http://www.vogella.com/articles/JUnit/article.html

**JUnit - Tutorial**

**Lars Vogel**

Version 2.1

Copyright © 2007, 2008, 2009, 2010, 2011, 2012 Lars Vogel

02.04.2012

|  |  |  |
| --- | --- | --- |
| **Revision History** | | |
| Revision 0.1-0.5 | 03.09.2007 | Lars Vogel |
| JUnit description | | |
| Revision 0.6 - 2.1 | 10.05.2008 - 02.04.2012 | Lars Vogel |
| bugfixes and enhancements | | |

**Unit testing with JUnit**

This tutorial explains unit testing with JUnit 4.x. It explains the creation of JUnit tests and how to run them in Eclipse or via own code.

**Table of Contents**

[**1. Introduction**](http://www.vogella.com/articles/JUnit/article.html#unittesting)

[**1.1. Unit Testing**](http://www.vogella.com/articles/JUnit/article.html#junit_intro)

[**1.2. Unit Testing with JUnit**](http://www.vogella.com/articles/JUnit/article.html#unittesting_junit)

[**2. Installation of JUnit**](http://www.vogella.com/articles/JUnit/article.html#installation)

[**3. Using JUnit**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse)

[**3.1. Preparation**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse_prep)

[**3.2. Create a Java class**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse_class)

[**3.3. Create a JUnit test**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse_test)

[**3.4. Run your test via Eclipse**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse_eclipse)

[**3.5. Run your test via code**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse_code)

[**4. JUnit (more) in Detail**](http://www.vogella.com/articles/JUnit/article.html#usingjunit)

[**4.1. Static imports with Eclipse**](http://www.vogella.com/articles/JUnit/article.html#usingjunit_staticimports)

[**4.2. Annotations**](http://www.vogella.com/articles/JUnit/article.html#usingjunit_annotations)

[**4.3. Assert statements**](http://www.vogella.com/articles/JUnit/article.html#usingjunit_asserts)

[**5. Mocking with EasyMock**](http://www.vogella.com/articles/JUnit/article.html#easymock)

[**6. Thank you**](http://www.vogella.com/articles/JUnit/article.html#thankyou)

[**7. Questions and Discussion**](http://www.vogella.com/articles/JUnit/article.html#questions)

[**8. Links and Literature**](http://www.vogella.com/articles/JUnit/article.html#resources)

[**8.1. JUnit Resources**](http://www.vogella.com/articles/JUnit/article.html#junit_links)

[**8.2. vogella Resources**](http://www.vogella.com/articles/JUnit/article.html#resources_general)

This tutorial is part of this Kindle book:

**[](http://www.amazon.com/dp/B006O1J39S)**

**1. Introduction**

**1.1. Unit Testing**

A unit test is a piece of code written by a developer that executes a specific functionality in the code under test. Unit tests ensure that code is working as intended and validate that this is still the case after code changes.

**1.2. Unit Testing with JUnit**

JUnit 4.x is a test framework which uses annotations to identify methods that are tests. JUnit assumes that all test methods can be executed in an arbitrary order. Therefore tests should not depend on other tests.

To write a test with JUnit

* Annotate a method with @org.junit.Test
* Use a method provided by JUnit to check the expected result of the code execution versus the actual result

You can use Eclipse or the class "org.junit.runner.JUnitCore" to run the test.

**2. Installation of JUnit**

If you use Eclipse you can use the integrated JUnit in Eclipse for your testing.

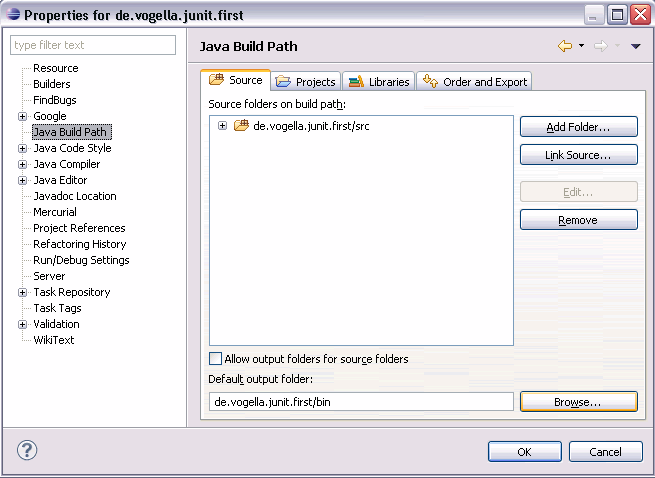
If you want to control the used JUnit library explicitly, download JUnit4.x.jar from the JUnit website at [**http://www.junit.org/**](http://www.junit.org/) . The download contains the "junit-4.\*.jar" which is the JUnit library. Add this library to your Java project and add it to the classpath.

**3. Using JUnit**

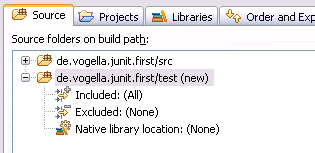
**3.1. Preparation**

Create a new project de.vogella.junit.first. We want to create the unit tests in a separate folder. The creation of a separate folder for tests is not mandatory. But it is a good practice to keep the code separated from the regular code. You might even create a separate project for the test classes, but we skip this step to make this example simpler.

Create a new source folder test via right-clicking on your project, select "Properties" and choose the "Java Build Path". Select the "Source" tab.



Press "Add folder" then press "Create new folder". Create the folder "test".



Alternatively you can add a new source folder by right-clicking on a project and selecting New → Source Folder.

**3.2. Create a Java class**

In the "src" folder, create the de.vogella.junit.first package and the following class.

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

**public** **class** MyClass {

**public** **int** multiply(**int** x, **int** y) {

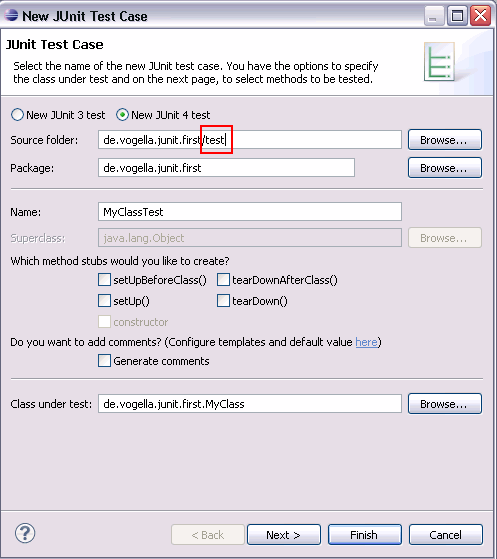
**return** x / y;

}

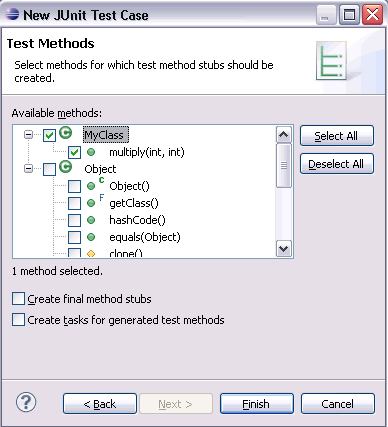
}

**3.3. Create a JUnit test**

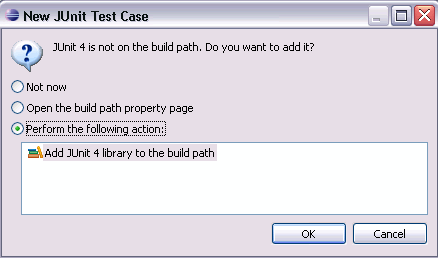
Right click on your new class in the Package Explorer and select New → JUnit Test Case. Select "New JUnit 4 test" and set the source folder to "test", so that your test class gets created in this folder.



Press "Next" and select the methods which you want to test.



If the JUnit library in not part of your classpath, Eclipse will prompt you to do so.



Create a test with the following code.

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

**import** org.junit.Test;

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

**public** **class** MyClassTest {

*@Test*

**public** **void** testMultiply() {

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

assertEquals("Result", 50, tester.multiply(10, 5));

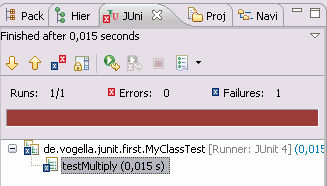
}

}

**3.4. Run your test via Eclipse**

Right click on your new test class and select Run-As → JUnit Test.

The result of the tests will be displayed in the JUnit View.

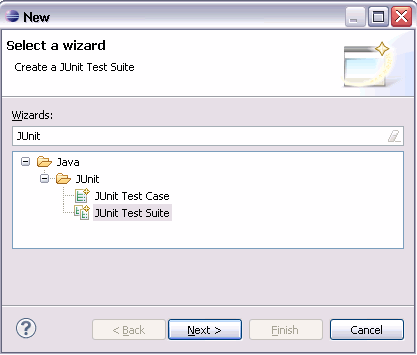


The test should be failing (indicated via a red bar).

This is because our multiplier class is currently not working correctly (it does a division instead of multiplication). Fix the bug and re-run test to get a green bar.

If you have several tests you can combine them into a test suite. Running a test suite will execute all tests in that suite.

To create a test suite, select your test classes → right click on it → New → Other → JUnit → Test Suite.



Select "Next" and select the methods for which you want to create a test.

Change the code to the following to make your test suite run your test. If you develop another test later you can add it to @Suite.SuiteClasses.

**package** mypackage;

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

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

*@RunWith(Suite.class)*

*@Suite.SuiteClasses( { MyClassTest.class })*

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

}

**3.5. Run your test via code**

You can also run your tests from via your own code. The class org.junit.runner.JUnitCore provides the method runClasses() which 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.

In your "test" folder create a new class MyTestRunner with the following code. This class will execute your test class and write potential failures to the console.

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

**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());

}

}

}

**4. JUnit (more) in Detail**

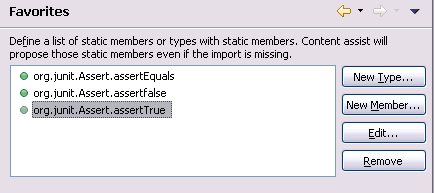
**4.1. Static imports with Eclipse**

JUnit uses a lot of static methods and Eclipse cannot always correctly automatically import static imports.

You can make the JUnit test methods available via the content assists.

Open the Preferences via Window -> Preferences and select Java → Editor → Content Assist → Favorites.

Use the new "New Member" button to add the methods you need. The example below makes the assertTrue, assertFalse and assertEquals methods available.



You can now use Content Assist (Ctrl+Space) to add the method and the import.

I suggest to add at least the following new members:

* org.junit.Assert.assertTrue
* org.junit.Assert.assertFalse
* org.junit.Assert.assertEquals
* org.junit.Assert.fail

**4.2. Annotations**

The following table gives an overview of the available annotations in JUnit 4.x.

**Table 1. Annotations**

| **Annotation** | **Description** |
| --- | --- |
| @Test public void method() | The annotation @Test identifies that a method is a test method. |
| @Before public void method() | Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class). |
| @After public void method() | Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults). |
| @BeforeClass public void method() | Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database. |
| @AfterClass public void method() | Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database. |
| @Ignore | Will ignore the test method. 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. |
| @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. |

**4.3. Assert statements**

The following table gives an overview of the available assert statements.

**Table 2. Test methods**

| **Statement** | **Description** |
| --- | --- |
| fail(String) | Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented. |
| assertTrue(true) / assertTrue(false) | Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented. |
| assertTrue([message], boolean condition) | Checks that the boolean condition is true. |
| assertsEquals([String message], expected, actual) | Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays. |
| assertsEquals([String 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([String], expected, actual) | Checks that both variables refer to the same object. |
| assertNotSame([String], expected, actual) | Checks that both variables refer to different objects. |

**5. Mocking with EasyMock**

Unit testing uses also mocking of objects. In this case the real object is replaced by a replacement which has a predefined behavior the test. There are several frameworks available for mocking. To learn more about mock frameworks please see **[EasyMock Tutorial](http://www.vogella.com/articles/EasyMock/article.html" \t "_top)**

**6. Thank you**

Please help me to support this article: