



[ANDROID ▾](#)
[CORE JAVA ▾](#)
[DESKTOP JAVA ▾](#)
[ENTERPRISE JAVA ▾](#)
[JAVA BASICS ▾](#)
[JVM LANGUAGES ▾](#)
[SOFTWARE DEVELOPMENT ▾](#)
[DEVOPS ▾](#)

Home » Core Java » PowerMockito » PowerMockito Tutorial for Beginners

ABOUT MOHAMMAD MERAJ ZIA



I did my Engineering in Information Technology from IET, Lucknow, India. Currently doing MSc in Information Technology from Derby University. I have worked in Java/J2EE domain for the last 10 years. Have good understanding of Payment and Finance domains.



PowerMockito Tutorial for Beginners

Posted by: Mohammad Meraj Zia in PowerMockito May 31st, 2016 1 Comment 9613 Views

A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. In this example we will learn about PowerMockito which is an extension of Mockito. PowerMockito extends Mockito functionality with several new features such as mocking static and private methods and more. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

Table Of Contents

1. Introduction
2. Mocking
3. Creating a project
 - 3.1 Dependencies
4. Simple Example

- 4.1 Domain Class
- 4.2 User Service
- 4.3 User Controller
- 4.4 User Controller Test
5. Mock static methods
6. Mock private methods
7. Mock final classes
8. Mock constructor
9. Download the source file

1. Introduction

Writing unit tests can be hard and sometimes good design has to be sacrificed for the sole purpose of testability. Often testability corresponds to good design, but this is not always the case. For example final classes and methods cannot be used, private methods sometimes need to be protected or unnecessarily moved to a collaborator, static methods should be avoided completely and so on simply because of the limitations of existing frameworks.

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or can use the mocking frameworks like Mockito, EasyMock, jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

PowerMock is a framework that extends other mock libraries such as EasyMock with more powerful capabilities. PowerMock uses a custom classloader and bytecode manipulation to enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers and more. By using a custom classloader no changes need to be done to the IDE or continuous integration servers which simplifies adoption. Developers familiar with the supported mock frameworks will find PowerMock easy to use, since the entire expectation API is the same, both for static methods and constructors. PowerMock aims to extend the existing API's with a small number of methods and annotations to enable the extra features. Currently PowerMock supports EasyMock and Mockito.

NEWSLETTER

158,323 insiders are already receiving weekly updates and complimentary whitepapers!

Join them now to gain **ACCESS** to the latest news & insights about Android, Groovy and other related tech

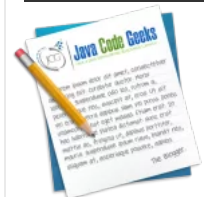
Email address:

Your email address

☒ Receive Java & Development news in your Area

Sign up

JOIN US



With 1,500 unique visitors a day, we are placed at the top of the related content. So if you are looking for unique content then you should check out

for creating mock/object/class and initiating verification, and expectations, everything else you can still use Mockito to setup and verify expectation (e.g.

```
times()
```

```
,  
anyInt()
```

). All usages require

```
@RunWith(PowerMockRunner.class)
```

and

```
@PrepareForTest
```

annotated at class level.

2. Mocking

Mocking is a way to test the functionality of a class in isolation. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

To understand how PowerMockito works first we need to look at few of the terms which we use when using these frameworks.

A *stub* class is an partial implementation for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually do responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls.

A *mock object* is a dummy implementation for an interface or a class in which you define the output of certain method calls. You can create these mock objects manually (via code) or use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects at runtime and define their behavior.

Below we define few differences between Mockito and PowerMock

- Mockito does not include specific language characteristics like constructors or static methods for mocking whereas PowerMock offers constructors and static methods to Mockito and other frameworks, through its individual classloader and bytecode management.
- Mockito does not require `@RunWith` annotation and base test class, while performing tests in suite whereas PowerMock requires both `@RunWith` annotation and a base test class for testing a suite.
- Mockito does not support mocking of constructors whereas PowerMock supports mocking of constructors and also supports mocking of (i) final (ii) static (iii) native and (iv) private methods.

PowerMock consists of two extension API's. One for EasyMock and one for Mockito. To use PowerMock you need to depend on one of these API's as well as a test framework. Currently PowerMock supports JUnit and TestNG. There are three different JUnit test executors available, one for JUnit 4.4+, one for JUnit 4.0-4.3 and one for JUnit 3. There's one test executor for TestNG which requires version 5.11+ depending on which version of PowerMock you use.

3. Creating a project

Below are the steps we need to take to create the project.

- Open Eclipse. Go to File=>New=>Java Project. In the 'Project name' enter 'PowerMockito'.

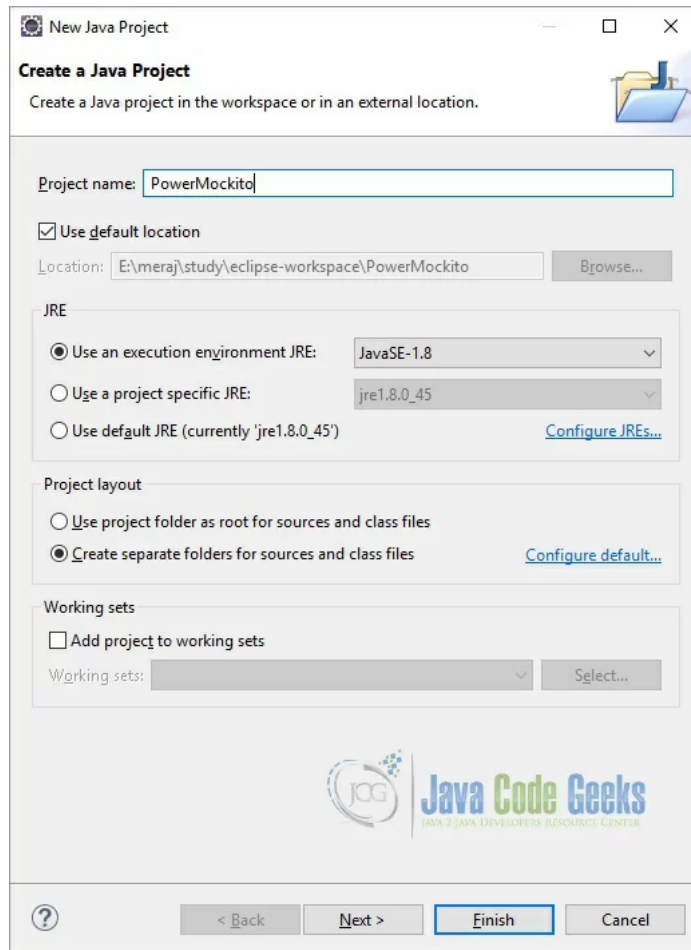


Figure 1. Create Java Project

- Eclipse will create a 'src' folder. Right click on the 'src' folder and choose New=>Package. In the 'Name' text-box enter 'com.javacodegeeks'. Click 'Finish'.

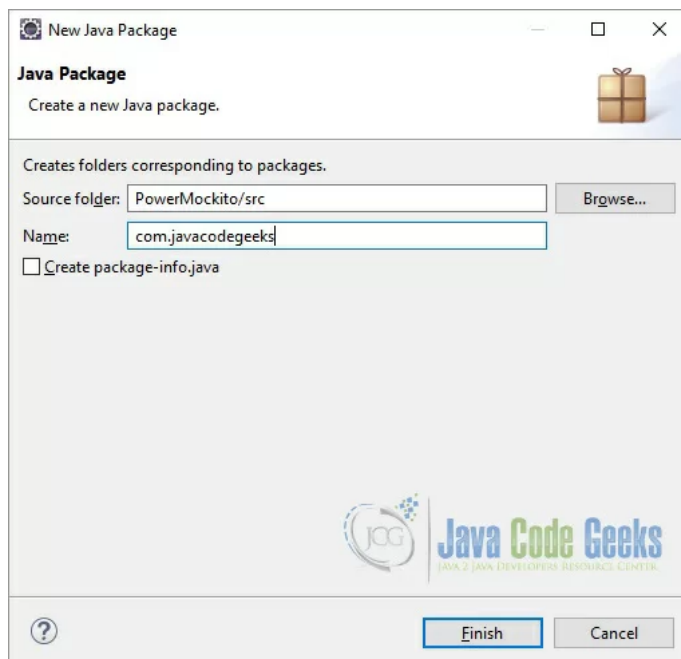


Figure 2. Java Package

3.1 Dependencies

For this example we need the below mentioned jars:

- calib-noden-3.2.2.jar

- junit-4.12.jar
- objenesis-2.2.jar
- powermock-api-easymock-1.6.5.jar
- powermock-mockito-release-full-1.6.4-full.jar

These jars can be downloaded from Maven repository. These are the latest (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path=>Configure Build Path. The click on the 'Add External JARs' button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

4. Simple Example

First we will see a simple example of using PowerMockito to mock methods. We will create a Controller and a service class. The controller class will have a reference to the service class which it will use to perform user actions.

4.1 Domain class

First we will see the User domain class. This class represents the User entity. (Please note that the actual User entity will have lot more fields but since here we are showing how to use PowerMock we are using a very simple User representation class.)

User.java

```
01 package com.javacodegeeks.user.domain;
02
03 /**
04  * Class representing User entity.
05  * @author Meraj
06  */
07
08 public class User {
09
10     private String firstName;
11     private String surname;
12
13     public String getFirstName() {
14         return firstName;
15     }
16     public void setFirstName(String firstName) {
17         this.firstName = firstName;
18     }
19     public String getSurname() {
20         return surname;
21     }
22     public void setSurname(String surname) {
23         this.surname = surname;
24     }
25 }
```

4.2 User Service

Now we will see how the User service class looks like. We will first create a class called

DefaultUserService

which will implements the

UserService

interface. Here we are interested in the

getUserCount()

method which throws the

UnsupportedOperationException

DefaultUserService.java

```
01 package com.javacodegeeks.user.service;
02
03 import com.javacodegeeks.user.domain.User;
04
05 /**
06  * Default implementation of {@link UserService}
07  * @author Meraj
08  */
09
10 public class DefaultUserService implements UserService {
11
12     @Override
13     public User getUserById(Long userId) {
14         return null;
15     }
16
17     @Override
18     public void updateUserDetails(User newUserDetails) {
19     }
```

```

26 |     public Long getUserCount() {
27 |         throw new UnsupportedOperationException("Not implemented");
28 |     }
29 | }

```

4.3 User Controller

This is the controller class for user related actions. It has the reference to the UserService class which gets sets when initiating the Controller class by calling it's constructor.

```

1 | public UserController(UserService userService) {
2 |     this.userService = userService;
3 | }

```

UserController.java

```

01 | package com.javacodegeeks.user.controller;
02 |
03 | import com.javacodegeeks.user.service.UserService;
04 |
05 | /**
06 |  * Controller class handling the user operations
07 |  * @author Meraj
08 |  */
09 |
10 | public class UserController {
11 |
12 |     private UserService userService;
13 |
14 |     public UserController(UserService userService) {
15 |         this.userService = userService;
16 |     }
17 |
18 |     public Long getUserCount() {
19 |         return userService.getUserCount();
20 |     }
21 | }

```

4.4 User Controller Test

Now we will write the test class for this controller class and will see how we can mock the Service class. There are few things which needs some attention here. We will use the

@RunWith

class annotation. This is defined in the junit library as shown below.

```

1 | @Retention(value=RUNTIME)
2 | @Target(value=TYPE)
3 | @Inherited
4 | public @interface RunWith

```

When a class is annotated with

@RunWith

or extends a class annotated with

@RunWith

, JUnit will invoke the class it references to run the tests in that class instead of the runner built into JUnit.

We will use the

org.powermock.modules.junit4.PowerMockRunner

class to run this tests. We will mock the

DefaultUserService

class by calling the

mock()

method of

org.powermock.api.mockito.PowerMockito

.

```

1 | PowerMockito.mock(DefaultUserService.class);

```

We will then pass this mock reference to the

UserController

to set the service.

UserControllerTest.java

```

01 | package com.javacodegeeks.user.test;

```

```

08 | import org.powermock.core.classloader.annotations.PrepareForTest;
09 | import org.powermock.modules.junit4.PowerMockRunner;
10 |
11 | import com.javacodegeeks.user.controller.UserController;
12 | import com.javacodegeeks.user.service.DefaultUserService;
13 |
14 | /**
15 |  * Test class for UserController
16 |  * @author Meraj
17 |  *
18 |  */
19 | @RunWith(PowerMockRunner.class)
20 | @PrepareForTest(UserController.class)
21 | public class UserControllerTest {
22 |
23 |     private DefaultUserService mockUserService;
24 |     private UserController userController;
25 |
26 |     @Test
27 |     public void testGetUserCount() {
28 |         mockUserService = PowerMockito.mock(DefaultUserService.class);
29 |         PowerMockito.when(mockUserService.getUserCount()).thenReturn(100L);
30 |         userController = new UserController(mockUserService);
31 |         assertEquals(100L, userController.getUserCount().longValue());
32 |     }
33 | }

```

5. Mock static methods

We can use PowerMock to mock static methods. In this section we will see how we can mock a static method using PowerMock. We will use

```
java.util.UUID
```

class for this. A UUID represents an immutable universally unique identifier (128 bit value). More details about this class can be found here: [UUID Java Docs](#). In this class there is a method called

```
randomUUID()
```

. It is used to retrieve a type 4 (pseudo randomly generated) UUID. The UUID is generated using a cryptographically strong pseudo random number generator.

We will create a simple method in the

```
UserController
```

class to create random user id.

```

1 | public String createUserId(User user) {
2 |     return String.format("%s_%s", user.getSurname(), UUID.randomUUID().toString());
3 | }

```

To tests this we will create a new test method in the

```
UserControllerTest
```

class.

```

1 | @Test
2 | public void testMockStatic() throws Exception {
3 |     PowerMock.mockStaticPartial(UUID.class, "randomUUID");
4 |     EasyMock.expect(UUID.randomUUID()).andReturn(UUID.fromString("067e6162-3b6f-4ae2-a171-2470b63dff00"));
5 |     PowerMock.replayAll();
6 |     UserController userController = new UserController();
7 |     Assert.assertTrue(userController.createUserId(getNewUser()).contains("067e6162-3b6f-4ae2-a171-2470b63dff00"));
8 |     PowerMock.verifyAll();
9 | }

```

First we will call the

```
mockStaticPartial()
```

method of the

```
org.powermock.api.easymock.PowerMock
```

class passing the class and the static method name as string which we want to mock:

```
1 | PowerMock.mockStaticPartial(UUID.class, "randomUUID");
```

Then we will define the expectation by calling the expect method of EasyMock and returning the test value of the random UUID.

```
1 | EasyMock.expect(UUID.randomUUID()).andReturn(UUID.fromString("067e6162-3b6f-4ae2-a171-2470b63dff00"));
```

Now we will call the

```
replayAll()
```

method of

```
PowerMock
```

```
PrepareForTest
```

or

```
PrepareOnlyThisForTest
```

annotations and all classes that have had their static initializers removed by using the

```
SuppressStaticInitializationFor
```

annotation. It also includes all mock instances created by PowerMock such as those created or used by

```
createMock(Class, Method...)
```

```
mockStatic(Class, Method...)
```

```
expectNew(Class, Object...)
```

```
createPartialMock(Class, String...)
```

etc.

To make it easy to pass in additional mocks not created by the PowerMock API you can optionally specify them as additionalMocks. These are typically those mock objects you have created using pure EasyMock or EasyMock class extensions. No additional mocks needs to be specified if you're only using PowerMock API methods. The additionalMocks are also automatically verified when invoking the

```
verifyAll()
```

method.

Now we will call the

```
createUserId()
```

method of the

```
UserController
```

class by passing in the test user details.

```
1 | Assert.assertTrue(userController.createUserId(getNewUser()).contains("067e6162-3b6f-4ae2-a171-2470b"))
```

In the end we will call the

```
verifyAll()
```

```
1 | PowerMock.verifyAll();
```

It verifies all classes and mock objects known by PowerMock. This includes all classes that are prepared for test using the

```
PrepareForTest
```

or

```
PrepareOnlyThisForTest
```

annotations and all classes that have had their static initializers removed by using the

```
SuppressStaticInitializationFor
```

annotation. It also includes all mock instances created by PowerMock such as those created or used by

```
createMock(Class, Method...)
```

```
mockStatic(Class, Method...)
```

```
expectNew(Class, Object...)
```

```
createPartialMock(Class, String...)
```

etc.

Note that all additionalMocks passed to the

```
replayAll(Object...)
```

6. Mock private methods

In this section we will see how we can mock a private method using PowerMock. It's a very useful feature which uses java reflection. We will define a public method in our UserController class as below:

```
1 | public String getGreetingText(User user) {
2 |     return String.format(getGreetingFormat(), user.getFirstName(), user.getSurname());
3 | }
```

As we can see that this method calls a private method

```
getGreetingFormat()
```

, which is defined as below:

```
1 | private String getGreetingFormat() {
2 |     return "Hello %s %s";
3 | }
```

We will try to change the behavior of this private method using PowerMock.

You can create spies of real objects. When you use the spy then the real methods are called (unless a method was stubbed). Spying on real objects can be associated with "partial mocking" concept. We will first create a spy on the

```
UserController
```

class.

```
1 | UserController spy = spy(new UserController());
```

Then we will define how the private method should behave when it's been called by using

```
org.powermock.api.mockito.PowerMockito.when
```

```
1 | when(spy, method(UserController.class, "getGreetingFormat")).withNoArguments().thenReturn("Good Mor
```

Now we will call the public method (which uses the private method) on the spied object

```
1 | assertEquals("Good Morning Code Geeks", spy.getGreetingText(user));
```

Below is the code snippet of the whole method

```
1 | @Test
2 | public void testMockPrivateMethod() throws Exception {
3 |     UserController spy = spy(new UserController());
4 |     when(spy, method(UserController.class, "getGreetingFormat")).withNoArguments().thenReturn("Good M
5 |     User user = new User();
6 |     user.setFirstName("Code");
7 |     user.setSurname("Geeks");
8 |     assertEquals("Good Morning Code Geeks", spy.getGreetingText(user));
9 | }
```

7. Mock final classes

EasyMock does not allow you to mock a final class but PowerMock does. You can simply do

```
ExampleFinalClass clazz = PowerMock.createMock(ExampleFinalClass.class);
```

Below is the list of this which we need to do for mocking a final class:

1. Use the `@RunWith(PowerMockRunner.class)` annotation at the class-level of the test case.
2. Use the `@PrepareForTest(ClassWithFinal.class)` annotation at the class-level of the test case.
3. Use `PowerMock.createMock(FinalClazz.class)` to create a mock object for all methods of this class (let's call it mockObject).
4. Use `PowerMock.replay(mockObject)` to change the mock object to replay mode.
5. Use `PowerMock.verify(mockObject)` to change the mock object to verify mode.

First we will create a very simple final class with just one final method.

SimpleFinalClazz.java

```
1 | package com.javacodegeeks.powermock;
2 |
3 | public final class SimpleFinalClazz {
4 |
5 |     public final String simpleFinalMethod() {
6 |         return "Final String";
7 |     }
8 |
9 | }
```

Now we will define another class which will use this class.

FinalClazzUser.java


```

05 | private SimpleFinalClazz simpleFinalClazz;
06 |
07 | public FinalClazzUser(SimpleFinalClazz simpleFinalClazz) {
08 |     this.simpleFinalClazz = simpleFinalClazz;
09 | }
10 |
11 | public String getMeSomething() {
12 |     return "Get Me Something " + simpleFinalClazz.simpleFinalMethod();
13 | }
14 |
15 | }

```

Now we will see how we can mock the final class SimpleFinalClazz to change its behavior.

```

01 | @Test
02 | public void testMockFinal() {
03 |     SimpleFinalClazz simpleFinalClazz = PowerMock.createMock(SimpleFinalClazz.class);
04 |     FinalClazzUser finalClazzUser = new FinalClazzUser(simpleFinalClazz);
05 |     EasyMock.expect(simpleFinalClazz.simpleFinalMethod()).andReturn("Hurray!!!");
06 |     PowerMock.replay(simpleFinalClazz);
07 |     String actual = finalClazzUser.getMeSomething();
08 |     PowerMock.verify(simpleFinalClazz);
09 |     Assert.assertEquals(actual, "Get Me Something Hurray!!!");
10 | }

```

First we will call the createMock() method of PowerMock and will pass the SimpleFinalClazz reference as the parameter to mock it.

```
1 | SimpleFinalClazz simpleFinalClazz = PowerMock.createMock(SimpleFinalClazz.class);
```

Then we will create the instance of FinalClazzUser class by passing the mock object created above.

```
1 | FinalClazzUser finalClazzUser = new FinalClazzUser(simpleFinalClazz);
```

Then we will define the expectation of the method defined in the final class.

```
1 | EasyMock.expect(simpleFinalClazz.simpleFinalMethod()).andReturn("Hurray!!!");
```

Now we will replay the mock object.

```
1 | PowerMock.replay(simpleFinalClazz);
```

Now we will verify the expected and actual behavior

```

1 | PowerMock.verify(simpleFinalClazz);
2 | Assert.assertEquals(actual, "Get Me Something Hurray!!!");

```

8. Mock constructor

One of the biggest problems in writing unit-tests is the profligate use of "new" — i.e, constructors. When the class you want to test in turn explicitly constructs its dependencies, you may have a big problem. In this section we will see how we can use PowerMockito to mock constructors.

First we will define a simple class SimpleClass with one method getMeCurrentDateAsString()

```

01 | package com.javacodegeeks.powermock.constructor;
02 |
03 | import java.util.Calendar;
04 |
05 | public class SimpleClass {
06 |
07 |     @SuppressWarnings("deprecation")
08 |     public String getMeCurrentDateAsString() {
09 |         return Calendar.getInstance().getTime().toGMTString();
10 |     }
11 | }
12 |
13 | }

```

Now we will define another class PowerMockConstructorExample which has the reference to the first class (SimpleClass)

```

01 | package com.javacodegeeks.powermock.constructor;
02 |
03 | public class PowerMockConstructorExample {
04 |
05 |     public String getMeSimpleObject() {
06 |         SimpleClass simpleClass = new SimpleClass();
07 |
08 |         String returnValue = simpleClass.getMeCurrentDateAsString();
09 |         return returnValue;
10 |     }
11 | }
12 |

```

Now we will see how we can mock the

SimpleClass

class.

Annotate the SimpleClass reference with

@Mock

Now use the

```
expectNew()
```

method of PowerMock to set the mock class instead of the real one. It allows specifying expectations on new invocations. Note that you must replay the class when using this method since this behavior is part of the class mock.

```
1 | expectNew(SimpleClass.class).andReturn(mockSimpleClass);
```

Now we will define the method expectation like below

```
1 | expect(mockSimpleClass.getMeCurrentDateAsString()).andReturn("Mock Result");
```

Now we will replay the mock class.

```
1 | replay(SimpleClass.class, mockSimpleClass);
```

Now we verify the result

```
1 | String value = instance.getMeSimpleObject();
2 | verify(SimpleClass.class, mockSimpleClass);
3 | assertEquals("Mock Result", value);
```

Below is the full test class which we used to test this:

PowerMockConstructorExampleTest.java

```
01 | package com.javacodegeeks.powermock.constructor;
02 |
03 | import static org.easymock.EasyMock.expect;
04 | import static org.powermock.api.easymock.PowerMock.expectNew;
05 | import static org.powermock.api.easymock.PowerMock.replay;
06 |
07 | import org.junit.Test;
08 | import org.junit.runner.RunWith;
09 | import org.powermock.api.easymock.annotation.Mock;
10 | import org.powermock.core.classloader.annotations.PrepareForTest;
11 | import org.powermock.modules.junit4.PowerMockRunner;
12 | import static org.powermock.api.easymock.PowerMock.verify;
13 | import static org.junit.Assert.assertEquals;
14 |
15 | @RunWith(PowerMockRunner.class)
16 | @PrepareForTest(PowerMockConstructorExample.class)
17 | public class PowerMockConstructorExampleTest {
18 |
19 |     @Mock private SimpleClass mockSimpleClass;
20 |
21 |     private PowerMockConstructorExample instance;
22 |
23 |     @Test
24 |     public void testMockConstructor() throws Exception {
25 |         instance = new PowerMockConstructorExample();
26 |         expectNew(SimpleClass.class).andReturn(mockSimpleClass);
27 |
28 |         expect(mockSimpleClass.getMeCurrentDateAsString()).andReturn("Mock Result");
29 |
30 |         replay(SimpleClass.class, mockSimpleClass);
31 |         String value = instance.getMeSimpleObject();
32 |         verify(SimpleClass.class, mockSimpleClass);
33 |         assertEquals("Mock Result", value);
34 |     }
35 |
36 | }
```

9. Download the source file

In this article we saw the usages of PowerMock and how it provides extra features which Mockito/EasyMock doesn't provide..

Download

You can download the full source code of this example here : **PowerMockito**



(No Ratings Yet) 1 Comment 9613 Views Tweet it!

Do you want to know how to develop your skillset to become a **Java Rockstar**?

Subscribe to our newsletter to start Rocking right now!

To get you started we give you our best selling eBooks for **FREE!**


1. JPA Mini Book
2. JVM Troubleshooting Guide
3. JUnit Tutorial for Unit Testing
4. Java Annotations Tutorial
5. Java Interview Questions
6. Spring Interview Questions
7. Android UI Design

☒ Receive Java & Developer job alerts in your Area

[Sign up](#)

LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS

1 [Leave a Reply](#)

 Join the discussion...

 1

 0

 1





 1




[Subscribe](#) ▼

▲ newest

▲ oldest

▲ most voted



Sir Montes

[Link icon](#)

The testMockConstructor method failed due NullPointerException on line Nr.28. Fixed by this statement after line 25: mockSimpleClass = mock(SimpleClass.class);

+ 1 -

[Reply](#)

🕒 5 months ago

JCGs (Java Code Geeks) is an independent online community focused on ultimate Java to Java developers resource center; targeted at the technical team lead (senior developer), project manager and junior dev. JCGs serve the Java, SOA, Agile and Telecom communities with daily domain experts, articles, tutorials, reviews, announcements, code snippets and source projects.

DISCLAIMER

All trademarks and registered trademarks appearing on Java Code Ge property of their respective owners. Java is a trademark or registered tr Oracle Corporation in the United States and other countries. Examples is not connected to Oracle Corporation and is not sponsored by Oracle

THE CODE GEEKS NETWORK

Spring JdbcTemplate Example