[Got it!](http://www.vogella.com/tutorials/Mockito/article.html#null)

We use cookies to personalise ads and to analyse our traffic. We also share information about your use of our site with our advertising and analytics partners. By using this site you agree to our cookie use. [See details](http://www.vogella.com/legal.html)

**Unit tests with Mockito - Tutorial**

Lars Vogel, Fabian Pfaff (c) 2012, 2016 vogella GmbHVersion 1.9,10.04.2017

Table of Contents

* [1. Prerequisites](http://www.vogella.com/tutorials/Mockito/article.html#prerequisites)
* [2. Testing with mock objects](http://www.vogella.com/tutorials/Mockito/article.html#testing-with-mock-objects)
* [3. Adding Mockito as dependencies to a project](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation)
* [4. Using the Mockito API](http://www.vogella.com/tutorials/Mockito/article.html#mockitousage)
* [5. Exercise: Write an instrumented unit test using Mockito](http://www.vogella.com/tutorials/Mockito/article.html#exercise_mockito_android)
* [6. Exercise: Creating mock objects using Mockito](http://www.vogella.com/tutorials/Mockito/article.html#exercise-creating-mock-objects-using-mockito)
* [7. Using PowerMock with Mockito](http://www.vogella.com/tutorials/Mockito/article.html#using-powermock-with-mockito)
* [8. Using a wrapper instead of Powermock](http://www.vogella.com/tutorials/Mockito/article.html#using-a-wrapper-instead-of-powermock)
* [9. About this website](http://www.vogella.com/tutorials/Mockito/article.html#about-this-website)
* [10. Mockito resources](http://www.vogella.com/tutorials/Mockito/article.html#resources_mockito)
* [Appendix A: Copyright and License](http://www.vogella.com/tutorials/Mockito/article.html#copyright-and-license)

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

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

The following tutorial is based on 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).

[**2. Testing with mock objects**](http://www.vogella.com/tutorials/Mockito/article.html#testing-with-mock-objects)

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

A unit test should test functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test, if possible.

This can be done via using test replacements (*test doubles*) for the real dependencies. Test doubles can be classified like the following:

* 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 don’t respond 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. Mock objects are configured to perform a certain behavior during a test. They typical record the interaction with the system and test can validate that.

Test doubles can be passed to other objects which are tested. Your tests can validate that the class reacts correctly during tests. For example, 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 typically require less code to configure and should therefore be preferred. |

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

You can create 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 an implementation to connect to the real data source is used. But for testing a mock object simulates the data source 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. You can, for example, validate that only certain methods have been called on the mock object.

[**2.3. Using Mockito for mocking objects**](http://www.vogella.com/tutorials/Mockito/article.html#using-mockito-for-mocking-objects)

*Mockito* 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.

If you use Mockito in tests you typically:

* Mock away external dependencies and insert the mocks into the code under test
* Execute the code under test
* Validate that the code executed correctly

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

[**3.1. Using Gradle for a Java project**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation_gradle)

If you use Gradle in a Java project, add the following dependency to the Gradle build file.

repositories { jcenter() }

dependencies { testCompile 'org.mockito:mockito-core:2.7.22' }

[**3.2. Using Gradle for an Android project**](http://www.vogella.com/tutorials/Mockito/article.html#using-gradle-for-an-android-project)

Add the following dependency to the Gradle build file:

dependencies {

*// ... more entries*

testCompile 'junit:junit:4.12'

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

testCompile 'org.mockito:mockito-core:2.7.22'

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

androidTestCompile 'org.mockito:mockito-android:2.7.22'

}

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

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

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

The Eclipse IDE supports the Gradle as well as the Maven build system. These build system allow to manage your software dependencies. Therefore, you are advised to use either the Gradle or Maven tooling in Eclipse.

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

If you are using IntelliJ, you should use either Gradle or Maven to manage your dependencies to Mockito.

[**3.6. OSGi or Eclipse plug-in development**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_installation_p2)

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 used in Eclipse based applications or plug-ins.

The Orbit repositories can be found here <http://download.eclipse.org/tools/orbit/downloads>

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

[**4.1. Creating mock objects with Mockito**](http://www.vogella.com/tutorials/Mockito/article.html#creating-mock-objects-with-mockito)

Mockito provides several methods to create mock objects:

* Using the static mock() method.
* Using the @Mock annotation.

If you use the @Mock annotation, you must trigger the creation of annotated objects. The MockitoRule allows this. It invokes the static method MockitoAnnotations.initMocks(this) to populate the annotated fields. Alternatively you can use @RunWith(MockitoJUnitRunner.class).

The usage of the @Mock annotation and the MockitoRule rule is demonstrated by the following example.

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

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

@Mock

MyDatabase databaseMock;

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

@Test

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

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

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

assertTrue(check);

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

}

}

|  |  |  |
| --- | --- | --- |
|  | Tells Mockito to mock the databaseMock instance | |
|  | Tells Mockito to create the mocks based on the @Mock annotation | |
|  | Instantiates the class under test using the created mock | |
|  | Executes some code of the class under test | |
|  | Asserts that the method call returned true | |
|  | Verify that the query method was called on the MyDatabase mock | |
|  | | *Static imports*  By adding the org.mockito.Mockito.\*; static import, you can use methods like mock() directly in your tests. Static imports allow you to call static members, i.e., methods and fields of a class directly without specifying the class.  Using static imports greatly improves the readability of your test code, you should use it. |

[**4.2. Configuring mocks**](http://www.vogella.com/tutorials/Mockito/article.html#configuring-mocks)

Mockito allows to configure the return values of its mocks via a fluent API. Unspecified method calls return "empty" values:

* null for objects
* 0 for numbers
* false for boolean
* empty collections for collections
* …​

|  |  |
| --- | --- |
|  | The following assert statements are only for demonstration purposes, a real test would use the mocks to unit test another functionality. |

[**4.2.1. "when thenReturn" and "when thenThrow**](http://www.vogella.com/tutorials/Mockito/article.html#when-thenreturn-and-when-thenthrow)

Mocks can return different values depending on arguments passed into a method. The when(…​.).thenReturn(…​.)method chain is used to specify a a return value for a method call with pre-defined parameters.

You also can use methods like anyString or anyInt to define that dependent on the input type a certain value should be returned.

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.

The following demonstrates the usage of when(…​.).thenReturn(…​.).

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

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

@Test

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

*// create mock*

MyClass test = 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<String> 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<String> 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<Integer> 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** testReturnValueInDependentOnMethodParameter2() {

Comparable<Todo> c= mock(Comparable.class);

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

*//assert*

assertEquals(0, c.compareTo(**new** Todo(1)));

}

The when(…​.).thenReturn(…​.) method chain can be used to throw an exception.

Properties properties = mock(Properties.class);

when(properties.get(”Anddroid”)).thenThrow(**new** IllegalArgumentException(...));

**try** {

properties.get(”Anddroid”);

fail(”Anddroid is misspelled”);

} **catch** (IllegalArgumentException ex) {

*// good!*

}

[**4.2.2. "doReturn when" and "doThrow when"**](http://www.vogella.com/tutorials/Mockito/article.html#doreturn-when-and-dothrow-when)

The doReturn(…​).when(…​).methodCall call chain works similar to when(…​.).thenReturn(…​.). It is useful for mocking methods which give an exception during a call, e.g., if you use use functionality like [Wrapping Java objects with Spy](http://www.vogella.com/tutorials/Mockito/article.html#mockito_spy).

doReturnWhen.java

The doThrow variant can be used for methods which return void to throw an exception. This usage is demonstrated by the following code snippet.

Properties properties = **new** Properties();

Properties spyProperties = spy(properties);

doReturn(“42”).when(spyProperties).get(”shoeSize”);

String value = spyProperties.get(”shoeSize”);

assertEquals(”42”, value);

[**4.3. Wrapping Java objects with Spy**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_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.\*;

@Test

**public** **void** testLinkedListSpyWrong() {

*// Lets mock a LinkedList*

List<String> list = **new** LinkedList<>();

List<String> spy = spy(list);

*// this does not work*

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

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

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

assertEquals("foo", spy.get(0));

}

@Test

**public** **void** testLinkedListSpyCorrect() {

*// Lets mock a LinkedList*

List<String> list = **new** LinkedList<>();

List<String> spy = spy(list);

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

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

assertEquals("foo", spy.get(0));

}

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

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. For example, you can verify that a method has been called with certain parameters. This kind of testing is sometimes called *behavior testing*. Behavior testing 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(ArgumentMatchers.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(test, never()).someMethod("never called");

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

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

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

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

*// This let's you check that no other methods where called on this object.*

*// You call it after you have verified the expected method calls.*

verifyNoMoreInteractions(test);

}

In case you do not care about the value, use the anyX, e.g., anyInt, anyString(), or any(YourClass.class)methods.

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

You also have the @InjectMocks annotation which tries to do constructor, method or field dependency injection based on the type. For example, assume that you have the following class.

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

**private** User user;

**private** ArticleDatabase database;

**public** ArticleManager(User user, ArticleDatabase database) {

super();

this.user = user;

this.database = database;

}

**public** **void** initialize() {

database.addListener(**new** ArticleListener());

}

}

This class can be constructed via Mockito and its dependencies can be fulfilled with mock objects as demonstrated by the following code snippet.

@RunWith(MockitoJUnitRunner.class)

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

@Mock ArticleCalculator calculator;

@Mock ArticleDatabase database;

@Mock User user;

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

@InjectMocks **private** ArticleManager manager;

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

*// calls addListener with an instance of ArticleListener*

manager.initialize();

*// validate that addListener was called*

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

}

}

|  |  |
| --- | --- |
|  | creates an instance of ArticleManager and injects the mocks into it |

Mockito can inject mocks either via constructor injection, setter injection, or property injection and in this order. So if ArticleManager would have a constructor that would only take User and setters for both fields, only the mock for User would be injected.

[**4.6. Capturing the arguments**](http://www.vogella.com/tutorials/Mockito/article.html#capturing-the-arguments)

The ArgumentCaptor class allows to access the arguments of method calls during the verification. This allows to capture these arguments of method calls and to use them for tests.

To run this example you need to add [hamcrest-library](https://mvnrepository.com/artifact/org.hamcrest/hamcrest-library) to your project.

**import** static org.hamcrest.Matchers.hasItem;

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

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

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

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Rule;

**import** org.junit.Test;

**import** org.mockito.ArgumentCaptor;

**import** org.mockito.Captor;

**import** org.mockito.junit.MockitoJUnit;

**import** org.mockito.junit.MockitoRule;

**public** **class** **MockitoTests** {

@Rule

**public** MockitoRule rule = MockitoJUnit.rule();

@Captor

**private** ArgumentCaptor<List<String>> captor;

@Test

**public** **final** **void** shouldContainCertainListItem() {

List<String> asList = Arrays.asList("someElement\_test", "someElement");

**final** List<String> mockedList = mock(List.class);

mockedList.addAll(asList);

verify(mockedList).addAll(captor.capture());

**final** List<String> capturedArgument = captor.getValue();

assertThat(capturedArgument, hasItem("someElement"));

}

}

[**4.7. Using Answers for complex mocks**](http://www.vogella.com/tutorials/Mockito/article.html#mockito_answers)

It is possible to define a Answer object for complex results. While thenReturn returns a predefined value every time, with answers you can calculate a response based on the arguments given to your stubbed method. This can be useful if your stubbed method is supposed to call a function on one of the arguments or if your method is supposed to return the first argument to allow method chaining. There exists a static method for the latter. Also note that there a different ways to configure an answer:

**import** static org.mockito.AdditionalAnswers.returnsFirstArg;

@Test

**public** **final** **void** answerTest() {

*// with doAnswer():*

doAnswer(returnsFirstArg()).when(list).add(anyString());

*// with thenAnswer():*

when(list.add(anyString())).thenAnswer(returnsFirstArg());

*// with then() alias:*

when(list.add(anyString())).then(returnsFirstArg());

}

Or if you need to do a callback on your argument:

@Test

**public** **final** **void** callbackTest() {

ApiService service = mock(ApiService.class);

when(service.login(any(Callback.class))).thenAnswer(i -> {

Callback callback = i.getArgument(0);

callback.notify("Success");

**return** null;

});

}

It is even possible to mock a persistence service like an DAO, but you should consider creating a fake class instead of a mock if your Answers become too complex.

List<User> userMap = **new** ArrayList<>();

UserDao dao = mock(UserDao.class);

when(dao.save(any(User.class))).thenAnswer(i -> {

User user = i.getArgument(0);

userMap.add(user.getId(), user);

**return** null;

});

when(dao.find(any(Integer.class))).thenAnswer(i -> {

**int** id = i.getArgument(0);

**return** userMap.get(id);

});

[**4.8. Mocking final classes**](http://www.vogella.com/tutorials/Mockito/article.html#mocking-final-classes)

Since Mockito v2 it is possible to mock final classes. This feature is incubating and is deactivated by default. To activate the mocking of final classes create the file org.mockito.plugins.MockMaker in either src/test/resources/mockito-extensions/ or src/mockito-extensions/. Add this line to the file: *mock-maker-inline*. With this modification we now can mock a final class.

**final** **class** **FinalClass** {

**public** **final** String finalMethod() { **return** "something"; }

}

@Test

**public** **final** **void** mockFinalClassTest() {

FinalClass instance = **new** FinalClass();

FinalClass mock = mock(FinalClass.class);

when(mock.finalMethod()).thenReturn("that other thing");

assertNotEquals(mock.finalMethod(), instance.finalMethod());

}

[**4.9. Clean test code with the help of the strict stubs rule**](http://www.vogella.com/tutorials/Mockito/article.html#clean-test-code-with-the-help-of-the-strict-stubs-rule)

The strict stubs rule helps you to keep your test code clean and checks for common oversights. It adds the following:

* test fails early when a stubbed method gets called with different arguments than what it was configured for (with PotentialStubbingProblem exception).
* test fails when a stubbed method isn’t called (with UnnecessaryStubbingException exception).
* org.mockito.Mockito.verifyNoMoreInteractions(Object) also verifies that all stubbed methods have been called during the test

@Test

**public** **void** withoutStrictStubsTest() **throws** Exception {

DeepThought deepThought = mock(DeepThought.class);

when(deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything")).thenReturn(42);

when(deepThought.otherMethod("some mundane thing")).thenReturn(null);

System.out.println(deepThought.getAnswerFor("Six by nine"));

assertEquals(42, deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything"));

verify(deepThought, times(1)).getAnswerFor("Ultimate Question of Life, The Universe, and Everything");

}

*// activate the strict subs rule*

@Rule **public** MockitoRule rule = MockitoJUnit.rule().strictness(Strictness.STRICT\_STUBS);

@Test

**public** **void** withStrictStubsTest() **throws** Exception {

DeepThought deepThought = mock(DeepThought.class);

when(deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything")).thenReturn(42);

*// this fails now with an UnnecessaryStubbingException since it is never called in the test*

when(deepThought.otherMethod("some mundane thing")).thenReturn(null);

*// this will now throw a PotentialStubbingProblem Exception since we usually don't want to call methods on mocks without configured behavior*

deepThought.someMethod();

assertEquals(42, deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything"));

*// verifyNoMoreInteractions now automatically verifies that all stubbed methods have been called as well*

verifyNoMoreInteractions(deepThought);

}

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

Mockito has certain limitations. For example, you cannot mock static methods and [private methods](https://github.com/mockito/mockito/wiki/Mockito-And-Private-Methods).

See [FAQ for Mockito limitations for the details](https://github.com/mockito/mockito/wiki/FAQ#what-are-the-limitations-of-mockito)

[**4.11. Behavior testing vrs. state testing**](http://www.vogella.com/tutorials/Mockito/article.html#behavior-testing-vrs-state-testing)

Mockito puts a focus on behavior testing, vrs. result testing. This is not always correct, for example, if you are testing a sort algorithm, you should test the result not the internal behavior.

*// state testing*

testSort() {

testList = [1, 7, 3, 8, 2]

MySorter.sort(testList)

**assert** testList equals [1, 2, 3, 7, 8]

}

*// incorrect would be behavior testing*

*// the following tests internal of the implementation*

testSort() {

testList = [1, 7, 3, 8, 2]

MySorter.sort(testList)

**assert** that compare(1, 2) was called once

**assert** that compare(1, 3) was not called

**assert** that compare(2, 3) was called once

....

}

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

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

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

**package** com.vogella.android.testing.mockito.contextmock;

**import** android.content.Context;

**import** android.content.Intent;

**public** **class** **Util** {

**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;

}

}

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

dependencies {

*// ... more entries*

testCompile 'junit:junit:4.12'

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

testCompile 'org.mockito:mockito-core:2.7.22'

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

androidTestCompile 'org.mockito:mockito-android:2.7.22'

}

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

Create a new unit test running on Android using Mockito in the androidTest folder. This test should check if the intent contains the correct extras. For this you mock the Context object with Mockito.

**package** com.vogella.android.testing.mockito.contextmock;

**import** android.content.Context;

**import** android.content.Intent;

**import** android.os.Bundle;

**import** android.support.test.runner.AndroidJUnit4;

**import** org.junit.Test;

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

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

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

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

@RunWith(AndroidJUnit4.class)

**public** **class** **UtilTest2** {

@Test

**public** **void** shouldContainTheCorrectExtras() **throws** Exception {

Context context = mock(Context.class);

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

assertNotNull(intent);

Bundle extras = intent.getExtras();

assertNotNull(extras);

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

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

}

}

[**6. Exercise: Creating mock objects using Mockito**](http://www.vogella.com/tutorials/Mockito/article.html#exercise-creating-mock-objects-using-mockito)

[**6.1. Target**](http://www.vogella.com/tutorials/Mockito/article.html#target)

Create an API, which can be mocked and use Mockito to do the job.

[**6.2. Create a sample Twitter API**](http://www.vogella.com/tutorials/Mockito/article.html#create-a-sample-twitter-api)

Implement a TwitterClient, which works with ITweet instances. But imagine these ITweet instances are pretty cumbersome to get, e.g., by using a complex service, which would have to be started.

**public** **interface** **ITweet** {

String getMessage();

}

**public** **class** **TwitterClient** {

**public** **void** sendTweet(ITweet tweet) {

String message = tweet.getMessage();

*// send the message to Twitter*

}

}

[**6.3. Mocking ITweet instances**](http://www.vogella.com/tutorials/Mockito/article.html#mocking-itweet-instances)

In order to avoid starting up a complex service to get ITweet instances, they can also be mocked by Mockito.

@Test

**public** **void** testSendingTweet() {

TwitterClient twitterClient = **new** TwitterClient();

ITweet iTweet = mock(ITweet.class);

when(iTweet.getMessage()).thenReturn("Using mockito is great");

twitterClient.sendTweet(iTweet);

}

Now the TwitterClient can make use of a mocked ITweet instance and will get "Using Mockito is great" as message when calling getMessage() on the mocked ITweet.

[**6.4. Verify method invocation**](http://www.vogella.com/tutorials/Mockito/article.html#verify-method-invocation)

Ensure that getMessage() is at least called once.

@Test

**public** **void** testSendingTweet() {

TwitterClient twitterClient = **new** TwitterClient();

ITweet iTweet = mock(ITweet.class);

when(iTweet.getMessage()).thenReturn("Using mockito is great");

twitterClient.sendTweet(iTweet);

verify(iTweet, atLeastOnce()).getMessage();

}

[**6.5. Validate**](http://www.vogella.com/tutorials/Mockito/article.html#validate)

Run the test and validate that it is successful.

[**7. Using PowerMock with Mockito**](http://www.vogella.com/tutorials/Mockito/article.html#using-powermock-with-mockito)

[**7.1. Powermock for mocking static methods**](http://www.vogella.com/tutorials/Mockito/article.html#powermock-for-mocking-static-methods)

Mockito cannot mock static methods. For this you can use Powermock. PowerMock provides a class called "PowerMockito" 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()).

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

**public** **final** **class** **NetworkReader** {

**public** **static** String getLocalHostname() {

String hostname = "";

**try** {

InetAddress addr = InetAddress.getLocalHost();

*// Get hostname*

hostname = addr.getHostName();

} **catch** ( UnknownHostException e ) {

}

**return** hostname;

}

}

To write a test which mocks away the NetworkReader as dependency you can use the following snippet.

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

**import** org.powermock.core.classloader.annotations.PrepareForTest;

@RunWith( PowerMockRunner.class )

@PrepareForTest( NetworkReader.class )

**public** **class** **MyTest** {

*// Find the tests here*

@Test

**public** **void** testSomething() {

mockStatic( NetworkUtil.class );

when( NetworkReader.getLocalHostname() ).andReturn( "localhost" );

*// now test the class which uses NetworkReader*

}

[**8. Using a wrapper instead of Powermock**](http://www.vogella.com/tutorials/Mockito/article.html#using-a-wrapper-instead-of-powermock)

Sometimes you can also use a wrapper around a static method, which can be mocked with Mockito.

**class** **FooWraper** {

**void** someMethod() {

Foo.someStaticMethod()

}

}

[**8.1. Learn more about Powermock**](http://www.vogella.com/tutorials/Mockito/article.html#learn-more-about-powermock)

See [Using PowerMock with Mockito](https://github.com/jayway/powermock/wiki/MockitoUsage) for more information

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

**[Support free content](http://www.vogella.com/support.html)**

**[Questions and discussion](http://www.vogella.com/contact.html)**

**[Tutorial & code license](http://www.vogella.com/license.html)**

**[Get the source code](http://www.vogella.com/code/index.html)**

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

[Mockito home page](http://site.mockito.org/)

[Dzone reference card](https://dzone.com/refcardz/mockito)

[Mockito project hosting page](https://github.com/mockito/mockito)

[Mockito release notes](https://github.com/mockito/mockito/blob/master/doc/release-notes/official.md)

[Martin Fowler about Mocks, Stubs etc.](http://martinfowler.com/articles/mocksArentStubs.html)

[Chiu-Ki Chan Advanced Android Espresso presentation](http://chiuki.github.io/advanced-android-espresso/)

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

| [**TRAINING**](http://www.vogella.com/training/) | [**SERVICE & SUPPORT**](http://www.vogella.com/consulting/) |
| --- | --- |
| The vogella company provides comprehensive [training and education services](http://www.vogella.com/training/) from experts in the areas of Eclipse RCP, Android, Git, Java, Gradle and Spring. We offer both public and inhouse training. Whichever course you decide to take, you are guaranteed to experience what many before you refer to as [“The best IT class I have ever attended”](http://www.vogella.com/training/). | The vogella company offers [expert consulting](http://www.vogella.com/consulting/) services, development support and coaching. Our customers range from Fortune 100 corporations to individual developers. |

[**Appendix A: Copyright and License**](http://www.vogella.com/tutorials/Mockito/article.html#copyright-and-license)

Copyright © 2012-2017 vogella GmbH. Free use of the software examples is granted under the terms of the EPL License. This tutorial is published under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Germany](http://creativecommons.org/licenses/by-nc-sa/3.0/de/deed.en)license.

See [Licence](http://www.vogella.com/license.html).

http://vg08.met.vgwort.de/na/27d90952a5254731bed50a1ef8bd6fb3

Version 1.9  
Last updated 2017-07-11 08:59:39 +02:00