**Unit Testing with JUnit – Tutorial**

Contents

[Unit Testing with JUnit – Tutorial 1](#_Toc535160206)

[1. The purpose of software tests 3](#_Toc535160207)

[1.1. What are software tests? 3](#_Toc535160208)

[1.2. Why are software tests helpful? 3](#_Toc535160209)

[2. Testing terminology 4](#_Toc535160210)

[2.1. Code (or application) under test 4](#_Toc535160211)

[2.2. Test fixture 4](#_Toc535160212)

[2.3. Unit tests and unit testing 4](#_Toc535160213)

[2.4. Integration tests 4](#_Toc535160214)

[2.5. Performance tests 4](#_Toc535160215)

[2.6. Behavior vs. state testing 5](#_Toc535160216)

[2.7. Testing frameworks for Java 5](#_Toc535160217)

[2.8. Where should the test be located? 5](#_Toc535160218)

[2.9. Which part of the software should be tested? 5](#_Toc535160219)

[3. Using JUnit 6](#_Toc535160220)

[3.1. The JUnit framework 6](#_Toc535160221)

[3.2. How to define a test in JUnit? 6](#_Toc535160222)

[3.3. Example JUnit test 6](#_Toc535160223)

[3.4. JUnit naming conventions 7](#_Toc535160224)

[3.5. JUnit naming conventions for Maven 7](#_Toc535160225)

[3.6. Run your test from the command line 7](#_Toc535160226)

[3.7. Test execution order 8](#_Toc535160227)

[4. Using JUnit 4 8](#_Toc535160228)

[4.1. Defining test methods 8](#_Toc535160229)

[4.2. Assert statements 9](#_Toc535160230)

[4.3. JUnit test suites 10](#_Toc535160231)

[4.4. Disabling tests 11](#_Toc535160232)

[4.5. Parameterized test 11](#_Toc535160233)

[4.6. JUnit Rules 14](#_Toc535160234)

[4.7. Writing custom JUnit rules 16](#_Toc535160235)

[4.8. Categories 17](#_Toc535160236)

[4.9 Junit static imports 18](#_Toc535160237)

[6. Installation of JUnit 19](#_Toc535160238)

[6.1. Using JUnit with Maven 19](#_Toc535160239)

[7. Setting Eclipse up for using JUnits static imports 19](#_Toc535160240)

[8. Overview of JUnit 5 20](#_Toc535160241)

[8.1. Usage of JUnit 5 with Maven 21](#_Toc535160242)

[8.2. Defining test methods 22](#_Toc535160243)

[8.3. Disabling tests 24](#_Toc535160244)

[8.4. Test Suites 24](#_Toc535160245)

[8.5. Expecting Exceptions 25](#_Toc535160246)

[8.6. Grouped assertions 25](#_Toc535160247)

[8.7. Timeout tests 25](#_Toc535160248)

[8.8. Running the same test repeatedly on a data set 26](#_Toc535160249)

[8.8.1. Using Dynamic Tests 26](#_Toc535160250)

[8.8.2. Using Parameterized Tests 27](#_Toc535160251)

[9. Comparison of annotations between JUnit 4 and 5 30](#_Toc535160252)

[10. Testing with mock objects 31](#_Toc535160253)

[10.1. Target and challenge of unit testing 31](#_Toc535160254)

[10.2. Mock object generation 32](#_Toc535160255)

[10.3. Using Mockito for mocking objects 32](#_Toc535160256)

[11. Adding Mockito as dependencies to a project 33](#_Toc535160257)

[11.1. Using Maven 33](#_Toc535160258)

[12. Using the Mockito API 33](#_Toc535160259)

[12.1. Creating mock objects with Mockito 33](#_Toc535160260)

[12.2. Configuring mocks 35](#_Toc535160261)

[12.2.0. Resetting Mock 35](#_Toc535160262)

[12.2.1. "when thenReturn" and "when thenThrow 36](#_Toc535160263)

[12.2.2. "doReturn when" and "doThrow when" 38](#_Toc535160264)

[12.3. Wrapping Java objects with Spy 39](#_Toc535160265)

[12.4. Verify the calls on the mock objects 40](#_Toc535160266)

[12.5 Using InOrder to verify the Order 41](#_Toc535160267)

[12.6. Using @InjectMocks for dependency injection via Mockito 42](#_Toc535160268)

[12.7. Capturing the arguments 43](#_Toc535160269)

[12.8. Using Answers for complex mocks 45](#_Toc535160270)

[12.9. Mocking final classes 46](#_Toc535160271)

[12.10. Clean test code with the help of the strict stubs rule 47](#_Toc535160272)

[12.11. Limitations 48](#_Toc535160273)

[12.12. Behavior testing vrs. state testing 49](#_Toc535160274)

[13. Using PowerMock with Mockito 49](#_Toc535160275)

[13.1. Powermock for mocking static methods 50](#_Toc535160276)

[14. Using a wrapper instead of Powermock 51](#_Toc535160277)

[14.1. Learn more about Powermock 51](#_Toc535160278)

*This tutorial explains unit testing with JUnit 4.x and JUnit5. It explains the creation of JUnit tests. It also covers the usage of the Eclipse IDE for developing software tests.*

[1. The purpose of software tests](http://www.vogella.com/tutorials/JUnit/article.html#testintroduction)

[1.1. What are software tests?](http://www.vogella.com/tutorials/JUnit/article.html#whataresoftwaretests)

A software test is a piece of software, which executes another piece of software. It validates if that code results in the expected state (state testing) or executes the expected sequence of events (behavior testing).

[1.2. Why are software tests helpful?](http://www.vogella.com/tutorials/JUnit/article.html#whytesting)

Software unit tests help the developer to verify that the logic of a piece of the program is correct.

Running tests automatically helps to identify software regressions introduced by changes in the source code. Having a high-test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

[2. Testing terminology](http://www.vogella.com/tutorials/JUnit/article.html#testingterminology)

[2.1. Code (or application) under test](http://www.vogella.com/tutorials/JUnit/article.html#code-or-application-under-test)

The code which is tested is typically called the *code under test*. If you are testing an application, this is called the *application under test*.

[2.2. Test fixture](http://www.vogella.com/tutorials/JUnit/article.html#junit_testfixture)

A *test fixture* is a fixed state in code which is tested used as input for a test. Another way to describe this is a test precondition.

For example, a test fixture might be a fixed string, which is used as input for a method. The test would validate if the method behaves correctly with this input.

[2.3. Unit tests and unit testing](http://www.vogella.com/tutorials/JUnit/article.html#junit_intro)

A *unit test* is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state.

The percentage of code which is tested by unit tests is typically called *test coverage*.

A unit test targets a small unit of code, e.g., a method or a class. External dependencies should be removed from unit tests, e.g., by replacing the dependency with a test implementation or a (mock) object created by a test framework.

Unit tests are not suitable for testing complex user interface or component interaction. For this, you should develop integration tests.

[2.4. Integration tests](http://www.vogella.com/tutorials/JUnit/article.html#junit_organization)

An *integration test* aims to test the behavior of a component or the integration between a set of components. The term *functional test* is sometimes used as synonym for integration test. Integration tests check that the entire system works as intended, therefore they are reducing the need for intensive manual tests.

These kinds of tests allow you to translate your user stories into a test suite. The test would resemble an expected user interaction with the application.

[2.5. Performance tests](http://www.vogella.com/tutorials/JUnit/article.html#junit_performancetests)

Performance tests are used to benchmark software components repeatedly. Their purpose is to ensure that the code under test runs fast enough even if it’s under high load.

[2.6. Behavior vs. state testing](http://www.vogella.com/tutorials/JUnit/article.html#behaviorvsstatetesting)

A test is a behavior test (also called interaction test) if it checks if certain methods were called with the correct input parameters. A behavior test does not validate the result of a method call.

State testing is about validating the result. Behavior testing is about testing the behavior of the application under test.

If you are testing algorithms or system functionality, in most cases you may want to test state and not interactions. A typical test setup uses mocks or stubs of related classes to abstract the interactions with these other classes away Afterwards you test the state or the behavior depending on your need.

[2.7. Testing frameworks for Java](http://www.vogella.com/tutorials/JUnit/article.html#testing-frameworks-for-java)

There are several testing frameworks available for Java. The most popular ones are JUnit and TestNG

This description focuses on JUnit. It covers both JUnit 4.x and JUnit 5.

[2.8. Where should the test be located?](http://www.vogella.com/tutorials/JUnit/article.html#junit_testorganization)

Typical, unit tests are created in a separate project or separate source folder to keep the test code separate from the real code. The standard convention from the Maven and Gradle build tools is to use:

* src/main/java - for Java classes
* src/test/java - for test classes

[2.9. Which part of the software should be tested?](http://www.vogella.com/tutorials/JUnit/article.html#junit_whattotest)

What should be tested is a highly controversial topic. Some developers believe every statement in your code should be tested.

In any case you should write software tests for the critical and complex parts of your application. If you introduce new features a solid test suite also protects you against regression in existing code.

In general, it safe to ignore trivial code. For example, it is typical useless to write tests for getter and setter methods which simply assign values to fields. Writing tests for these statements is time consuming and pointless, as you would be testing the Java virtual machine. The JVM itself already has test cases for this. If you are developing end user applications you are safe to assume that a field assignment works in Java.

If you start developing tests for an existing code base without any tests, it is good practice to start writing tests for code in which most of the errors happened in the past. This way you can focus on the critical parts of your application.

[3. Using JUnit](http://www.vogella.com/tutorials/JUnit/article.html#junittesting)

[3.1. The JUnit framework](http://www.vogella.com/tutorials/JUnit/article.html#unittesting_junit)

[JUnit](http://junit.org/) is a test framework which uses annotations to identify methods that specify a test. JUnit is an open source project hosted at [Github](https://github.com/junit-team/junit).

[3.2. How to define a test in JUnit?](http://www.vogella.com/tutorials/JUnit/article.html#unittesting_junit_test)

A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*. To define that a certain method is a test method, annotate it with the @Test annotation.

This method executes the code under test. You use an *assert* method, provided by JUnit or another assert framework, to check an expected result versus the actual result. These method calls are typically called *asserts* or *assert statements*.

You should provide meaningful messages in assert statements. That makes it easier for the user to identify and fix the problem. This is especially true if someone looks at the problem, who did not write the code under test or the test code.

[3.3. Example JUnit test](http://www.vogella.com/tutorials/JUnit/article.html#unittesting_junitexample)

The following code shows a JUnit test using the JUnit 5 version. This test assumes that the MyClass class exists and has a multiply(int, int) method.

**package** com.skills.test.basic;

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

**import** org.junit.Test;

**public** **class** **MyTests** {

@Test

**public** **void** multiplicationOfZeroIntegersShouldReturnZero() {

MyClass tester = **new** MyClass(); *// MyClass is tested*

*// assert statements*

assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");

assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");

assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");

}

}

[3.4. JUnit naming conventions](http://www.vogella.com/tutorials/JUnit/article.html#junit_namingconventions)

There are several potential naming conventions for JUnit tests. A widely-used solution for classes is to use the "Test" suffix at the end of test classes names.

As a general rule, a test name should explain what the test does. If that is done correctly, reading the actual implementation can be avoided.

One possible convention is to use the "should" in the test method name. For example, "ordersShouldBeCreated" or "menuShouldGetActive". This gives a hint what should happen if the test method is executed.

Another approach is to use "Given[ExplainYourInput]When[WhatIsDone]Then[ExpectedResult]" for the display name of the test method.

[3.5. JUnit naming conventions for Maven](http://www.vogella.com/tutorials/JUnit/article.html#junit_namingconventions_maven)

If you are using the Maven build system, you should use the "Test" suffix for test classes. The Maven build system (via its surfire plug-in) automatically includes such classes in its test scope.

[3.6. Run your test from the command line](http://www.vogella.com/tutorials/JUnit/article.html#juniteclipse_code)

You can also run your JUnit tests outside our IDE via standard Java code. Build systems like Apache Maven or Gradle in combination with a Continuous Integration Server (like Jenkins) can be used to execute tests automatically on a regular basis.

The org.junit.runner.JUnitCore class provides the runClasses() method. This method allows you to run one or several tests classes. As a return parameter you receive an object of the type org.junit.runner.Result. This object can be used to retrieve information about the tests.

The following class demonstrates how to run the MyClassTest. This class executes your test class and write potential failures to the console.

**package** com.skills.test.basic;

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** **MyTestRunner** {

**public** **static** **void** main(String**[]** args) {

Result result = JUnitCore.runClasses(MyClassTest.class);

**for** (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

}

}

This class can be executed like any other Java program on the command line. You only need to add the JUnit library JAR file to the classpath.

[3.7. Test execution order](http://www.vogella.com/tutorials/JUnit/article.html#usingjunit_executionorder)

JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests.

As of JUnit 4.11 the default is to use a deterministic, but not predictable, order for the execution of the tests.

You can use an annotation to define that the test methods are sorted by method name, in lexicographic order. To activate this feature, annotate your test class with the @FixMethodOrder(MethodSorters.NAME\_ASCENDING) annotation. You can also explicitely set the default by using the MethodSorters.DEFAULT parameter in this annotation. You can also use MethodSorters.JVM which uses the JVM defaults, which may vary from run to run.

[4. Using JUnit 4](http://www.vogella.com/tutorials/JUnit/article.html#usingjuni4)

[4.1. Defining test methods](http://www.vogella.com/tutorials/JUnit/article.html#usingjunit_annotations)

JUnit uses annotations to mark methods as test methods and to configure them. The following table gives an overview of the most important annotations in JUnit for the 4.x and 5.x versions. All these annotations can be used on methods.

| *Table 1. Annotations* | |
| --- | --- |
| **JUnit 4** | **Description** |
| import org.junit.\* | Import statement for using the following annotations. |
| @Test | Identifies a method as a test method. |
| @Before | Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class). |
| @After | Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures. |
| @BeforeClass | Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit. |
| @AfterClass | Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit. |
| @Ignore or @Ignore("Why disabled") | Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled. |
| @Test (expected = Exception.class) | Fails if the method does not throw the named exception. |
| @Test(timeout=100) | Fails if the method takes longer than 100 milliseconds. |

[4.2. Assert statements](http://www.vogella.com/tutorials/JUnit/article.html#usingjunit_asserts)

JUnit provides static methods to test for certain conditions via the Assert class. These *assert statements* typically start with assert. They allow you to specify the error message, the expected and the actual result. An *assertion method* compares the actual value returned by a test to the expected value. It throws an AssertionException if the comparison fails.

The following table gives an overview of these methods. Parameters in [] brackets are optional and of type String.

| *Table 2. Methods to assert test results* | |
| --- | --- |
| **Statement** | **Description** |
| fail([message]) | Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional. |
| assertTrue([message,] boolean condition) | Checks that the boolean condition is true. |
| assertFalse([message,] boolean condition) | Checks that the boolean condition is false. |
| assertEquals([message,] expected, actual) | Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays. |
| assertEquals([message,] expected, actual, tolerance) | Test that float or double values match. The tolerance is the number of decimals which must be the same. |
| assertNull([message,] object) | Checks that the object is null. |
| assertNotNull([message,] object) | Checks that the object is not null. |
| assertSame([message,] expected, actual) | Checks that both variables refer to the same object. |
| assertNotSame([message,] expected, actual) | Checks that both variables refer to different objects. |

[4.3. JUnit test suites](http://www.vogella.com/tutorials/JUnit/article.html#juniteclipse_testsuite)

If you have several test classes, you can combine them into a test suite. Running a test suite executes all test classes in that suite in the specified order. A test suite can also contain other test suites.

The following example code demonstrates the usage of a test suite. It contains two test classes (MyClassTest and MySecondClassTest). If you want to add another test class, you can add it to the @Suite.SuiteClasses statement.

**package** com.vogella.junit.first;

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

**import** org.junit.runners.Suite;

**import** org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)

@SuiteClasses({

MyClassTest.class,

MySecondClassTest.class })

**public** **class** **AllTests** {

}

[4.4. Disabling tests](http://www.vogella.com/tutorials/JUnit/article.html#disabling-tests)

The @Ignore annotation allow to statically ignore a test. Alternatively you can use Assume.assumeFalse or Assume.assumeTrue to define a condition for the test.Assume.assumeFalse marks the test as invalid, if its condition evaluates to true.Assume.assumeTrue evaluates the test as invalid if its condition evaluates to false. For example, the following disables a test on Linux:

Assume.assumeFalse(System.getProperty("os.name").contains("Linux"));

[4.5. Parameterized test](http://www.vogella.com/tutorials/JUnit/article.html#junitadvanced)

JUnit allows you to use parameters in a tests class. This class can contain **one** test method and this method is executed with the different parameters provided.

You mark a test class as a parameterized test with the @RunWith(Parameterized.class)annotation.

Such a test class must contain a static method annotated with the @Parameters annotation. That method generates and returns a collection of arrays. Each item in this collection is used as parameter for the test method.

You can use the @Parameter annotation on public fields to get the test values injected in the test.

The following code shows an example for a parameterized test. It tests the multiply() method of the MyClass class which is included as inner class for the purpose of this example.

**package** testing;

**import** org.junit.Test;

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

**import** org.junit.runners.Parameterized;

**import** org.junit.runners.Parameterized.Parameters;

**import** java.util.Arrays;

**import** java.util.Collection;

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

**import** static org.junit.runners.Parameterized.\*;

@RunWith(Parameterized.class)

**public** **class** **ParameterizedTestFields** {

*// fields used together with @Parameter must be public*

@Parameter(0)

**public** **int** m1;

@Parameter(1)

**public** **int** m2;

@Parameter(2)

**public** **int** result;

*// creates the test data*

@Parameters

**public** **static** Collection<Object**[]**> data() {

Object**[][]** data = **new** Object**[][]** { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };

**return** Arrays.asList(data);

}

@Test

**public** **void** testMultiplyException() {

MyClass tester = **new** MyClass();

assertEquals("Result", result, tester.multiply(m1, m2));

}

*// class to be tested*

**class** **MyClass** {

**public** **int** multiply(**int** i, **int** j) {

**return** i \*j;

}

}

}

Alternatively to using the @Parameter annotation you can use a constructor in which you store the values for each test. The number of elements in each array provided by the method annotated with @Parameters must correspond to the number of parameters in the constructor of the class. The class is created for each parameter and the test values are passed via the constructor to the class.

**package** de.vogella.junit.first;

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

**import** java.util.Arrays;

**import** java.util.Collection;

**import** org.junit.Test;

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

**import** org.junit.runners.Parameterized;

**import** org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)

**public** **class** **ParameterizedTestUsingConstructor** {

**private** **int** m1;

**private** **int** m2;

**public** ParameterizedTestUsingConstructor(**int** p1, **int** p2) {

m1 = p1;

m2 = p2;

}

*// creates the test data*

@Parameters

**public** **static** Collection<Object**[]**> data() {

Object**[][]** data = **new** Object**[][]** { { 1 , 2 }, { 5, 3 }, { 121, 4 } };

**return** Arrays.asList(data);

}

@Test

**public** **void** testMultiplyException() {

MyClass tester = **new** MyClass();

assertEquals("Result", m1 \* m2, tester.multiply(m1, m2));

}

*// class to be tested*

**class** **MyClass** {

**public** **int** multiply(**int** i, **int** j) {

**return** i \*j;

}

}

}

If you run this test class, the test method is executed with each defined parameter. In the above example the test method is executed three times.

A more flexible and easier to write approach is provided by the JUnitParams from <https://github.com/Pragmatists/JUnitParams>.

[4.6. JUnit Rules](http://www.vogella.com/tutorials/JUnit/article.html#junitadvanced_rules)

Via JUnit rules you can add behavior to each tests in a test class. You can annotate fields of type TestRule with the @Rule annotation. You can create objects which can be used and configured in your test methods. This adds more flexibility to your tests. You could, for example, specify which exception message you expect during the execution of your test code.

**package** de.vogella.junit.first;

**import** org.junit.Rule;

**import** org.junit.Test;

**import** org.junit.rules.ExpectedException;

**public** **class** **RuleExceptionTesterExample** {

@Rule

**public** ExpectedException exception = ExpectedException.none();

@Test

**public** **void** throwsIllegalArgumentExceptionIfIconIsNull() {

exception.expect(IllegalArgumentException.class);

exception.expectMessage("Negative value not allowed");

ClassToBeTested t = **new** ClassToBeTested();

t.methodToBeTest(-1);

}

}

JUnit already provides several useful rule implementations. For example, the TemporaryFolderclass allows to setup files and folders which are automatically removed after each test run.

The following code shows an example for the usage of the TemporaryFolder implementation.

**package** de.vogella.junit.first;

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

**import** java.io.File;

**import** java.io.IOException;

**import** org.junit.Rule;

**import** org.junit.Test;

**import** org.junit.rules.TemporaryFolder;

**public** **class** **RuleTester** {

@Rule

**public** TemporaryFolder folder = **new** TemporaryFolder();

@Test

**public** **void** testUsingTempFolder() **throws** IOException {

File createdFolder = folder.newFolder("newfolder");

File createdFile = folder.newFile("myfilefile.txt");

assertTrue(createdFile.exists());

}

}

**Rules provided by JUnit include:**

* **ErrorCollector**: collects multiple errors in a single test allowing it to continue to find all errors.
* **ExpectedException**: permits assertions to be made about exceptions thrown during a test.
* **ExternalResource**: sets up an external resource before a test with a guarantee to tear it down afterwards.
* **TemporaryFolder**: creates files and folders before a test and guarantees deletion afterwards.
* **TestName**: grants the tester access to the test name during the test method execution.
* **TestWatcher**: allows access to test events before and after the test method execution.
* **Timeout**: causes a test to fail after a set time.
* **Verifier**: permits a tester to add a final evaluation to a test, possibly failing an otherwise passing test.

For more examples of existing rules see <https://github.com/junit-team/junit4/wiki/Rules>.

[4.7. Writing custom JUnit rules](http://www.vogella.com/tutorials/JUnit/article.html#writing-custom-junit-rules)

To write your custom rule, you need to implement the TestRule interface. This interface defines the apply(Statement, Description) method which must return an instance of Statement. Statement represent the tests within the JUnit runtime and Statement#evaluate() run these. Description describes the individual test. It allows to read information about the test via reflection.

The following is a simple example for adding a log statement to an Android application before and after test execution.

**package** com.skills.test.rule;

**import** org.junit.rules.TestRule;

**import** org.junit.runner.Description;

**import** org.junit.runners.model.Statement;

**public** **class** **MyCustomRule** **implements** TestRule {

**private** Statement base;

**private** Description description;

**public** Statement apply(Statement base, Description description) {

this.base = base;

this.description = description;

**return** **new** MyStatement(base);

}

**public** **class** **MyStatement** **extends** Statement {

**private** **final** Statement base;

**public** MyStatement(Statement base) {

this.base = base;

}

@Override

**public** **void** evaluate() **throws** Throwable {

System.out.println("MyCustomRule",description.getMethodName() + "Started" );

**try** {

base.evaluate();

} **finally** {

System.out.println("MyCustomRule",description.getMethodName() + "Finished");

}

}

}

}

To use this rule, simple add a field annotated with @Rule to your test class.

@Rule

**public** MyCustomRule myRule = **new** MyCustomRule();

[4.8. Categories](http://www.vogella.com/tutorials/JUnit/article.html#junitadvanced_categories)

It is possible to define categories of tests and include or exclude them based on annotations. The following example is based on the [JUnit 4.8 release notes](https://github.com/junit-team/junit/blob/master/doc/ReleaseNotes4.8.md).

**public** **interface** **FastTests** { */\* category marker \*/*

}

**public** **interface** **SlowTests** { */\* category marker \*/*

}

**public** **class** **A** {

@Test

**public** **void** a() {

fail();

}

@Category(SlowTests.class)

@Test

**public** **void** b() {

}

}

@Category({ SlowTests.class, FastTests.class })

**public** **class** **B** {

@Test

**public** **void** c() {

}

}

@RunWith(Categories.class)

@IncludeCategory(SlowTests.class)

@SuiteClasses({ A.class, B.class })

*// Note that Categories is a kind of Suite*

**public** **class** **SlowTestSuite** {

*// Will run A.b and B.c, but not A.a*

}

@RunWith(Categories.class)

@IncludeCategory(SlowTests.class)

@ExcludeCategory(FastTests.class)

@SuiteClasses({ A.class, B.class })

*// Note that Categories is a kind of Suite*

**public** **class** **SlowTestSuite** {

*// Will run A.b, but not A.a or B.c*

}

4.9 Junit static imports

Static import is a feature that allows fields and methods defined in a class as public static to be used without specifying the class in which the field is defined.

JUnit assert statements are typically defined as public static to allow the developer to write short test statements. The following snippet demonstrates an assert statement with and without static imports.

*// without static imports you have to write the following statement*

Assert.assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));

*// alternatively define assertEquals as static import*

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

*// more code*

*// use assertEquals directly because of the static import*

assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));

[6. Installation of JUnit](http://www.vogella.com/tutorials/JUnit/article.html#installation-of-junit)

[6.1. Using JUnit with Maven](http://www.vogella.com/tutorials/JUnit/article.html#using-junit-with-maven)

To use JUnit in your Maven build, add the following dependency to your pom file.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

[7. Setting Eclipse up for using JUnits static imports](http://www.vogella.com/tutorials/JUnit/article.html#setting-eclipse-up-for-using-junits-static-imports)

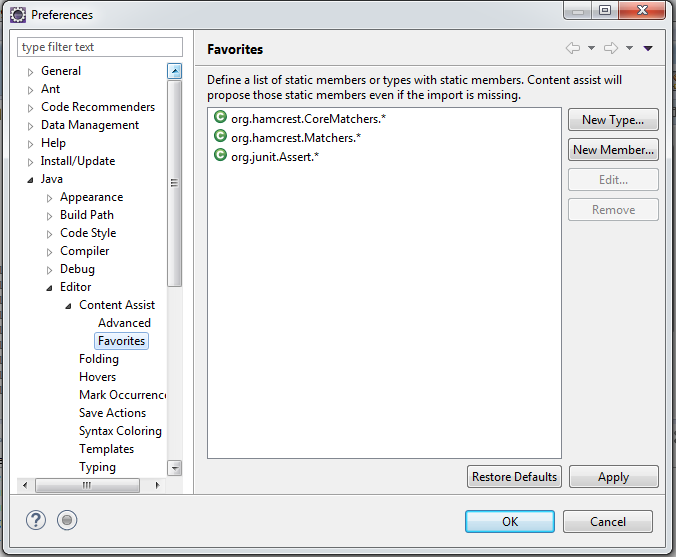
The Eclipse IDE cannot always create the corresponding static import statements automatically.

You can configure the Eclipse IDE to use code completion to insert typical JUnit method calls and to add the static import automatically. For this open the Preferences via Window ▸ Preferencesand select Java ▸ Editor ▸ Content Assist ▸ Favorites.

Use the New Type button to add the following entries to it:

* org.junit.Assert
* org.hamcrest.CoreMatchers
* org.hamcrest.Matchers

This makes, for example, the assertTrue, assertFalse and assertEquals methods directly available in the Content Assists.



You can now use Content Assists (shortcut: Ctrl+Space) to add the method and the import

[8. Overview of JUnit 5](http://www.vogella.com/tutorials/JUnit/article.html#junit5)

JUnit 5 is the latest major release of JUnit.

JUnit 5 consists of a number of discrete components:

* JUnit Platform - foundation layer which enables different testing frameworks to be launched on the JVM
* Junit Jupiter - is the JUnit 5 test framework which is launched by JUnit Platform
* JUnit Vintage - legacy TestEngine which runs older tests

[8.1. Usage of JUnit 5 with Maven](http://www.vogella.com/tutorials/JUnit/article.html#usage-of-junit-5-with-maven)

This example shows how to import all components of JUnit 5 into your project.

We need to register the individual components with Maven surefire:

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>${java.version}</source>

<target>${java.version}</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.19.1</version>

<configuration>

<includes>

<include>\*\*/Test\*.java</include>

<include>\*\*/\*Test.java</include>

<include>\*\*/\*Tests.java</include>

<include>\*\*/\*TestCase.java</include>

</includes>

<properties>

*<!-- <includeTags>fast</includeTags> -->*

<excludeTags>slow</excludeTags>

</properties>

</configuration>

<dependencies>

<dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-surefire-provider</artifactId>

<version>${junit.platform.version}</version>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>${junit.jupiter.version}</version>

</dependency>

<dependency>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

<version>${junit.vintage.version}</version>

</dependency>

</dependencies>

</plugin>

</plugins>

</build>

And add the dependencies:

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>${junit.jupiter.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

</dependencies>

You can find a complete example of a working maven configuration here: <https://github.com/junit-team/junit5-samples/blob/r5.0.0-M4/junit5-maven-consumer/pom.xml>

|  |  |
| --- | --- |
|  | The above works for Java projects but not yet for Android projects. |

[8.2. Defining test methods](http://www.vogella.com/tutorials/JUnit/article.html#usingjunit_annotations)

JUnit uses annotations to mark methods as test methods and to configure them. The following table gives an overview of the most important annotations in JUnit for the 4.x and 5.x versions. All these annotations can be used on methods.

| *Table 3. Annotations* | |
| --- | --- |
| import org.junit.jupiter.api.\* | **Import statement for using the following annotations.** |
| @Test | Identifies a method as a test method. |
| @RepeatedTest(<Number>) | Repeats the test a <Number> of times |
| @TestFactory | Method is a Factory for dynamic tests |
| @BeforeEach | Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class). |
| @AfterEach | Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures. |
| @BeforeAll | Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit. |
| @AfterAll | Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit. |
| @Nested | Lets you nest inner test classes to force a certain execution order |
| @Tag("<TagName>") | Tests in JUnit 5 can be filtered by tag. Eg., run only tests tagged with "fast". |
| @ExtendWith | Lets you register an Extension class that integrates with one or more extension points |
| @Disabled or @Disabled("Why disabled") | Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled. |
| @DisplayName("<Name>") | <Name> that will be displayed by the test runner. In contrast to method names the DisplayName can contain spaces. |

[8.3. Disabling tests](http://www.vogella.com/tutorials/JUnit/article.html#usingjunit_executionorder)

The @Disabled annotation allows to statically ignore a test.

Alternatively you can use Assumptions.assumeFalse or Assumptions.assumeTrue to define a condition for test deactivation. Assumptions.assumeFalse marks the test as invalid, if its condition evaluates to true. Assumptions.assumeTrue evaluates the test as invalid if its vcondition evaluates to false. For example, the following disables a test on Linux:

Assumptions.assumeFalse(System.getProperty("os.name").contains("Linux"));

[8.4. Test Suites](http://www.vogella.com/tutorials/JUnit/article.html#test-suites)

To run multiple tests together, you can use test suites. They allow to aggregate multiple test classes. JUnit 5 provides two annotations:

* @SelectPackages - used to specify the names of packages for the test suite
* @SelectClasses - used to specify the classes for the test suite. They can be located in different packages.

@RunWith(JUnitPlatform.class)

@SelectPackages("com.vogella.junit5.examples")

**public** **class** **AllTests** {}

@RunWith(JUnitPlatform.class)

@SelectClasses({AssertionTest.class, AssumptionTest.class, ExceptionTest.class})

**public** **class** **AllTests** {}

[8.5. Expecting Exceptions](http://www.vogella.com/tutorials/JUnit/article.html#expecting-exceptions)

Exception is handling with org.junit.jupiter.api.Assertions.expectThrows(). You define the expected Exception class and provide code that should throw the exception.

**import** static org.junit.jupiter.api.Assertions.expectThrows;

@Test

**void** exceptionTesting() {

*// set up user*

Throwable exception = expectThrows(IllegalArgumentException.class, () -> user.setAge("23"));

assertEquals("Age must be an Integer.", exception.getMessage());

}

This lets you define which part of the test should throw the exception. The test will still fail if an exception is thrown outside of this scope.

[8.6. Grouped assertions](http://www.vogella.com/tutorials/JUnit/article.html#grouped-assertions)

@Test

**void** groupedAssertions() {

Address address = **new** Address();

*// In a grouped assertion all assertions are executed, even after a failure.*

*// The error messages get grouped together.*

assertAll("address name",

() -> assertEquals("John", address.getFirstName()),

() -> assertEquals("User", address.getLastName())

);

}

=> org.opentest4j.MultipleFailuresError: address name (2 failures)

expected: <John> but was: <null>

expected: <User> but was: <null>

[8.7. Timeout tests](http://www.vogella.com/tutorials/JUnit/article.html#timeout-tests)

If you want to ensure that a test fails if it isn’t done in a certain amount of time you can use the assertTimeout() method. This method will wait until

**import** static org.junit.jupiter.api.Assertions.assertTimeout;

**import** static java.time.Duration.ofSeconds;

**import** static java.time.Duration.ofMinutes;

@Test

**void** timeoutNotExceeded() {

assertTimeout(ofMinutes(1), () -> service.doBackup());

}

*// if you have to check a return value*

@Test

**void** timeoutNotExceededWithResult() {

String actualResult = assertTimeout(ofSeconds(1), () -> {

**return** restService.request(request);

});

assertEquals(200, request.getStatus());

}

=> org.opentest4j.AssertionFailedError: execution exceeded timeout of 1000 ms by 212 ms

If you want your tests to cancel after the timeout period is passed you can use the assertTimeoutPreemptively() method.

@Test

**void** timeoutNotExceededWithResult() {

String actualResult = assertTimeoutPreemptively(ofSeconds(1), () -> {

**return** restService.request(request);

});

assertEquals(200, request.getStatus());

}

=> org.opentest4j.AssertionFailedError: execution timed out after 1000 ms

[8.8. Running the same test repeatedly on a data set](http://www.vogella.com/tutorials/JUnit/article.html#running-the-same-test-repeatedly-on-a-data-set)

Sometimes we want to be able to run the same test on a data set. Holding the data set in a Collection and iterating over it with the assertion in the loop body has the problem that the first assertion failure will stop the test execution. JUnit 5 offers multiple ways to overcome this limitation.

[8.8.1. Using Dynamic Tests](http://www.vogella.com/tutorials/JUnit/article.html#dynamic_tests)

JUnit 5 offers the possibility to define dynamic tests. We can use this to rewrite our example. Dynamic test methods are annotated with @TestFactory and can return an Iterable, a Collection or a Stream of DynamicTests. JUnit then runs every DynamicTest when the test is executed. @BeforeEach and @AfterEach methods will not be called for dynamic tests. This means that you can’t use them to reset the test object if you change it’s state in the lambda expression for a dynamic test.

In this example we return a Stream:

**import** static org.junit.jupiter.api.Assertions.\*;

**import** static org.junit.jupiter.api.DynamicTest.dynamicTest;

**import** java.util.Arrays;

**import** java.util.stream.Stream;

**import** org.junit.jupiter.api.DynamicTest;

**import** org.junit.jupiter.api.TestFactory;

**public** **class** **DynamicTestCreationTest** {

@TestFactory

**public** Stream<DynamicTest> testMultiplyException() {

MyClass tester = **new** MyClass();

**int[][]** data = **new** **int[][]** { { 1, 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };

**return** Arrays.stream(data).map(entry -> {

**int** m1 = entry[0];

**int** m2 = entry[1];

**int** expected = entry[2];

**return** dynamicTest(m1 + " \* " + m2 + " = " + expected, () -> {

assertEquals(expected, tester.multiply(m1, m2));

});

});

}

*// class to be tested*

**class** **MyClass** {

**public** **int** multiply(**int** i, **int** j) {

**return** i \* j;

}

}

}

[8.8.2. Using Parameterized Tests](http://www.vogella.com/tutorials/JUnit/article.html#using-parameterized-tests)

Junit5 also supports parameterized tests. To use them you have to add the junit-jupiter-params package as a test dependencies. If you are using gradle:

dependencies {

*// ..*

testCompile group: 'org.junit.jupiter', name: 'junit-jupiter-params', version: '5.0.0'

}

For this example we use the @MethodSource annotation. We give it the name of the function(s) we want it to call to get it’s test data. The function has to be static and must return either a Collection, an Iterator, a Stream or an Array. On execution the test method gets called once for every entry in the data source. In contrast to [Dynamic Tests](http://www.vogella.com/tutorials/JUnit/article.html#dynamic_tests) @BeforeEach and @AfterEachmethods will be called for parameterized tests.

**import** static org.junit.jupiter.api.Assertions.\*;

**import** org.junit.jupiter.params.ParameterizedTest;

**import** org.junit.jupiter.params.provider.MethodSource;

**public** **class** **DynamicTestCreationTest** {

**public** **static** **int[][]** data() {

**return** **new** **int[][]** { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };

}

@ParameterizedTest

@MethodSource(names = "data")

**void** testWithStringParameter(**int[]** data) {

MyClass tester = **new** MyClass();

**int** m1 = data[0];

**int** m2 = data[1];

**int** expected = data[2];

assertEquals(expected, tester.multiply(m1, m2));

}

*// class to be tested*

**class** **MyClass** {

**public** **int** multiply(**int** i, **int** j) {

**return** i \* j;

}

}

}

##### [Data sources](http://www.vogella.com/tutorials/JUnit/article.html#data-sources)

The following table gives an overview of all possible test data sources for parameterized tests.

| *Table 4. Table Parameterized Tests Data Sources* | |
| --- | --- |
| **Annotation** | **Description** |
| @ValueSource(ints = { 1, 2, 3 }) | Lets you define an array of test values. Permissible types are String, int, long, or double. |
| @EnumSource(value = Months.class, names = {"JANUARY", "FEBRUARY"}) | Lets you pass Enum constants as test class. With the optional attribute names you can choose which constants should be used. Otherwise all attributes are used. |
| @MethodSource(names = "genTestData") | The result of the named method is passed as argument to the test. |
| @CsvSource({ "foo, 1", "'baz, qux', 3" })  **void** testMethod(String first, **int** second) { | Expects strings to be parsed as Csv. The delimiter is ','. |
| @ArgumentsSource(MyArgumentsProvider.class) | Specifies a class that provides the test data. The referenced class has to implement the ArgumentsProvider interface. |

##### [Argument conversion](http://www.vogella.com/tutorials/JUnit/article.html#argument-conversion)

JUnit tries to automatically convert the source strings to match the expected arguments of the test method.

If you need explicit conversion you can specify a converter with the @ConvertWith annotation. To define your own converter you have to implement the ArgumentConverter interface. In the following example we use the abstract SimpleArgumentConverter base class.

@ParameterizedTest

@ValueSource(ints = {1, 12, 42})

**void** testWithExplicitArgumentConversion(@ConvertWith(ToOctalStringArgumentConverter.class) String argument) {

System.err.println(argument);

assertNotNull(argument);

}

**static** **class** **ToOctalStringArgumentConverter** **extends** SimpleArgumentConverter {

@Override

**protected** Object convert(Object source, Class<?> targetType) {

assertEquals(Integer.class, source.getClass(), "Can only convert from Integers.");

assertEquals(String.class, targetType, "Can only convert to String");

**return** Integer.toOctalString((Integer) source);

}

}

[9. Comparison of annotations between JUnit 4 and 5](http://www.vogella.com/tutorials/JUnit/article.html#usingjunit_annotations)

|  |  |  |
| --- | --- | --- |
| *Table 5. Annotations* | | |
| **JUnit 5** | **JUnit 4** | **Description** |
| import org.junit.jupiter.api.\* | import org.junit.\* | Import statement for using the following annotations. |
| @Test | @Test | Identifies a method as a test method. |
| @BeforeEach | @Before | Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class). |
| @AfterEach | @After | Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures. |
| @BeforeAll | @BeforeClass | Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as  static   to work with JUnit. |
| @AfterAll | @AfterClass | Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with Junit. |
| @Disabled  or  @Disabled("Why disabled") | @Ignore   or  @Ignore("Why disabled") | Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled. |
| Not available, is replaced by  org.junit.jupiter.api.Assertions.expectThrows() | @Test (expected = Exception.class) | Fails if the method does not throw the named exception. |
| Not available, is replaced by  AssertTimeout.assertTimeout()   and  AssertTimeout.assertTimeoutPreemptively() | @Test(timeout=100) | Fails if the method takes longer than 100 milliseconds. |

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

[10.1. 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 typically record the interaction with the system and tests 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. |

[10.2. 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.

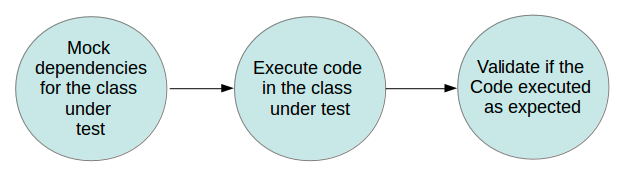
Mocking or mock frameworks allows testing the expected interaction with the mock object. You can, for example, validate that only certain methods have been called on the mock object.

[10.3. 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



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

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

Using the below Maven dependency, we can able to add Mockito to our project.

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>2.21.0</version>

<scope>test</scope>

</dependency>

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

[12.1. 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. |

[12.2. 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. |

12.2.0. Resetting Mock

Mockito provides the capability to a reset a mock so that it can be reused later. Take a look at the following code snippet.

//reset mock

reset(calcService);

Please see the below example, it will reset the whole mock object including stubbed methods. When we are calling again the ‘*arithmatic.add(1,1)*‘ will return 0. Because, it has been reset by *‘reset(arithmatic)’*.

//Resetting the mock

@Test

**public** **void** testResetMockito(){

Arithmatic arithmatic=*mock*(Arithmatic.**class**);

*when*(arithmatic.add(*anyInt*(), *anyInt*())).thenReturn(2);

System.***out***.println("Before Reset : "+arithmatic.add(1, 1));

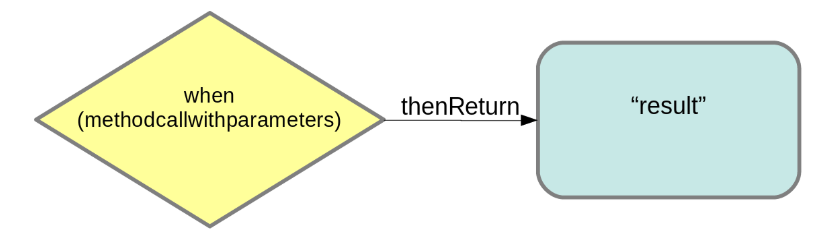
*reset*(arithmatic);

System.***out***.println("After Reset : "+arithmatic.add(1, 1));

}

[12.2.1. "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.

There are two Matchers are available to provide methods like anyString().

1. ArgumentMatchers

This will provide basic matcher functions like,

1. anyString()
2. anyInt()
3. anyByte()
4. anyList()
5. anySet()
6. eq()
7. isNull()
8. and so on..
9. AdditionalMatchers

This will provide additional and useful matchers like,

1. gt()
2. lt()
3. geq()
4. leq()
5. aryEq()
6. and()
7. and so on..

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!*

}

[12.2.2. "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);

[12.3. 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));

}

[12.4. 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.

**Other Verifier Options:**

1. verifyNoMoreInteractions – All methods from mock object has been verified or not.
2. verifyZeroInteractions – To check no methods or interactions happened on mock objects
3. VerficationCollector – It is a collector to collect and report the errors. We can use assertLazily() to perform particular test method lazily and report it separately.

12.5 Using InOrder to verify the Order

Mockito provides Inorder class which takes care of the order of method calls that the mock is going to make in due course of its action. With InOrder class, we can verify the order of method interactions happened with Mock object.

**package** com.skills.test.mockito;

**import** **static** org.mockito.ArgumentMatchers.*anyInt*;

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

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

**import** org.junit.Before;

**import** org.junit.Test;

**import** org.mockito.InOrder;

**import** org.mockito.Mock;

**import** org.mockito.Mockito;

**import** com.skills.junit.mockito.Arithmatic;

**public** **class** InOrderTest {

Arithmatic arithmatic;

@Before

**public** **void** before(){

arithmatic=*mock*(Arithmatic.**class**);

}

@Test

**public** **void** testArithmatic(){

*when*(arithmatic.add(*anyInt*(), *anyInt*())).thenCallRealMethod();

*when*(arithmatic.multiply(*anyInt*(), *anyInt*())).thenCallRealMethod();

System.***out***.println(arithmatic.add(1, 2));

System.***out***.println(arithmatic.multiply(1, 4));

InOrder inOrder=*inOrder*(arithmatic);

inOrder.verify(arithmatic).add(*anyInt*(), *anyInt*());

inOrder.verify(arithmatic).multiply(*anyInt*(), *anyInt*());

}

}

[12.6. 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 Userand setters for both fields, only the mock for User would be injected.

Points to Remember:

1. @InjectMocks is injecting mocks, it won’t generate mock object. So, It will call its real method.
2. It will auto inject other dependency, which is marked as @Mock or @Spy
3. Three types of Injection
4. Constructor – If same type is available two or more, same mock object will be assigned
5. Getter/Setter - If same type is available two or more, then based on name it will be inject. It will inject single object, other object will be null.
6. Variable – Same as Getter/Setter
7. Instead of @Mock

[12.7. 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"));

}

}

**Possible of assertThat**

import static org.junit.Assert.\*;

import static org.hamcrest.CoreMatchers.\*;

import org.junit.Before;

import org.junit.Test;

public class JUnitTestAssertThatAssertions {

int totalNumberOfApplicants = 0;

@Before

public void setData(){

this.totalNumberOfApplicants = 9;

}

@Test

public void testAssertThatEqual() {

assertThat("123",is("123"));

}

@Test

public void testAssertThatNotEqual() {

assertThat(totalNumberOfApplicants,is(123));

}

@Test

public void testAssertThatObject() {

assertThat("123",isA(String.class));

}

@Test

public void testAssertThatWMessage(){

assertThat("They are not equal!","123",is("1234"));

}

}

[12.8. 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);

});

[12.9. 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());

}

[12.10. 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 UnnecessaryStubbingExceptionexception).
* 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);

}

We can enable STRICT\_STUBS in three ways:

1. Rule

@Rule

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

1. Annotation

@RunWith(StrictStubs.**class**)

1. Mockito Session

@Before **public** **void** setup() {

//initialize session to start mocking

mockito = Mockito.mockitoSession()

.initMocks(**this**)

.strictness(Strictness.***STRICT\_STUBS***)

.startMocking();

}

[12.11. 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 some of the limitations

* Requires Java 6+
* Cannot mock static methods
* Cannot mock constructors
* Cannot mock equals(), hashCode(). Firstly, you should not mock those methods. Secondly, Mockito defines and depends upon a specific implementation of these methods. Redefining them might break Mockito.
* Mocking is only possible on VMs that are [supported by Objenesis](https://github.com/easymock/objenesis/blob/2.5/SupportedJVMs.md). Don't worry, most VMs should work just fine.
* Spying on real methods where real implementation references outer Class via OuterClass.this is impossible. Don't worry, this is extremely rare case.

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

[12.12. 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

....

}

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

[13.1. 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*

}

[14. 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()

}

}

[14.1. 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