**Unit tests with Mockito - Tutorial**

**Lars Vogel**

Version 1.7

Copyright © 2012-2016 vogella GmbH

29.04.2016

**Testing with Mockito**

This tutorial explains testing with the Mockito framework for writting software tests.

**Table of Contents**

[**1. Prerequisites**](http://www.vogella.com/tutorials/Mockito/article.html#prerequisites)

[**2. Testing with test doubles**](http://www.vogella.com/tutorials/Mockito/article.html#testingwithmocks)

[**2.1. Target and challenge of unit testing**](http://www.vogella.com/tutorials/Mockito/article.html#testingwithmocks_targets)

[**2.2. Classifications of different test classes**](http://www.vogella.com/tutorials/Mockito/article.html#stubsvsmockobject)

[**2.3. Mock object generation**](http://www.vogella.com/tutorials/Mockito/article.html#usingstubsandmocks)

[**3. Mocking frameworks**](http://www.vogella.com/tutorials/Mockito/article.html#mockframeworks)

[**4. Introduction to Mockito for mocking objects**](http://www.vogella.com/tutorials/Mockito/article.html#mockitointroduction)

[**5. Adding Mockito as dependencies to a project**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation)

[**5.1. Using Grade**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation_gradle)

[**5.2. Using Maven**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation_maven)

[**5.3. Using Eclipse**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation_eclipse)

[**5.4. Using the Orbit P2 Update Site**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation_p2)

[**6. Using the Mockito API**](http://www.vogella.com/tutorials/Mockito/article.html#mockitousage)

[**6.1. Creating and configuring mock objects**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_usage)

[**6.2. Verify the calls on the mock objects**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_verify)

[**6.3. Spy**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_spy)

[**6.4. Using @InjectMocks for dependency injection via Mockito**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_dependencyinjection)

[**6.5. Creating mock objects based on annotations**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_annotations)

[**6.6. Limitations**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_limitations)

[**7. Using Mockito on Android**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_android)

[**8. Exercise: Write an instrumented unit test using Mockito**](http://www.vogella.com/tutorials/Mockito/article.html#exercise_mockito_android)

[**8.1. Create Application under tests on Android**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_android1)

[**8.2. Add the Mockito dependency to the app/build.gradle file**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_android2)

[**8.3. Create test**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_android_createtest)

[**9. About this website**](http://www.vogella.com/tutorials/Mockito/article.html#supportandlicense)

[**10. Links and Literature**](http://www.vogella.com/tutorials/Mockito/article.html#resources)

[**10.1. Mockito resources**](http://www.vogella.com/tutorials/Mockito/article.html#resources_mockito)

[**10.2. vogella GmbH training and consulting support**](http://www.vogella.com/tutorials/Mockito/article.html#resources_general)

**1. Prerequisites**

The following tutorial requires an understanding of unit testing with the JUnit framework.

In case your are not familiar with JUnit, please check the following **[JUnit Tutorial](http://www.vogella.com/tutorials/JUnit/article.html" \t "_top)**.

**2. Testing with test doubles**

**2.1. Target and challenge of unit testing**

A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. The achievement of this desired goal is typical complicated by the fact that Java classes usually depend on other classes.

To solve this, you can use test doubles.

**2.2. Classifications of different test classes**

A *dummy object* is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.

*Fake* objects have working implementations, but are usually simplified, for example they use an in memory database and not a real database.

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.

Test doubles can be passed to other objects which are tested. Your tests can validate that the class reacts correctly during tests, i.e., you can validate if certain methods on the mock object were called. This helps to ensure that you only test the class while running tests and that your tests are not affected by any side effects.

Mock objects are typically configured. Mock objects typically require less code to configure and should therefore be preferred.

For a detailed discussion on this topic, see [**Mocks Aren't Stubs from Martin Fowler**](http://martinfowler.com/articles/mocksArentStubs.html).

**2.3. Mock object generation**

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.

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.

These mock objects can be provided to the class which is tested. Therefore, the class to be tested should avoid any hard dependency on external data.

Mocking or mock frameworks allows testing the expected interaction with the mock object, e.g., you test which methods have been called on the mock object.

**3. Mocking frameworks**

Mocking frameworks make the creation of mock objects as simple as possible. Popular mock frameworks are EasyMock, jMock and Mockito.

The following lists the links to these frameworks.

# jMock

http://jmock.org/

# EasyMock

http://easymock.org/

# Mockito

https://github.com/mockito/mockito

**4. Introduction to Mockito for mocking objects**

[**Mockito**](http://site.mockito.org/) is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

**5. Adding Mockito as dependencies to a project**

**5.1. Using Grade**

If you use Gradle as your build system, add the following dependency to your build file.

repositories { jcenter() }

dependencies { testCompile "org.mockito:mockito-core:1.+" }

**5.2. Using Maven**

Maven users can declare a dependency. Search for g:"org.mockito", a:"mockito-core" via the [**Maven search website**](http://search.maven.org/) to find the correct pom entry.

**5.3. Using Eclipse**

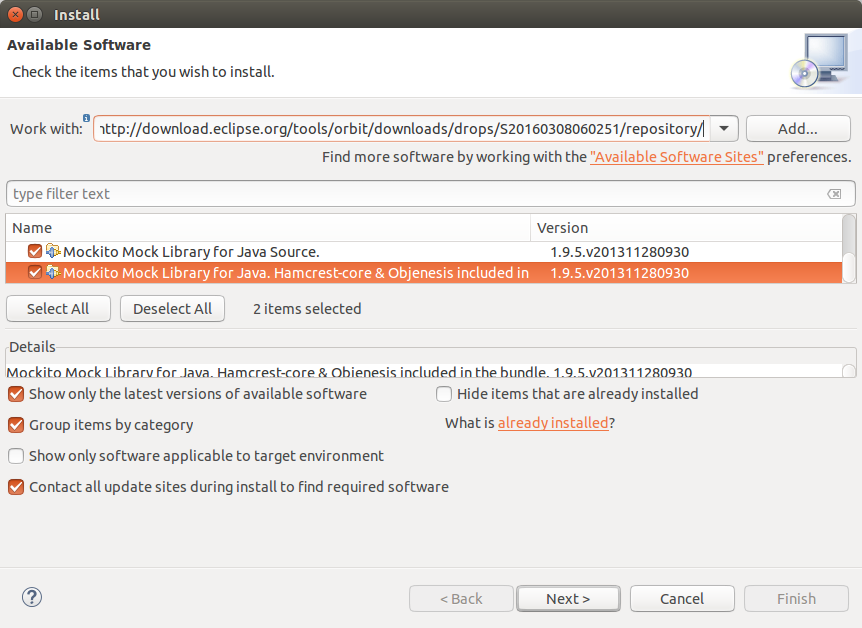
The Eclipse IDE supports Gradle as well as Maven. Mockito does not provide a "all" download in its latest version, therefore for the usage of Mockito you are adviced to use either Gradle or Maven as build system.

**5.4. Using the Orbit P2 Update Site**

In Eclipse RCP applications dependencies are usually obtained from p2 update sites.

The Orbit Repositories are a good source for third party libraries, which can be included in Eclipse RCP applications.

The Orbit Repositories can be found here: [**Orbit Downloads**](http://download.eclipse.org/tools/orbit/downloads/).



**6. Using the Mockito API**

**6.1. Creating and configuring mock objects**

Mockito supports the creation of mock objects with the static mock(). If you add a static import fororg.mockito.Mockito.\*;, you can access Mockitos methods like mock() directly. Static imports allows you to call static members, i.e., methods and fields of a class directly without specifying the class.

To configure which values are returned at a method call *Mockito* defines a fluent API. You can use the verify() method to ensure that a method was called.

The when(....).thenReturn(....) call chain can be used to specify a condition and a return value for this condition. If you specify more than one value, they are returned in the order of specification until the last one is used. Afterwards the last specified value is returned. Mocks can also return different values depending on arguments passed into a method. You also use methods like anyString or anyInt to define that independent of the input value a certain return value should be returned.

**import** **static** org.mockito.Mockito.\*;

**import** **static** org.junit.Assert.\*;

*@Test*

**public** **void** test1() {

*// create mock*

MyClass test = Mockito.mock(MyClass.**class**);

*// define return value for method getUniqueId()*

when(test.getUniqueId()).thenReturn(43);

*// use mock in test....*

assertEquals(test.getUniqueId(), 43);

}

*// Demonstrates the return of multiple values*

*@Test*

**public** **void** testMoreThanOneReturnValue() {

Iterator i= mock(Iterator.**class**);

when(i.next()).thenReturn("Mockito").thenReturn("rocks");

String result=i.next()+" "+i.next();

*//assert*

assertEquals("Mockito rocks", result);

}

*// this test demonstrates how to return values based on the input*

*@Test*

**public** **void** testReturnValueDependentOnMethodParameter() {

Comparable c= mock(Comparable.**class**);

when(c.compareTo("Mockito")).thenReturn(1);

when(c.compareTo("Eclipse")).thenReturn(2);

*//assert*

assertEquals(1,c.compareTo("Mockito"));

}

*// this test demonstrates how to return values independent of the input value*

*@Test*

**public** **void** testReturnValueInDependentOnMethodParameter() {

Comparable c= mock(Comparable.**class**);

when(c.compareTo(anyInt())).thenReturn(-1);

*//assert*

assertEquals(-1 ,c.compareTo(9));

}

*// return a value based on the type of the provide parameter*

*@Test*

**public** **void** testReturnValueInDependentOnMethodParameter() {

Comparable c= mock(Comparable.**class**);

when(c.compareTo(isA(Todo.**class**))).thenReturn(0);

*//assert*

Todo todo = **new** Todo(5);

assertEquals(todo ,c.compareTo(**new** Todo(1)));

}

The doReturn(...).when(...).methodCall call chain works similar but is useful for void methods. The doThrow variant can be used for methods which return void to throw an exception. This usage is demonstrated by the following code snippet.

**import** **static** org.mockito.Mockito.\*;

**import** **static** org.junit.Assert.\*;

*// this test demonstrates how use doThrow*

*@Test(expected=IOException.class)*

**public** **void** testForIOException() {

*// create an configure mock*

OutputStream mockStream = mock(OutputStream.**class**);

doThrow(**new** IOException()).when(mockStream).close();

*// use mock*

OutputStreamWriter streamWriter= **new** OutputStreamWriter(mockStream);

streamWriter.close();

}

**6.2. Verify the calls on the mock objects**

Mockito keeps track of all the method calls and their parameters to the mock object. You can use the verify() method on the mock object to verify that the specified conditions are met, i.e., that a method has been called with certain parameters. This kind of testing is sometimes called *behavior testing*, because it does not check the result of a method call, but it checks that a method is called with the right parameters.

**import** **static** org.mockito.Mockito.\*;

*@Test*

**public** **void** testVerify() {

*// create and configure mock*

MyClass test = Mockito.mock(MyClass.**class**);

when(test.getUniqueId()).thenReturn(43);

*// call method testing on the mock with parameter 12*

test.testing(12);

test.getUniqueId();

test.getUniqueId();

*// now check if method testing was called with the parameter 12*

verify(test).testing(Matchers.eq(12));

*// was the method called twice?*

verify(test, times(2)).getUniqueId();

*// other alternatives for verifiying the number of method calls for a method*

verify(mock, never()).someMethod("never called");

verify(mock, atLeastOnce()).someMethod("called at least once");

verify(mock, atLeast(2)).someMethod("called at least twice");

verify(mock, times(5)).someMethod("called five times");

verify(mock, atMost(3)).someMethod("called at most 3 times");

}

**6.3. Spy**

@Spy or the spy() method can be used to wrap a real object. Every call, unless specified otherwise, is delegated to the object.

**import** **static** org.mockito.Mockito.\*;

*// Lets mock a LinkedList*

List list = **new** LinkedList();

List spy = spy(list);

*//You have to use doReturn() for stubbing*

doReturn("foo").when(spy).get(0);

*// this would not work*

*// real method is called so spy.get(0)*

*// throws IndexOutOfBoundsException (list is still empty)*

when(spy.get(0)).thenReturn("foo");

The verifyNoMoreInteractions() allows you to check that no other method was called.

**6.4. Using @InjectMocks for dependency injection via Mockito**

You also have the @InjectMocks annotation which tries to do constructor, method or field dependency injection based on the type. The following code is a slightly modified example from the Javadoc.

*// Mockito can construct this class via constructor*

**public** **class** ArticleManager {

ArticleManager(ArticleCalculator calculator, ArticleDatabase database) {

}

}

*// Mockito can also perform method injection*

**public** **class** ArticleManager {

ArticleManager() { }

**void** setDatabase(ArticleDatabase database) { }

**void** setCalculator(ArticleCalculator calculator) { }

}

*// Mockito can also perform field injection*

**public** **class** ArticleManager {

**private** ArticleDatabase database;

**private** ArticleCalculator calculator;

}

*@RunWith(MockitoJUnitRunner.class)*

**public** **class** ArticleManagerTest {

*@Mock* **private** ArticleCalculator calculator;

*@Mock* **private** ArticleDatabase database;

*@Spy* **private** UserProvider userProvider = **new** ConsumerUserProvider();

*// creates instance of ArticleManager*

*// and performs constructor injection on it*

*@InjectMocks* **private** ArticleManager manager;

*@Test* **public** **void** shouldDoSomething() {

*// assume that ArticleManager has a method called initialize which calls a method*

*// addListener with an instance of ArticleListener*

manager.initialize();

*// validate that addListener was called*

verify(database).addListener(any(ArticleListener.**class**));

}

}

For more information see the **[Javadoc of InjectMock](http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/InjectMocks.html" \t "_top)**.

**6.5. Creating mock objects based on annotations**

Mockito also supports the creation of mock objects based on the @Mock

If you use annotations, you must initialize this mock objects. The MockitoRule allows this, it invokes the static methodMockitoAnnotations.initMocks(this) to populate the annotated fields. Alternatively you can use@RunWith(MockitoJUnitRunner.class).

**import** **static** org.mockito.Mockito.\*;

**public** **class** MockitoTest {

*// assume there is a class MyDatabase*

*@Mock*

MyDatabase databaseMock;

*@Rule* **public** MockitoRule mockitoRule = MockitoJUnit.rule();

*@Test*

**public** **void** testQuery() {

*// assume there is a class called ClassToTest*

*// which could be tested*

ClassToTest t = **new** ClassToTest(databaseMock);

*// call a method*

**boolean** check = t.query("\* from t");

*// test the return type*

assertTrue(check);

*// test that the query() method on the*

*// mock object was called*

verify(databaseMock).query("\* from t");

}

}

**6.6. Limitations**

Mockito has certain limitations. It can not test the following constructs:

* final classes
* anonymous classes
* primitive types

**7. Using Mockito on Android**

Mockito can also be directly used in Android unit tests simply by adding the dependency to Mockito to the Gradle build file of the application. To use it in instrumented Android tests (since the release 1.9.5). Which requires that dexmaker and dexmaker-mockito are also added as dependency. in the Gradle build file.

dependencies {

testCompile 'junit:junit:4.12'

// required if you want to use Mockito for unit tests

testCompile 'org.mockito:mockito-core:1.+'

// required if you want to use Mockito for Android instrumentation tests

androidTestCompile 'org.mockito:mockito-core:1.+'

androidTestCompile "com.google.dexmaker:dexmaker:1.2"

androidTestCompile "com.google.dexmaker:dexmaker-mockito:1.2"

}

**8. Exercise: Write an instrumented unit test using Mockito**

**8.1. Create Application under tests on Android**

Create an Android application with the package name com.vogella.android.testing.mockito.contextmock. Add a static method which allows to create an intent with certain parameters as in the following example.

**public** **static** Intent createQuery(Context context, String query, String value) {

*// Reuse MainActivity for simplification*

Intent i = **new** Intent(context, MainActivity.**class**);

i.putExtra("QUERY", query);

i.putExtra("VALUE", value);

**return** i;

}

**8.2. Add the Mockito dependency to the app/build.gradle file**

dependencies {

*// the following is required to use Mockito and JUnit for your*

*// instrumentation unit tests on the JVM*

androidTestCompile 'junit:junit:4.12'

androidTestCompile 'org.mockito:mockito-core:1.+'

androidTestCompile 'com.android.support.test:runner:0.3'

androidTestCompile "com.google.dexmaker:dexmaker:1.2"

androidTestCompile "com.google.dexmaker:dexmaker-mockito:1.2"

*// the following is required to use Mockito and JUnit for your unit*

*// tests on the JVM*

testCompile 'junit:junit:4.12'

testCompile 'org.mockito:mockito-core:1.+'

}

**8.3. Create test**

Create a new unit test using Mockito to check that the intent is triggered with the correct extra data.

For this you mock the Context object with Mockito as in the following example.

**package** com.vogella.android.testing.mockitocontextmock;

**import** android.content.Context;

**import** android.content.Intent;

**import** android.os.Bundle;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.mockito.Mockito;

**import** **static** org.junit.Assert.assertEquals;

**import** **static** org.junit.Assert.assertNotNull;

**public** **class** TextIntentCreation {

*@Test*

**public** **void** testIntentShouldBeCreated() {

Context context = Mockito.mock(Context.**class**);

Intent intent = MainActivity.createQuery(context, "query", "value");

assertNotNull(intent);

Bundle extras = intent.getExtras();

assertNotNull(extras);

assertEquals("query", extras.getString("QUERY"));

assertEquals("value", extras.getString("VALUE"));

}

}