An example of using the `@Before` annotation would be as follows:

```
@Before
public void initialize() {
    PowerMockito.mockStatic(Employee.class);
```

```
    PowerMockito.doNothing().when(Employee.class);
    Employee.giveIncrementOf(9);
}

@Test
public void shouldInvoke_giveIncrementOfMethodOn
    EmployeeWhileGivingIncrement() {
    EmployeeService employeeService = new EmployeeService();
    employeeService.giveIncrementToAllEmployeesOf(9);

    //We first have to inform PowerMock that we will now verify
    //the invocation of a static method by calling verifyStatic.
    PowerMockito.verifyStatic();
    //Then we need to inform PowerMock
    //about the method we want to verify.
    //This is done by actually invoking the static method.
    Employee.giveIncrementOf(9);
}
```

In the above example, notice how we have moved the mock setup code in the `initialize` method. This method is annotated with the `@Before` annotation. This method will be automatically invoked by JUnit before every test method is executed.

Other verification modes

We saw one overload of the `verify` method, the one that takes in `Mockito.never()`. The second argument to the `verify` method is of type `VerificationMode`. There are various other useful verification modes as well. The following verification modes are also valid for the `PowerMockito.verifyStatic()` method:

- `Mockito.times(int n)`: This verification mode asserts that the mocked method was invoked *exactly 'n' times*
- `Mockito.atLeastOnce()`: This verification mode asserts that the mocked method was invoked *at least once*
- `Mockito.atLeast(int n)`: This verification mode asserts that the mocked method was invoked *at least 'n' times*
- `Mockito.atMost(int n)`: This verification mode asserts that the mocked method was invoked *at most 'n' times*

InOrder verification

Sometimes, we want to make sure that methods are invoked in a certain sequence only. For example, while testing the `EmployeeService.saveEmployee` method, we want to make sure that `employee.isNew()` is called before calling either `employee.create()` or `employee.update()`. Let's look at an example of doing this:

```
@Test
public void shouldInvokeIsNewBeforeInvokingCreate() {
   Employee mock = PowerMockito.mock(Employee.class);

   EmployeeService employeeService = new EmployeeService();
   employeeService.saveEmployee(mock);

   //First we have to let PowerMock know that
   //the verification order is going to be important
   //This is done by calling Mockito.inOrder and passing
   //it the mocked object.
   InOrder inOrder = Mockito.inOrder(mock);

   //Next, we can continue our verification using
   //the inOrder instance using the same technique
   //as seen earlier.
   inOrder.verify(mock).isNew();
   inOrder.verify(mock).update();
   inOrder.verify(mock, Mockito.never()).create();
}
```

To verify that method invocations occur in a certain order, we have to do the following:

1. Let PowerMock know that the verification order is important. This is done by calling `Mockito.inOrder(mock)` and passing to it the mocked object.
2. Next, we can verify the method invocations in a given order using the instance of `InOrder`. The syntax to do the verification using the instance of `InOrder` is exactly the same as seen earlier.

Mocking final classes or methods (Simple)

One of the features that separates PowerMock from other mocking frameworks is its ability to mock final classes or methods with ease.

In this recipe, we will see how effortless it is to mock final methods and classes using PowerMock.

Getting ready

When we create new employees in the DB, we should have some unique identifier (such as `EmployeeId`) associated with them. This identifier will help us look up the employee records easily. Let's start by assigning a unique `EmployeeId` to all new employees.

How to do it...

1. The requirement is to assign a unique `EmployeeId` to all new employees.
2. Let's say we have a class called `EmployeeIdGenerator`.
   - This class is responsible for generating a unique `EmployeeId`
   - We want to implement very complex `EmployeeId` generation logic in this class
   - Because of this, we do not want anyone to make it a subclass
   - We will make this class a `final` class with all its methods as static final
3. The updated `saveEmployee` method is as follows:

```
/**
 * The method that will save
 * the employee instance to the DB.
 * @param employee instance to save.
 */
public void saveEmployee(Employee employee) {
   if(employee.isNew()) {
      employee.setEmployeeId
         (EmployeeIdGenerator.getNextId());
      employee.create();
      return;
   }
   employee.update();
}
```

4. Notice that we are using the static method `getNextId` of `EmployeeIdGenerator` to generate `EmployeeId`. The result of this method call is passed to the `setEmployeeId` method of the instance `employee`.
5. We generate `EmployeeId` if and only if the employee is new.
6. The `EmployeeIdGenerator` method is as follows:

```
/**
 * The class that is responsible
 * to generate employee Ids new employees.
 * @author Deep Shah
```

```
 */
public final class EmployeeIdGenerator {

   /**
    * Static method that is responsible to generate
    * the next employee id.
    * @return The next employee id.
    * Currently this method throws
    * UnsupportedOperationException.
    */
   public final static int getNextId() {
      throw new UnsupportedOperationException();
   }
}
```

7. This is the class that is going to hold the complex logic to generate `EmployeeId`, hence we have marked the class as `final` and the method `getNextId` is `final static`.

8. The test case to test this change to the `saveEmployee` method would be as follows:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(EmployeeIdGenerator.class)
public class EmployeeServiceTest {

   @Test
   public void shouldGenerateEmployeeIdIfEmployeeIsNew() {
      Employee mock = PowerMockito.mock(Employee.class);
      PowerMockito.when(mock.isNew()).thenReturn(true);

      PowerMockito.mockStatic(EmployeeIdGenerator.class);
      PowerMockito.when(EmployeeIdGenerator.getNextId())
         .thenReturn(90);

      EmployeeService employeeService = new
         EmployeeService();
      employeeService.saveEmployee(mock);

      PowerMockito.verifyStatic();
      EmployeeIdGenerator.getNextId();
      Mockito.verify(mock).setEmployeeId(90);
      Mockito.verify(mock).create();
   }
}
```

9. The test looks very similar, right? To mock a final class having final static methods, we have to do the following:
    - Annotate the test class with two annotations; that is, `@RunWith` and `@PrepareForTest`. Since the `EmployeeIdGenerator` class is final, and its methods are final static, we have to prepare this class for the test.
    - Then we simply mock the final static method `EmployeeIdGenerator.getNextId()` like it was a normal static method using the `PowerMockito.when().thenReturn()` syntax.
    - Again, to verify the invocation of the final static methods, we have to use the same syntax we used for normal static methods.

How it works...

Mocking of final classes or methods is no different from mocking any normal class or method. This is the beauty of using PowerMock; we do not have to do anything special to mock final classes or methods.

Most of the mocking frameworks are based on the use of the Proxy pattern (see `http://en.wikipedia.org/wiki/Proxy_pattern#Example`). The Proxy pattern is heavily dependent on the fact that a class can be subclassed and a method can be overridden. Because of this reason, most of the mocking frameworks cannot mock final methods or classes.

Since PowerMock uses a custom class loader and bytecode manipulation, it is able to achieve what other mocking frameworks fail to do.

Mocking constructors (Medium)

At times, we come across a class that does a lot of work in its constructor itself. This can cause the constructor to become overly complicated and a road block for testing other classes.

In this recipe, we will learn how to mock and verify the invocation of a constructor.

Getting ready

A new requirement has come up while creating new employees. We want to send an e-mail message to welcome the new employees.

We will encapsulate the functionality of sending an e-mail in a class called `WelcomeEmail`. With this requirement in mind, let's look at how we can mock and verify constructors.

How to do it...

1. Let's start off by creating a class called `WelcomeEmail`:

```
/**
 * The class that is responsible to send the Welcome Email
 * to new employees.
 * @author Deep Shah
 */
public class WelcomeEmail {

   /**
    * The constructor for the WelcomeEmail
    * is going to connect to the SMTP server
    * and keep the message ready to be relayed.
    * Currently this constructor throws
    * UnsupportedOperationException.
    */
   public WelcomeEmail(final Employee employee, final
      String message) {
      //Initialize the connection to SMTP server
      //Compose the message body.
      throw new UnsupportedOperationException();
   }

   /**
    * This method is responsible for actually sending the
    * email.
    * Currently this method throws
    * UnsupportedOperationException.
    */
   public void send() {
      throw new UnsupportedOperationException();
   }
}
```

2. It's the responsibility of this class to send a welcome e-mail to the employees. To achieve this, it does the following:
   - It talks to the SMTP server
   - It initializes a connection to the SMTP server
   - It composes the message in the constructor itself

3. In short, the construction of this class is going to be a complex operation. Currently, we are just throwing an instance of `UnsupportedOperationException` from the constructor.
4. This means that whoever constructs an instance of this class is going to get an exception. Even the `send` method throws an instance of `UnsupportedOperationException`.
5. The `EmployeeService` class with the modified `saveEmployee` method code is as follows:

```
/**
 * The method that will save
 * the employee instance to the DB.
 * @param employee instance to save.
 */
public void saveEmployee(Employee employee) {
   if(employee.isNew()) {
      employee.setEmployeeId(EmployeeIdGenerator
         .getNextId());
      employee.create();
      WelcomeEmail emailSender = new
         WelcomeEmail(employee,
         "Welcome to Mocking with PowerMock How-to!");
      emailSender.send();
      return;
   }
   employee.update();
}
```

6. Since we want to send a welcome e-mail to the new employees, we are creating an instance of `WelcomeEmail` by passing to it the `employee` object and the welcome message. Then we are invoking the `send` method on `WelcomeEmail` itself.
7. Usually, methods like this would be considered difficult to test, since we are constructing an instance of `WelcomeEmail`, which throws an exception from the constructor itself. But nothing is difficult for PowerMock, right?
8. Let's look at the test to test this new behavior:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({EmployeeIdGenerator.class,
   EmployeeService.class})
public class EmployeeServiceTest {

   @Test
   public void shouldSendWelcomeEmailToNewEmployees()
      throws Exception {
      Employee employeeMock =
```

```
            PowerMockito.mock(Employee.class);
        PowerMockito.when(employeeMock.isNew())
            .thenReturn(true);

        PowerMockito.mockStatic(EmployeeIdGenerator.class);

        //Creating the mock for WelcomeEmail.
        WelcomeEmail welcomeEmailMock =
            PowerMockito.mock(WelcomeEmail.class);

        /**
         *Notice the whenNew syntax.
         *PowerMockito.whenNew().withArguments().thenReturn()
         *informs PowerMock that,
         *1. When New instance of WelcomeEmail is created,
         *2. With employee instance and "Welcome to Mocking
         *with PowerMock How-to!" text,
         *3. Then return a mock of WelcomeEmail class.
         */
        PowerMockito.whenNew(WelcomeEmail.class)
            .withArguments(employeeMock, "Welcome to Mocking
             with PowerMock How-to!")
            .thenReturn(welcomeEmailMock);

        EmployeeService employeeService = new
            EmployeeService();
        employeeService.saveEmployee(employeeMock);

        /**
         * Verifying that the constructor for the
         * WelcomeEmail class is invoked
         * with arguments as the mocked employee instance and
         * text "Welcome to Mocking with PowerMock How-to!".
         */
        PowerMockito
            .verifyNew(WelcomeEmail.class)
            .withArguments(employeeMock, "Welcome to Mocking
             with PowerMock How-to!");

        //Verifying that the send method was called on the
        //mocked instance.
        Mockito.verify(welcomeEmailMock).send();
    }
}
```

9. The following are a couple of points to notice in the test:
    - Notice the annotation `@PrepareForTest`, we have to pass the `EmployeeService.class` file as an argument to it
    - `PowerMockito.whenNew().withArguments(...).thenReturn()` is the syntax used to mock constructors
    - To verify that the constructor was actually invoked, we have to use the `PowerMockito.verifyNew().withArguments()` syntax

How it works...

The `@PrepareForTest` annotation is used to inform PowerMock about the classes to be prepared for the test. When we want to mock the constructor call of a class (in our case, `WelcomeEmail`), the class that makes the actual call (in our case, `EmployeeService`) needs to be prepared for the test; hence, we need to pass `EmployeeService.class` to the annotation.

The `PowerMockito.whenNew().withArguments(employeeMock, "Welcome to Mocking with PowerMock How-to!").thenReturn(welcomeEmailMock)` syntax is read as follows:

1. When a new instance of the `WelcomeEmail` class is created.
2. With arguments as the instance of `Employee` and welcome message.
3. Return a mocked instance of the `WelcomeEmail` class.

And the `PowerMockito.verifyNew().withArguments()` syntax is read as follows:

1. Verify that a new instance of the `WelcomeEmail` class was created.
2. And the instance of `Employee` and welcome message were passed to the constructor as arguments.

There's more...

The `PowerMockito.verifyNew` method has an overloaded method that takes in an instance of the `VerificationMode` class as the second argument. All verification mode options described in the *Verifying method invocation (Simple)* recipe are valid for this method as well.

Understanding argument matchers (Medium)

Asserting that correct arguments are being passed to the method calls is as important as asserting that the correct method was invoked. PowerMock verifies the argument values using the `equals` method. But sometimes, we need some extra flexibility to assert that correct arguments are being passed. We can achieve this by using argument matchers. In this recipe, we will look at the effective use of argument matchers to assert that methods are invoked with correct arguments.

This is another example of how PowerMock does not reinvent the wheel. It simply uses this functionality from the underlining mocking frameworks (EasyMock and Mockito).

Getting ready

We want to add two new methods in `EmployeeController`:

- `findEmployeeByEmail`: To find the employee via an e-mail
- `isEmployeeEmailAlreadyTaken`: To check if the e-mail address is already taken

How to do it...

1. The code for the two new methods of `EmployeeController` is as follows:

```
/**
 * The method that will
 * find an employee by their email.
 * It delegates this task to the employee service.
 *
 * @param email the employee email to search.
 * @return Employee matching the email.
 */
public Employee findEmployeeByEmail(String email) {
   return employeeService.findEmployeeByEmail(email);
}

/**
 * This method is responsible to check if
 * email is already taken or not.
 * It delegates this task to the employee service.
 *
 * @param email The employee email to validate.
 * @return true if the employee email is taken,
 * false otherwise.
 */
```

```
public boolean isEmployeeEmailAlreadyTaken(String email) {
   return employeeService.employeeExists(new
      Employee(email));
}
```

2. The code is very straightforward. The `findEmployeeByEmail` method calls the `employeeService.findEmployeeByEmail` method to find the employee via an e-mail.
3. The `isEmployeeEmailAlreadyTaken` method calls the `employeeService.employeeExists` method to check if the e-mail is already taken. It creates a new instance of the `Employee` class by passing in the e-mail address.
4. The corresponding methods in the `EmployeeService` class do nothing; currently, they just throw `UnsupportedOperationException`:

```
/**
 * Finds the employee by email.
 * Currently this method throws
 * UnsupportedOperationException.
 * @param email the employee email to search.
 * @return Employee matching the email.
 */
public Employee findEmployeeByEmail(String email) {
   throw new UnsupportedOperationException();
}

/**
 * The method that will check whether
 * the employee exists based on various criterion's.
 * Currently this method throws
 * UnsupportedOperationException.
 * @param employee the employee instance to match.
 * @return true if th employee exists, false otherwise.
 */
public boolean employeeExists(Employee employee) {
   throw new UnsupportedOperationException();
}
```

5. So far, we have always written tests that match exact arguments. Let's look at a test that is a little more flexible to match arguments:

```
@Test
public void shouldFindEmployeeByEmail() {
   final EmployeeService mock =
      PowerMockito.mock(EmployeeService.class);

   final Employee employee = new Employee();
```

```
    //Notice that we are just check if the email address
    //starts with "deep" then we have found the matching
    //employee.
    PowerMockito.when(mock.findEmployeeByEmail(Mockito
       .startsWith("deep"))).thenReturn(employee);

    final EmployeeController employeeController = new
       EmployeeController(mock);

    //Following 2 invocations will match return valid
    //employee,
    //since the email address passed does start with "deep"
    Assert.assertSame(employee, employeeController
       .findEmployeeByEmail("deep@gitshah.com"));
    Assert.assertSame(employee, employeeController
       .findEmployeeByEmail("deep@packtpub.com"));

    //However, this next invocation would not return a valid
    //employee,
    //since the email address passed does not start with
    //"deep"
    Assert.assertNull(employeeController
       .findEmployeeByEmail("noreply@packtpub.com"));
}
```

6. The intent of this test is to verify that when we pass a certain e-mail address, the employee associated with that e-mail would be found. A few points to notice about this test are as follows:
   - The syntax we have used to match the arguments in the `PowerMockito.when()` method is a little different
   - Instead of passing the exact e-mail address, we are using the argument matcher `Mockito.startWith("deep")`
   - As the name suggests, this argument matcher will match any string that starts with the text `deep`.
   - This makes our mocks a little more flexible. Because of this, we are able to find an employee via the e-mail addresses `deep@gitshah.com` or `deep@packtpub.com`
   - However, in the last assert, we are not able to find the employee via the e-mail address `noreply@packtpub.com`, since it does not start with the text `deep`

7. Let's look at one more form of argument matcher:

```
@Test
public void shouldReturnNullIfNoEmployeeFoundByEmail() {
```

```
    final EmployeeService mock =
       PowerMockito.mock(EmployeeService.class);

    //No matter what email is passed
    //calling the findEmployeeByEmail on the
    //mocked EmployeeService instance is now going to return
    //null.
    PowerMockito.when(mock.findEmployeeByEmail(Mockito
       .anyString())).thenReturn(null);

    final EmployeeController employeeController = new
       EmployeeController(mock);

    Assert.assertNull(employeeController.findEmployeeByEmail
       ("deep@gitshah.com"));
    Assert.assertNull(employeeController.findEmployeeByEmail
       ("deep@packtpub.com"));
    Assert.assertNull(employeeController.findEmployeeByEmail
       ("noreply@packtpub.com"));
}
```

8. In this test, we want to assert that if the employee is not found, the method returns `null`:
   - Notice the `Mockito.anyString()` argument matcher. This argument matcher matches **any (all) strings** passed as an argument.
   - Because of this, when we invoke the `employeeController.findEmployeeByEmail` method passing various arguments, we always get the `null` response.

9. The final test we are going to see in this recipe is going to test the `EmployeeController.isEmployeeEmailAlreadyTaken` method:

```
@Test
public void shouldReturnTrueIfEmployeeEmailIsAlreadyTaken()
{
    final EmployeeService mock =
       PowerMockito.mock(EmployeeService.class);

    //A little more complex matcher using the
    //ArgumentMatcher class.
    //By implementing the matches method in this class we
    //can write any kind of complex logic
    //to validate that the correct arguments are being
    //passed.
    final String employeeEmail = "packt@gitshah.com";
    PowerMockito.when(mock.employeeExists(Mockito
```

```
            .argThat(new ArgumentMatcher<Employee>() {
        /**
         * This method currently only checks that
         * the email address set in the employee instance
         * matches the email address we passed to the
         * controller.
         * {@inheritDoc}
         */
           @Override
           public boolean matches(Object employee) {
              return ((Employee) employee).getEmail()
                 .equals(employeeEmail);
           }
        })).thenReturn(true);

        final EmployeeController employeeController = new
           EmployeeController(mock);
        Assert.assertTrue(employeeController
           .isEmployeeEmailAlreadyTaken(employeeEmail));
    }
```

10. This is a little more involved example. The following points should be noted:
    - This test uses the class `ArgumentMatcher` to match the arguments passed.
    - When using this class, we have to implement the `matches` method. Returning `true` from this method indicates that the arguments are matching.
    - In the preceding test, we verified that the instance of `Employee` passed to the `employeeService.employeeExists` method has the e-mail address equal to the value passed in the `employeeController.isEmployeeEmailAlreadyTaken` method.

How it works...

Argument matchers add great deal of flexibility to the tests. These matchers are powerful ways of matching what gets passed. That said, one has to be reasonable with custom matchers as they can make the tests less readable. Sometimes, it's just better to implement the `equals` method for the arguments.

Argument matchers can also be used for matching arguments in the `Mockito.verify` calls.

If we are using argument matchers for a certain method that takes more than one argument, all the arguments have to be provided using argument matchers.

For example, if we had a method `findEmployeeByFirstNameAndLastName` in `EmployeeController`, we would have to specify both the arguments using argument matchers to mock it. The following syntax is invalid since only the second argument uses an argument matcher:

```
PowerMockito.when(mock
    . findEmployeeByFirstNameAndLastName("Deep",
   Mockito.anyString())).thenReturn(null);
```

The correct syntax would be as follows:

```
PowerMockito.when(mock. findEmployeeByFirstNameAnd
LastName(Mockito.eq("Deep"), Mockito.anyString())).
thenReturn(null);
```

There's more...

There are various other built-in argument matchers that can be used.

Other built-in argument matchers

Some of the other built-in argument matchers are as follows:

- `Mockito.eq`: The matcher to verify that the argument is exactly equal to the passed value
- `Mockito.matches`: It matches the String argument using a regular expression
- `Mockito.any`: There are various `any` methods to match the different types of arguments, such as `anyBoolean`, `anyByte`, `anyShort`, `anyChar`, `anyInt`, `anyLong`, `anyFloat`, `anyDouble`, `anyList`, `anyCollection`, `anyMap`, `anySet`, and so on
- `Mockito.isNull`: It matches the `null` argument value for a certain class
- `Mockito.isNotNull`: It matches the not null argument value for a certain class
- `Mockito.isA`: It matches the argument value which is an instance of `A`
- `Mockito.endsWith`: Similar to `startsWith`, this argument matcher matches String arguments with values that end with the given value.

Understanding the Answer interface (Advanced)

In some edge cases, it might be impossible or impractical to create the mocks by simply using the `PowerMockito.when().thenReturn()` syntax. In such cases, the generic `Answer` interface could be very handy.

In this recipe, we will learn how to make use of the `Answer` interface to create some unusual mocking strategies.

Getting ready

Remember the `EmployeeController.findEmployeeByEmail` method we saw in the previous recipe? We are going to write one more test for that method in this recipe.

How to do it...

1. The requirement is as follows:
   - A valid employee would be found if the e-mail starts with `deep`
   - Or, if the e-mail address ends with packtpub.com
2. We can certainly use a custom argument matcher to write this test, but let's have a look at yet another method of achieving the same result using the `Answer` interface.
3. The test would be as follows:

```
@Test
public void
   shouldFindEmployeeByEmailUsingTheAnswerInterface() {
   final EmployeeService mock =
      PowerMockito.mock(EmployeeService.class);

   final Employee employee = new Employee();

   //Notice use of Answer interface.
   //Depending on what argument is passed we could either
   //return a valid employee
   //or return null.
   PowerMockito
      .when(mock.findEmployeeByEmail(Mockito.anyString()))
      .then(new Answer<Employee>() {
   /**
    * Implementing the answer method to return a valid
    * employee,
```

```
     * if email starts with "deep" or ends with
     * "packtpub.com"
     * in all other cases we return null.
     * {@inheritDoc}
     */
       @Override
       public Employee answer(InvocationOnMock invocation)
          throws Throwable {
          final String email = (String)
             invocation.getArguments()[0];
          if(email == null) return null;
          if(email.startsWith("deep"))
             return employee;
          if(email.endsWith("packtpub.com"))
             return employee;
          return null;
       }
    });

    final EmployeeController employeeController = new
       EmployeeController(mock);

    //Following 3 invocations will match and return valid
    //employee,
    //since the email address passed does start with "deep"
    //or ends with "packtpub.com"
    Assert.assertSame(employee, employeeController
       .findEmployeeByEmail("deep@gitshah.com"));
    Assert.assertSame(employee, employeeController
       .findEmployeeByEmail("deep@packtpub.com"));
    Assert.assertSame(employee, employeeController
       .findEmployeeByEmail("noreply@packtpub.com"));

    //However, this next invocation would not return a valid
    //employee,
    //since the email address passed does not start with
    //"deep" or ends with "packtpub.com"
    Assert.assertNull(employeeController.findEmployeeByEmail
       ("hello@world.com"));
}
```

4. Notice the use of the `PowerMockito.when().then()` syntax in the test.
5. The `then` method accepts an instance of the generic `Answer` interface. This interface has just one method called `answer`, which needs to be implemented.

6. In the implementation of the `answer` method, we get the arguments passed to the method invocation using the `InvocationOnMock.getArguments` method.
7. We then do our checks as follows:
    - If the argument passed is `null` then, no employee will be returned
    - If the argument passed starts with `deep` then, a valid employee will be returned
    - If the argument passed ends with `packtpub.com` then, a valid employee will be returned
    - In all other cases, no employee will be returned
8. Next, we put asserts in place:
    - In this test, the e-mail addresses `deep@gitshah.com`, `deep@packtpub.com`, and `noreply@packtpub.com` are all valid since they either start with the text `deep` or end with `packtpub.com`
    - However, the e-mail address `hello@world.com` is not valid since it does not match our criterion

How it works...

The `Answer` interface specifies an action to execute along with the return value. The return value is returned when the given method is invoked on the mock.

The instance of `InvocationOnMock` passed as an argument to the `answer` method of the `Answer` interface is quite resourceful. Using this we can do the following:

- `callRealMethod()`: Call the real method
- `getArguments()`: Get all arguments passed to the method invocation
- `getMethod()`: Return the method that was invoked on the mock instance
- `getMock()`: Get the mocked instance

There's more...

One of the overloads of `PowerMockito.mock` is that it takes in an instance of the generic `Answer` interface, which would act as the default answer to all un-stubbed methods.

PowerMockito.mock with default Answer

This instance of `Answer` passed to the `PowerMockito.mock` method will act as a catch for all clauses. This instance will be invoked when no specific dummy data has been set up for a given method.

Consider the following example:

```
@Test
public void shouldReturnCountOfEmployeesFromTheService
   WithDefaultAnswer() {
   //Creating a mock using the PowerMockito.mock method for the
   //EmployeeService class.
   EmployeeService mock = PowerMockito
      .mock(EmployeeService.class,
      /**
       * Passing in a default answer instance.
       * This method will be called when no matching mock
          methods have been setup.
       */
      new Answer() {
       /**
        * We are simply implementing the answer method of the
           interface
        * and returning hardcoded 10.
        * @param invocation The context of the invocation.
        * Holds useful information like what arguments where
           passed.
        * @return Object the value to return for this mock.
        */
         @Override
         public Object answer(InvocationOnMock invocation) {
            return 10;
         }
      });

   EmployeeController employeeController = new
      EmployeeController(mock);
   Assert.assertEquals(10, employeeController
      .getEmployeeCount());
}
```

In the preceding example, notice that we are not setting up any dummy data when the `getEmployeeCount` method of the `EmployeeService` mocked instance is invoked but because our default instance of `Answer` returns the hardcoded value `10`, our test case passes.

Partial mocking with spies (Advanced)

Sometimes, there is a real need when we want to mock only a few methods of a class, and invoke the real implementation of certain other methods of the same class. For such situations we could create spies. With spies, real methods will be invoked, unless we have explicitly mocked a method. This concept is also called as **partial mocking**.

In this recipe, we will look at the use of spies to partially mock a class.

Getting ready

The `EmployeeService.saveEmployee` method has started to look ugly. We can probably refactor it a little, by extracting out the create employee part in a separate method.

How to do it...

1. Currently, the `EmployeeService.saveEmployee` method is as follows:

```
 /**
  * The method that will save
  * the employee instance to the DB.
  * @param employee instance to save.
  */
public void saveEmployee(Employee employee) {
   if(employee.isNew()) {
      employee.setEmployeeId(EmployeeIdGenerator
         .getNextId());
      employee.create();
      WelcomeEmail emailSender = new
         WelcomeEmail(employee,
         "Welcome to Mocking with PowerMock How-to!");
      emailSender.send();
      return;
   }
employee.update();
}
```

2. The highlighted part of the code is responsible to create a new employee in the system. Let's extract it into a separate method called `createEmployee`.
3. The updated code would be as follows:

```
 /**
  * The method that will save
  * the employee instance to the DB.
  * @param employee instance to save.
```

```
     */
    public void saveEmployee(Employee employee) {
        if(employee.isNew()) {
            createEmployee(employee);
            return;
        }
        employee.update();
    }

    /**
     * The createEmployee method
     * extracted from the saveEmployee.
     * This method is only responsible
     * to do things that are required
     * to create a new employee.
     * @param employee instance to save.
     */
    void createEmployee(Employee employee) {
        employee.setEmployeeId(EmployeeIdGenerator
            .getNextId());
        employee.create();
        WelcomeEmail emailSender = new
            WelcomeEmail(employee,
            "Welcome to Mocking with PowerMock How-to!");
        emailSender.send();
    }
```

4. Looks better? Now, let's test the `saveEmployee` method (for the create flow), in such a way that we will only assert that it invokes the `EmployeeService.createEmployee` method.
5. We will assume that other things specific to create `Employee` flow are tested as part of testing the `createEmployee` method itself. This includes the following:
    - Generating the `EmployeeId`
    - Saving the employee
    - Sending the welcome e-mail
6. Basically, in this case we want to invoke the real `EmployeeService.saveEmployee` method but mock the `EmployeeService.createEmployee` method. Feels like an ideal candidate for using spies!
7. The test that achieves this requirement is as follows:

```
    @Test
    public void shouldInvokeTheCreateEmployeeMethod
        WhileSavingANewEmployee() {
```

```
        //Following is the syntax to create a spy using the
        //PowerMockito.spy method.
        //Notice that we have to pass an actual instance of
        //the EmployeeService class.
        //This is necessary since a spy will only mock few
        //methods of a class and
        //invoke the real methods for all methods that are
        //not mocked.
        final EmployeeService spy = PowerMockito.spy(new
            EmployeeService());

        final Employee employeeMock = PowerMockito
            .mock(Employee.class);
        PowerMockito.when(employeeMock.isNew())
            .thenReturn(true);

        //Notice that we have to use the PowerMockito
        //.doNothing().when(spy).createEmployee()
        //syntax to create the spy. This is required
        //because if we use the
        //PowerMockito.when(spy.createEmployee())
        //syntax will result in calling the actual method
        //on the spy.
        //Hence, remember when we are using spies,
        //always use the doNothing(), doReturn() or the
        //doThrow() syntax only.
        PowerMockito.doNothing().when(spy)
            .createEmployee(employeeMock);

        spy.saveEmployee(employeeMock);

        //Verification is simple enough and
        //we have to use the standard syntax for it.
        Mockito.verify(spy).createEmployee(employeeMock);
    }
```

8. There are a few things to notice about this test:
    - The first step is to create the `spy`. This is done by passing an instance of the class that we want to partially mock to the `PowerMockito.spy` method (in our case an instance of the `EmployeeService` class)
    - The next step is to mock the methods on the spy. Here, we have to use the `PowerMockito.doNothing().when(spy).createEmployee()` syntax
    - This tells PowerMock to **do nothing** when the `createEmployee` method is called on `spy`

- Next we invoke the method to test, which in this case is `saveEmployee`
- To verify that the `createEmployee` was called on `spy`, we just use our regular `Mockito.verify()` syntax

How it works...

Spies are designed in such a way that they will invoke real methods for all methods that are not mocked. This is the reason we need to pass in an instance of the class that is to be partially mocked (in our case the `EmployeeService` class) while creating a spy using the `PowerMockito.spy` method.

To mock any method on a spy, we have to necessarily use the `PowerMockito.doNothing()/doReturn()/doThrow()` syntax only. This is necessary because with the `PowerMockito.when().thenReturn()` syntax, PowerMock will not know whether we are setting up an expectation on the partial mock or actually invoking the real method. Hence, be careful while using spies with PowerMock, and remember to use the correct syntax.

Mocking private methods (Medium)

In a good Object Oriented Design, we will want to enforce the fact that some methods should be kept **private**. This poses a challenge for writing tests, since these private methods will not be visible outside the enclosing class. But with PowerMock we don't have to worry, as it can mock private methods as well.

In this recipe, we will learn to mock private methods.

Getting ready

In the previous recipe, we refactored the `EmployeeService.saveEmployee` method, and extracted the `EmployeeService.createEmployee` method. We kept this method visible at the package level.

How do we test the `EmployeeService.saveEmployee` method if `EmployeeService.createEmployee` was private?

How to do it...

1. Let's start by making the `EmployeeService.createEmployee` method private:

```
/**
 * The createEmployee method
 * extracted from the saveEmployee.
 * This method is only responsible
 * to do things that are required
```

```
     * to create a new employee.
     * @param employee instance to save.
     */
    private void createEmployee(Employee employee) {
        employee.setEmployeeId(EmployeeIdGenerator
            .getNextId());
        employee.create();
        WelcomeEmail emailSender = new
            WelcomeEmail(employee,
            "Welcome to Mocking with PowerMock How-to!");
        emailSender.send();
    }
```

2. The only change is in the method signature; we have made the method `private`.
3. Let's look at the modified test that enables us to mock, and verify the `createEmployee` private method:

```
@Test
public void shouldInvokeThePrivateCreateEmployeeMethod
    WhileSavingANewEmployee() throws Exception {
    final EmployeeService spy = PowerMockito.spy(new
        EmployeeService());

    final Employee employeeMock = PowerMockito
        .mock(Employee.class);
    PowerMockito.when(employeeMock.isNew())
        .thenReturn(true);
    //Since we cannot access the private method outside
    //the class,
    //We have to pass the name of the private method
    //along with the arguments passed
    //To the PowerMockito.doNothing().when() method.
    PowerMockito.doNothing().when(spy,
        "createEmployee", employeeMock);

    spy.saveEmployee(employeeMock);

    //Verification is similar to setting up the mock.
    //We have to inform PowerMock about which private
    //method to verify by invoking the
    //invoke method on PowerMockito.verifyPrivate().
    //The name of the private method along with its
    //arguments are passed to invoke method.
    PowerMockito.verifyPrivate(spy)
        .invoke("createEmployee", employeeMock);
}
```

4. A few things to notice about this test are as follows:
    - To mock the private method, we have to pass the method name along with its arguments to the `when` method. In our case, we have to pass the method name as `createEmployee`, and the argument as the mocked `Employee` class instance `employeeMock`.
    - To verify the invocation of the private method, we have to use the `PowerMockito.verifyPrivate().invoke()` syntax. Again, we have to pass the name of the private method and its arguments to the `invoke` method.

How it works...

PowerMock enables us to test our code without compromising our design in anyway. Its ability to mock private methods is one of its unique features that other mocking frameworks do not have.

Private methods cannot be accessed outside the enclosing class. This is the reason we see a slight variation in the syntax.

- The `PowerMockito.doNothing().when(spy, "createEmployee", employeeMock)` syntax tells PowerMock to do nothing, when the `createEmployee` method is called with `employeeMock` as argument
- The `PowerMockito.verifyPrivate(spy).invoke("createEmployee", employeeMock)` syntax tells PowerMock to verify the invocation of private method called `createEmployee` with `employeeMock` as argument

There's more...

Having the name of the private method passed as the argument while mocking and verification is not ideal; because of this, PowerMock provides one more alternate way of mocking and verifying private methods.

Alternate way of mocking private methods

PowerMock has overloaded methods that can help us mock and verify the private method without mentioning the method name.

1. Consider the following example:

```
@Test
public void shouldInvokeThePrivateCreateEmployeeMethod
    WithoutSpecifyingMethodName() throws Exception {
    final EmployeeService spy = PowerMockito.spy(new
        EmployeeService());
```

```
        final Employee employeeMock = PowerMockito
            .mock(Employee.class);
        PowerMockito.when(employeeMock.isNew())
            .thenReturn(true);

        //Finding the methods from EmployeeService class
        //that take Employee as their argument.
        final Method createEmployeeMethod = PowerMockito
            .method(EmployeeService.class, Employee.class);

        //Passing the method instance found in previous
        //step to the when method.
        //This sets up the mock on the private method.
        PowerMockito.doNothing().when(spy,
            createEmployeeMethod)
            .withArguments(employeeMock);

        spy.saveEmployee(employeeMock);

        //Verifying that the private method was indeed
        //invoked
        //using the same method instance we found earlier.
        PowerMockito.verifyPrivate(spy)
            .invoke(createEmployeeMethod)
            .withArguments(employeeMock);
    }
```

2. In the preceding example, notice the following points:
    - We find the private method to invoke using the `PowerMockito.method` method. This method takes in the class on which we want to find the private method and the argument types of the private method as arguments.
    - In our case, we want to find the private method on `EmployeeService.class` and which takes in an instance of `Employee.class` as argument.
    - Next, to mock this private method, we pass it as an argument to the overloaded `when` method. We can also specify what argument this private method takes using the `withArguments` method.
    - To verify the invocation of this private method, we can simply pass it to the `invoke` method.

The `PowerMockito.method` method will throw `TooManyMethodsFoundException` if the class (which encloses the private method) has more than one method with the same set of arguments but different names.

Basically, PowerMock does not make any assumptions about which method to mock. If there is more than one method matching the given signature, then we cannot use the previous approach. If we run the the preceding test with the current code base, it throws `TooManyMethodsFoundException`.

Breaking the encapsulation (Advanced)

Encapsulation is one of the fundamental principles of Object Oriented Programming. In a good Object Oriented Design, we will end up with some private methods that perform important business operations and fields that hold important state information. Sometimes, it might be important to test these in isolation. In this recipe, we will learn how to write tests for such situations.

Getting ready

Let's make our domain model a little richer. We will add a `Department` class. One employee will be associated with only one department, but a department can have many employees associated with it. In addition to this, `Department` will keep track of highest salary offered.

How to do it...

1. Let's start by looking at the `Department` class:

```
/**
 * The Department class that will
 * have a relationship with Employee class.
 * One Employee will be
 * associated with at max one Department,
 * but one Department will be associated with
 * one or more Employees
 *
 * @author Deep Shah
 */
public class Department {

    /**
     * The internal list of employee associated with
```

```
         department.
     */
    private List<Employee> employees = new
        ArrayList<Employee>();
    /**
     * The max salary offered by this department.
     */
    private long maxSalaryOffered;
    /**
     * The method to add a new employee to this department.
     * @param employee the instance to add to this
         departmnet.
     */
    public void addEmployee(final Employee employee) {
        employees.add(employee);
        updateMaxSalaryOffered();
    }
    /**
     * The private method that keeps track of
     * max salary offered by this department.
     */
    private void updateMaxSalaryOffered() {
        maxSalaryOffered = 0;
        for (Employee employee : employees) {
            if(employee.getSalary() > maxSalaryOffered) {
                maxSalaryOffered = employee.getSalary();
            }
        }
    }
}
```

2. As shown in the code, this class has two methods:
   - `addEmployee`: This method adds an employee to the `employees` list held privately
   - `updateMaxSalaryOffered`: This method updates the max salary offered by this department.

3. First, let's test the `addEmployee` method. In the test for `addEmployee`, we want to assert that the privately held employee list is updated correctly, and the passed-in employee is added to it. The test code is as follows:

```
@Test
public void shouldVerifyThatNewEmployeeIs
    AddedToTheDepartment() {
    final Department department = new Department();
```

```
        final Employee employee = new Employee();

        //Adding the employee to the department.
        department.addEmployee(employee);

        //Getting the privately held employees list
        //from the Department instance.
        final List<Employee> employees = Whitebox
            .getInternalState(department, "employees");

        //Asserting that the employee was added to the
        //list.
        Assert.assertTrue(employees.contains(employee));
    }

    @Test
    public void shouldAddNewEmployeeToTheDepartment() {
        final Department department = new Department();
        final Employee employee = new Employee();

        final ArrayList<Employee> employees = new
            ArrayList<Employee>();
        //Setting the privately held employees list with
        // our test employees list.
        Whitebox.setInternalState(department, "employees",
            employees);

        //Adding the employee.
        department.addEmployee(employee);

        //Since we substituted the privately held employees
        //within the department instance
        //we can simply assert whether our list has the
        // newly added employee or not.
        Assert.assertTrue(employees.contains(employee));
    }
```

4. We can test the `addEmployee` method using two approaches:
5. First Approach:
   - We are adding the `employee` to `Department`
   - Then getting the value of privately held `employees` list from the `Department` instance using the `Whitebox.getInternalState` method

- From the value retrieved, we assert that `employee` was actually added successfully

6. Second Approach:
   - We first substitute the privately held `employees` list within the `Department` instance using our own test list. This is done with the help of the `Whitebox.setInternalState` method.
   - Then we add the `employee` to the `Department` instance.
   - Finally, we assert that the `employee` was successfully added into our test `employees` list.

7. The last thing we are going to do in this recipe is write a test for the `Department.updateMaxSalaryOffered` method:

```
    @Test
    public void shouldVerifyThatMaxSalaryOfferedFor
        ADepartmentIsCalculatedCorrectly() throws Exception
{
        final Department department = new Department();
        final Employee employee1 = new Employee();
        final Employee employee2 = new Employee();
        employee1.setSalary(60000);
        employee2.setSalary(65000);

        //Adding two employees to the test employees list.
        final ArrayList<Employee> employees = new
            ArrayList<Employee>();
        employees.add(employee1);
        employees.add(employee2);

        //Substituting the privately held employees list
        // with our test list.
        Whitebox.setInternalState(department, "employees",
            employees);

        //Invoking the private method
        // updateMaxSalaryOffered on the department
        // instance.
        Whitebox.invokeMethod(department,
            "updateMaxSalaryOffered");

        //Getting the value of maxSalary from the private
        // field.
```

```
        final long maxSalary = Whitebox.getInternalState
            (department, "maxSalaryOffered");

        Assert.assertEquals(65000, maxSalary);
    }
```

8. Things to notice in this test are:
    - We are first setting up our test `employees` list by adding two `Employee` instances in it: one with a salary of `60000`, and other with a salary of `65000`
    - Next, we are substituting the privately held `employees` list in the `Department` instance with our test list using the `Whitebox.setInternalState` method
    - Then, using the `Whitebox.invokeMethod` method, we are invoking the `updateMaxSalaryOffered` method on the `Department` instance
    - Next, we get the value of `maxSalaryOffered` from the private field of the `Department` instance using the `Whitebox.getInternalState` method
    - Finally, we are asserting that `maxSalary` is equal to `65000`

How it works...

For any mutable object, the internal state may change after invoking any method. PowerMock helps us unit test such behavior using the `Whitebox` class.

- The `Whitebox.getInternalState(department, "employees")` method: This statement can be read as, get the value of private `employees` field from the `department` instance.
    - The first argument to this method is the instance which holds the private field (that is the `department` object)
    - The second argument is the name of private member (that is `employees`).
- The `Whitebox.setInternalState(department, "employees", employees)` method: This statement can be read as, set the `employees` field of the `department` instance with value held by the `employees` variable.
    - The first argument to this method is the instance which holds the private member (that is the `department` object).
    - The second argument is the name of the private member (that is `employees`).
    - The third argument is the value to set (that is the value held by the `employees` variable).

- The `Whitebox.invokeMethod(department, "updateMaxSalaryOffered")` method: This statement can be read as, invoke the method `updateMaxSalaryOffered` on the `department` instance.
    - The first argument to this method is the instance on which we need to invoke the private method (that is the `department` object).
    - The second argument is the name of the private method (that is `updateMaxSalaryOffered`).
    - The next arguments are any parameters that need to be passed to the private method. In our example, the private method `updateMaxSalaryOffered` does not take any parameters, hence we don't need to pass anything.

There's more...

There are many other methods to the `Whitebox` class. Using the `Whitebox` class, we can even instantiate a class that has a private constructor.

I strongly recommend visiting the API docs at `http://powermock.googlecode.com/svn/docs/powermock-1.5/apidocs/org/powermock/reflect/Whitebox.html` for more details.

Suppressing unwanted behavior (Advanced)

There are times when we may need to suppress a constructor, method, field, or static initializer because they perform some operations that are not very desirable for doing unit testing of their own code. Such a situation may arise while dealing with third-party libraries or legacy code.

In this recipe, we will look at ways to suppress such unwanted behaviors.

Getting ready

Let's modify the `Department` class and make it extend a base class called `BaseEntity`. This new `BaseEntity` class will make it difficult to unit test the `Department` class.

How to do it...

1. The code for the `BaseEntity` class is as follows:

```
/**
 * The base class which
 * is going to do things,
 * that would make its derived classes difficult to test.
```

```
 * @author Deep Shah
 */
public class BaseEntity {
   /**
    * The default constructor throws
    * UnsupportedOperationException.
    */
   public BaseEntity() {
      throw new UnsupportedOperationException();
   }
}
```

2. This is a very simple class whose default constructor throws `UnsupportedOperationException`.
3. The updated `Department` class is as follows:

```
/**
 * The Department class that will
 * have a relationship with Employee class.
 * One Employee will be
 * associated with at max one Department,
 * but one Department will be associated with
 * one or more Employees
 *
 * @author Deep Shah
 */
public class Department extends BaseEntity {
   /**
    * The department id field.
    */
   private int departmentId;

   /**
    * The constructor that takes in
    * departmentId as argument.
    * @param departmentId department id to set.
    */
   public Department(int departmentId) {
      super();
      this.departmentId = departmentId;
   }

//Rest of the code remains as it is.
}
```

4. The only change is that the `Department` class now extends `BaseEntity`. Since the `BaseEntity` constructor throws an exception, anyone who tries to instantiate the `Department` class will also get an exception.
5. We have also added a constructor that takes in `departmentId` as the argument. This constructor will first invoke the default constructor of the `BaseEntity` class, and then save the value of `departmentId` in its member field.
6. If we write the test for this constructor assignment without doing anything special, then we will get back an exception right in our face.
7. To test the `Department` class constructor, we will have to suppress the `BaseEntity` constructor. The code to do that is as follows:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(Department.class)
public class DepartmentTest {
   @Test
   public void shouldSuppressTheBase
      ConstructorOfDepartment() {
      PowerMockito.suppress(PowerMockito
         .constructor(BaseEntity.class));
      Assert.assertEquals(10, new
         Department(10).getDepartmentId());
   }
}
```

8. Two important things to note about this test are:
   - We will need to prepare the `Department` class for testing. This is required because we want to suppress the constructor of the `BaseEntity` class, which is invoked from the `Department` class.
   - Next, the syntax to suppress the constructor is `PowerMockito.suppress(PowerMockito.constructor(BaseEntity.class));`. This tells PowerMock that the constructor of `BaseEntity`, which does not take any argument, needs to be suppressed.
9. Let's add one more method to the `Department` class to set the department name. We would also want to audit this change and write to some log file. The code to audit the change will be placed in the `BaseEntity` class.

```
/**
 * Setter for the departmentId.
 * @return the value of departmentId.
 */
public void setName(String name) {
    this.name = name;
    super.performAudit(this.name);
}
```

10. `Department.setName` is a very simple setter method for updating department name. This method in turn calls `super.performAudit(this.name)` to do the auditing of department name.

```
/**
 * This method is responsible to write audit trails.
 * Currently this method throws an
 * UnsupportedOperationException.
 * @param auditInformation the audit information to
 * log.
 */
protected void performAudit(String auditInformation) {
    throw new UnsupportedOperationException();
}
```

11. The code to test `Department.setName` is as follows:

```
@Test
public void shouldSuppressThePerformAudit
    MethodOfBaseEntity() {
    PowerMockito.suppress(PowerMockito
        .constructor(BaseEntity.class));
    PowerMockito.suppress(PowerMockito
        .method(BaseEntity.class, "performAudit",
        String.class));
    final Department department = new Department();
    department.setName("Mocking with PowerMock");
    Assert.assertEquals("Mocking with PowerMock",
        department.getName());
}
```

12. The code looks very similar to what we have seen earlier:
    - First, we have to suppress the `BaseEntity` constructor
    - The `PowerMockito.suppress(PowerMockito.method(BaseEntity.class, "performAudit", String.class))` syntax tells PowerMock to suppress the `performAudit` method of the `BaseEntity` class, which takes in String as its only argument.
    - Then we test the `Department.setName` method

13. Let's do one last change to the `BaseEntity` class. We will add a static initializer to the `BaseEntity` class that will throw an exception:

```
public class BaseEntity {

    /**
     * Static initializer that will throw a
     * NullPointerException.
```

```
     */
    static {
        String x = null;
        x.toString();
    }
//Rest of the code is not shown in the interest of space.
}
```

14. As shown in the preceding code, the static initializer of `BaseEntity` will throw an instance of `NullPointerException`.
15. The test to suppress the static initializer of `BaseEntity` is as follows:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(Department.class)
@SuppressStaticInitializationFor("com.gitshah.powermock
    .BaseEntity")
public class DepartmentTest {

    @Test
    public void shouldSuppressTheInitializerForBaseEntity() {
        PowerMockito.suppress(PowerMockito
            .constructor(BaseEntity.class));
        Assert.assertNotNull(new Department());
    }
}
```

16. Things to notice about this test are as follows:
    - We need to inform PowerMock that we want to suppress the static initializer of the `BaseEntity` class. This is done by the annotation `@SuppressStaticInitializationFor("com.gitshah.powermock.BaseEntity")`.
    - Next, we simply suppress the default constructor of `BaseEntity` and assert that new instance of `Department` is not null.

How it works...

We have to inform PowerMock about the method to suppress by passing the method name as string. In our example, we are doing it as `PowerMockito.suppress(PowerMockito.method(BaseEntity.class, "performAudit", String.class));`.

The `PowerMockito.when().thenReturn()` syntax will not work when we want to suppress a method. That's because if we use the `PowerMockito.when().thenReturn()` syntax, it will result into a method invocation, and we don't want that to happen.

To suppress the static initializer, we had to give the fully qualified class name as an argument such as `@SuppressStaticInitializationFor("com.gitshah.powermock.BaseEntity")`. We cannot use `@SuppressStaticInitializationFor(BaseEntity.class)`, because as soon as we use the `BaseEntity`.class, it will load the class and call the static initializer. PowerMock will never get an opportunity to suppress the static initializer. Hence, to get around this problem, we have to pass fully qualified class name string to the `@SuppressStaticInitializationFor` annotation.

There's more...

PowerMock can suppress constructors, methods, and fields as well.

Suppressing constructors

`PowerMockito.suppress(PowerMockito.constructor(BaseEntity.class, String.class, Integer.class))`: This syntax would suppress a constructor in the `BaseEntity` class, which takes in `String` as its first argument and `Integer` as its second argument.

Suppressing fields

`PowerMockito.suppress(PowerMockito.field(BaseEntity.class, "identifier"))`: This tells PowerMock to suppress the field called `identifier` in the class `BaseEntity`. This syntax looks very similar to the suppress method syntax.

Thank you for buying
Instant Mock Testing with PowerMock

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

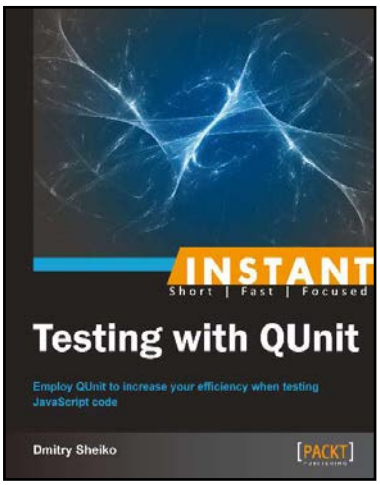
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING



Instant Testing with QUnit [Instant]

ISBN: 978-1-78328-217-3 Paperback: 64 pages

Employ QUnit to increase your efficiency when testing JavaScript code

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Learn about cross-browser testing with QUnit
3. Learn how to use popular QUnit plugins and develop your own plugins
4. Hands-on examples on all the essential QUnit methods

Web Services Testing with soapUI

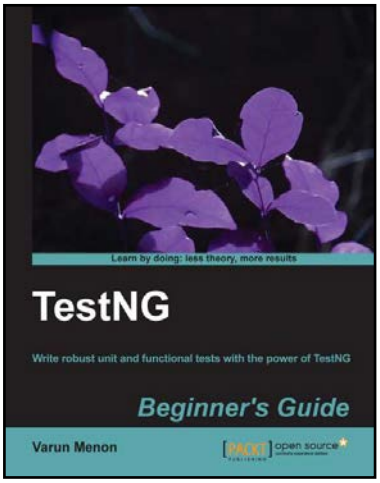
ISBN: 978-1-84951-566-5 Paperback: 332 pages

Build high quality service-oriented solutions by learning easy and efficient web services testing with this practical, hands-on guide

1. Become more proficient in testing web services included in your service-oriented solutions
2. Find, analyze, reproduce bugs effectively by adhering to best web service testing approaches
3. Learn with clear step-by-step instructions and hands-on examples on various topics related to web services testing using soapUI

Please check **www.PacktPub.com** for information on our titles

[PACKT] PUBLISHING

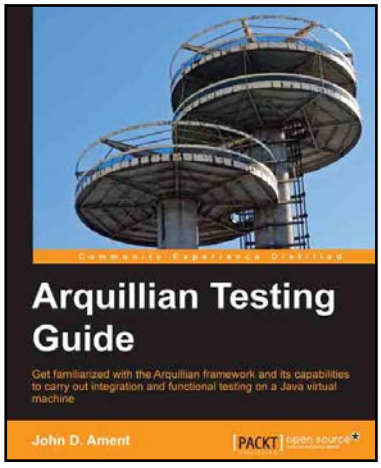


TestNG Beginner's Guide

ISBN: 978-1-78216-600-9 Paperback: 276 pages

Write robust unit and functional tests with the power of TestNG

1. Step-by-step guide to learn and practise any given feature
2. Detailed understanding of the features and core concepts
3. Learn about writing custom reporting



Arquillian Testing Guide

ISBN: 978-1-78216-070-0 Paperback: 242 pages

Get familiarized with the Arquillian framework and its capabilities to carry out integration and functional on a Java virtual machine

1. Build effective unit tests and integration using Arquillian and JUnit
2. Leverage Arquillian to test all aspects of your application – UI, SOAP and REST based applications
3. Run your tests the easy way using Arquillian in a container