**JUnit - Tutorial**

**Lars Vogel**

Version 2.3

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

06.12.2012

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision History** | | | |
| Revision 0.1-0.5 | 03.09.2007 | Lars Vogel | JUnit description |
| Revision 0.6 - 2.3 | 10.05.2008 - 06.12.2012 | 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 to unit testing**](http://www.vogella.com/articles/JUnit/article.html#unittesting)

[**1.1. Unit tests and 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)

[**1.3. Available JUnit annotations**](http://www.vogella.com/articles/JUnit/article.html#usingjunit_annotations)

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

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

[**1.6. Run your test outside Eclipse**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse_code)

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

[**2.1. Using JUnit integrated into Eclipse**](http://www.vogella.com/articles/JUnit/article.html#installation_eclipse)

[**2.2. Downloading the JUnit library**](http://www.vogella.com/articles/JUnit/article.html#installation_junit)

[**3. Eclipse support for JUnit**](http://www.vogella.com/articles/JUnit/article.html#eclipse)

[**3.1. Creating JUnit tests**](http://www.vogella.com/articles/JUnit/article.html#eclipse_creatingjunittest)

[**3.2. Running JUnit tests**](http://www.vogella.com/articles/JUnit/article.html#eclipse_runningjunittest)

[**3.3. JUnit static imports**](http://www.vogella.com/articles/JUnit/article.html#usingjunit_staticimports)

[**3.4. Wizard for creating test suites**](http://www.vogella.com/articles/JUnit/article.html#usingjunit_testsuites)

[**3.5. Testing exception**](http://www.vogella.com/articles/JUnit/article.html#usingjunit_execption)

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

[**4.1. Project preparation**](http://www.vogella.com/articles/JUnit/article.html#juniteclipse_prep)

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

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

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

[**5. Advanced JUnit options**](http://www.vogella.com/articles/JUnit/article.html#junitadvanced)

[**5.1. Parameterized test**](http://www.vogella.com/articles/JUnit/article.html#junitadvanced_parameterizedtests)

[**5.2. Rules**](http://www.vogella.com/articles/JUnit/article.html#junitadvanced_rules)

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

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

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

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

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

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

This tutorial is part of this book:

**[](http://www.amazon.com/dp/3943747042)**

**1. Introduction to unit testing**

**1.1. Unit tests and unit testing**

A *unit test* is a piece of code written by a developer that executes a specific functionality in the code which is tested. The percentage of code which is tested by unit tests is typically called *test coverage*.

Unit tests target small units of code, e.g. a method or a class, (local tests) whereas *component and integration tests* targeting to test the behavior of a component or the integration between a set of components or a complete application consisting of several components.

Unit tests ensure that code works as intended. They are also very helpful to ensure that the code still works as intended in case you need to modify code for fixing a bug or extending functionality. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

Typically unit tests are created in their own project or their own source folder to avoid that the normal code and the test code is mixed.

**1.2. Unit testing with JUnit**

*JUnit* *unit test* in version 4.x is a test framework which uses annotations to identify methods that specify a test. Typically these test methods are contained in a class which is only used for testing. It is typically called a *Test class*.

The following code shows a JUnit test method which can be created via *File* → *New* → *JUnit* →*JUnit Test case*.

*@Test*

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

*// MyClass is tested*

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

*// Check if multiply(10,5) returns 50*

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

}

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 you annotate a method with the @org.junit.Test annotation and use a method provided by JUnit to check the expected result of the code execution versus the actual result.

You can use the Eclipse user interface to run the test, via right-click on the test class and selecting *Run*→ *Run As* → *JUnit Test*. Outside of Eclipse you can useorg.junit.runner.JUnitCore class to run the test.

**1.3. Available JUnit 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() | This method is executed before each test. This method can prepare the test environment (e.g. read input data, initialize the class). |
| @After  public void method() | This method is executed after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures. |
| @BeforeClass  public static void method() | This method is executed once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database. Methods annotated with this annotation need to be defined as static to work with JUnit. |
| @AfterClass  public static void method() | This method is executed once, after all tests have been finished. This can be 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 | Ignores 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. |

**1.4. Assert statements**

JUnit provides static methods in the Assert class to test for certain conditions. These *assertion methods* typically start with asserts and 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, and throws an AssertionException if the comparison test fails.

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

**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 a failing test before the test code is 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. |

**Note**

You should provide meaningful messages in assertions so that it is easier for the developer to identify the problem. This help in fixing the issue, especially if someone looks at the problem, which did not write the code under test or the test code.

**1.5. Create a JUnit test suite**

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

The following example code shows a test suite which defines that two test classes should be executed. If you want to add another test class you can add it to @Suite.SuiteClassesstatement.

**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 {

}

**1.6. Run your test outside Eclipse**

Eclipse provides support for running your test interactively in the Eclipse IDE. You can also run your JUnit tests outside Eclipse via standard Java code. The org.junit.runner.JUnitCore class provides the runClasses() method 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());

}

}

}

To run your JUnit tests outside Eclipse you need to add the JUnit library jar to the classpath of your program. Typically build frameworks like Apache Ant or Apache Maven are used to execute tests automatically on a regular basis.

**2. Installation of JUnit**

**2.1. Using JUnit integrated into Eclipse**

Eclipse allows you to use the version of JUnit which is integrated in Eclipse. If you use Eclipse no additional setup is required. In this case you can skip the following section.

**2.2. Downloading the JUnit library**

If you want to control the used JUnit library explicitly, download JUnit4.x.jar from the following JUnit website. 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.

http:*//junit.org/*

**3. Eclipse support for JUnit**

**3.1. Creating JUnit tests**

You can write the JUnit tests manually but Eclipse supports the creation of JUnit tests via wizards.

For example to create a JUnit test or a test class for an existing class, right-click on your new class, select this class in the *Package Explorer* *view*, right-click on it and select *New* → *JUnit Test Case*.

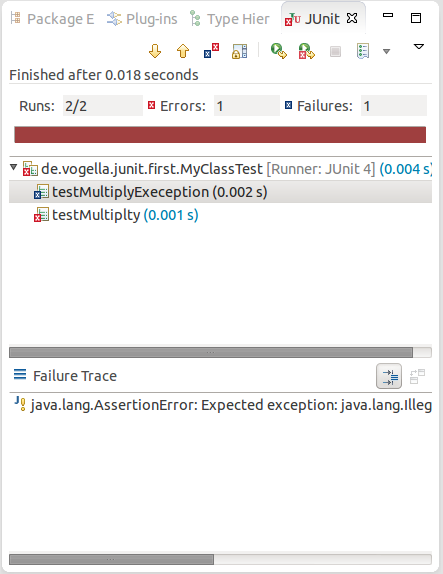
Alternatively you can also use the JUnit wizards available under *File* → *New*→ *Other...* → *Java* →*JUnit*.

**3.2. Running JUnit tests**

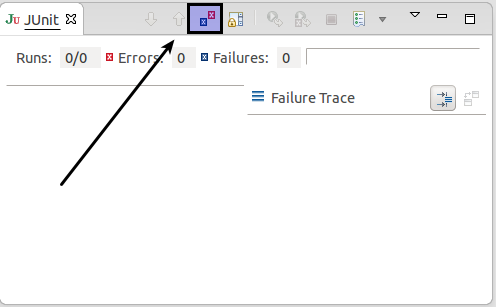
To run a test, select the class which contains the tests, right-click on it and select *Run-as* → *JUnit Test*. This starts JUnit and executes all test methods in this class.

Eclipse provides the **Alt**+**Shift**+**X,** **,T** shortcut to run the test in the selected class. If you position the cursor on one method name, this shortcut runs only the selected test method.

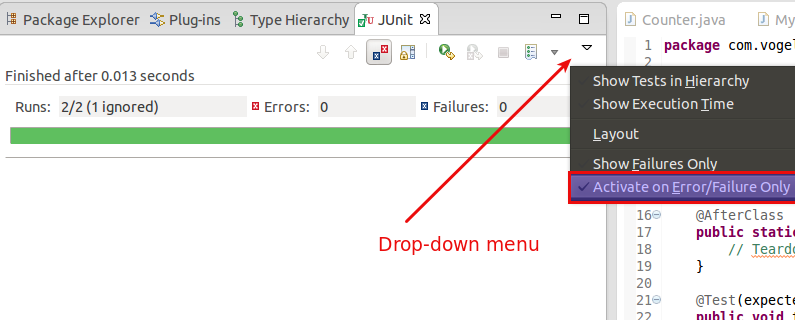
To see the result of an JUnit test, Eclipse uses the *JUnit* *view* which shows the results of the tests. You can also select individual unit test in this *view*, right-click them and select *Run* to execute them again.



By default this *view* shows all tests you can also configure, that it shows only failing tests.



You can also define that the *view* is only activated if you have a failing test.



**Note**

Eclipse creates run configurations for tests. You can see and modify these via the *Run* → *Run Configurations...* menu.

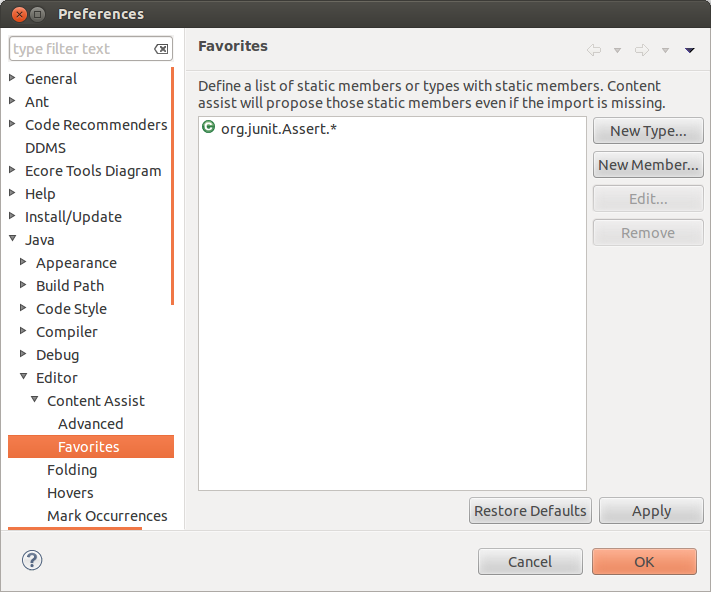
**3.3. JUnit static imports**

JUnit uses static methods and Eclipse cannot always create the corresponding static importstatements automatically.

You can make the JUnit test methods available via the *Content Assists*. *Content Assists* is a functionality in Eclipse which allows the developer to get context sensitive code completion in an editor upon user request.

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

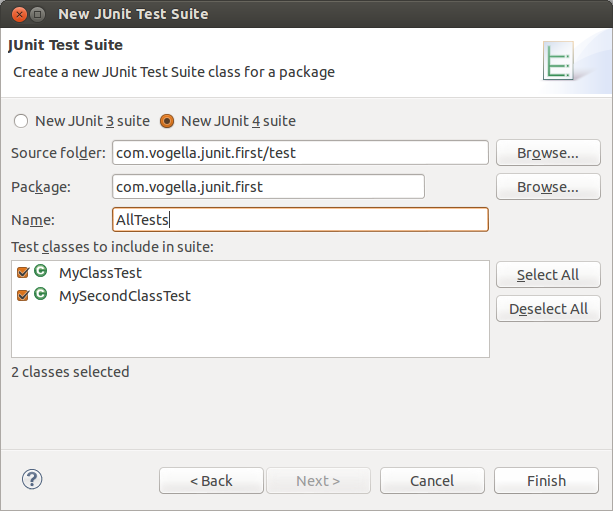
Use the new *New Type* button to add the org.junit.Assert type. This makes for example theassertTrue, assertFalse and assertEquals methods directly available in the *Content Assists*.



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

**3.4. Wizard for creating test suites**

To create a test suite in Eclipse you select the test classes which should be included into this in the *Package Explorer* *view*, right-click on them and select *New*→ *Other...* → *JUnit* → *JUnit Test Suite*.



**3.5. Testing exception**

The @Test (expected = Exception.class) annotation is limited as it can only test for one exception. To test exceptions you can use the following test pattern.

**try** {

mustThrowException();

fail();

} **catch** (Exception e) {

*// expected*

*// could also check for message of exception, etc.*

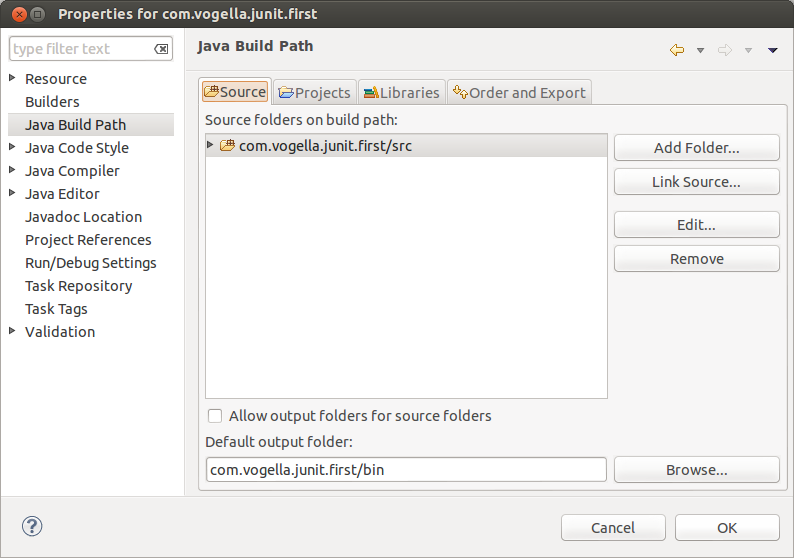
}

**4. Exercise: Using JUnit**

**4.1. Project preparation**

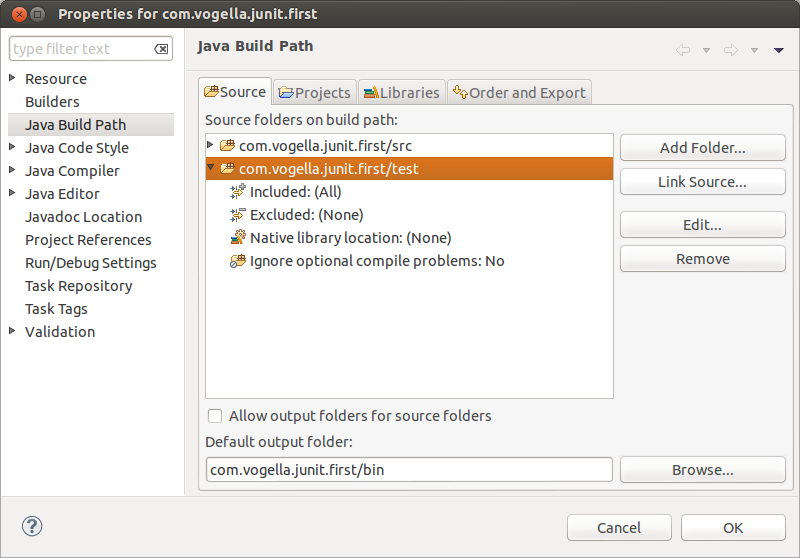
Create a new project called *com.vogella.junit.first*.

Create a new source folder test. For this right-click on your project, select *Properties* and choose the *Java Build Path*. Select the *Source* tab.



Press the *Add Folder* button, afterwards press the *Create New Folder* button. Create the *test*folder.

The result is depicted in the following sceenshot.



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

**4.2. Create a Java class**

In the *src* folder, create the com.vogella.junit.first package and the following class.

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

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

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

*// the following is just an example*

**if** (x > 999) {

**throw** **new** IllegalArgumentException("X should be less than 1000");

}

**return** x / y;

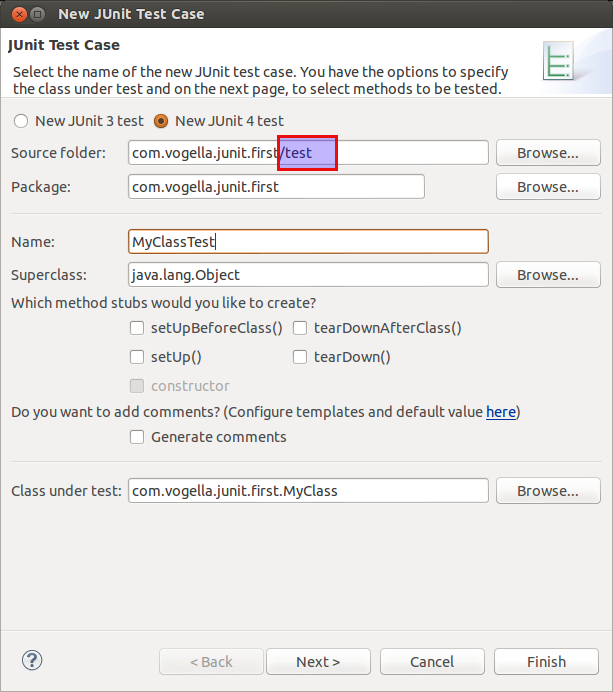
}

}

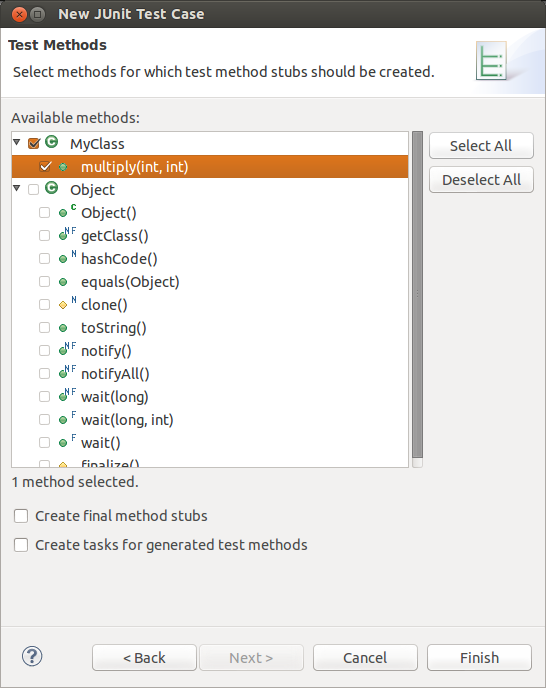
**4.3. Create a JUnit test**

Right-click on your new class in the *Package Explorer* *view* and select *New* → *JUnit Test Case*.

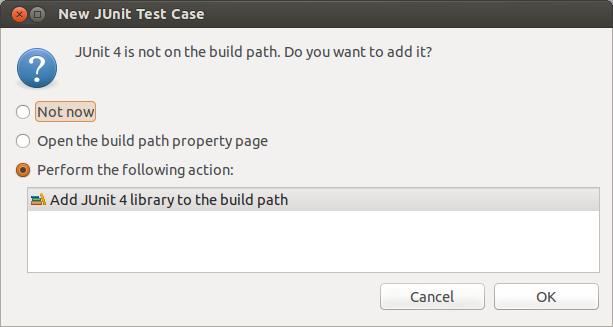
In the following wizard ensure that the *New JUnit 4 test* flag is selected and set the source folder to *test*, so that your test class gets created in this folder.



Press the *Next* button and select the methods that you want to test.



If the JUnit library is not part of the classpath of your project, Eclipse will prompt you to add it. Use this to add JUnit to your project.



Create a test with the following code.

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

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

**import** org.junit.AfterClass;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

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

*@BeforeClass*

**public** **static** **void** testSetup() {

}

*@AfterClass*

**public** **static** **void** testCleanup() {

*// Teardown for data used by the unit tests*

}

*@Test(expected = IllegalArgumentException.class)*

**public** **void** testExceptionIsThrown() {

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

tester.multiply(1000, 5);

}

*@Test*

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

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

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

}

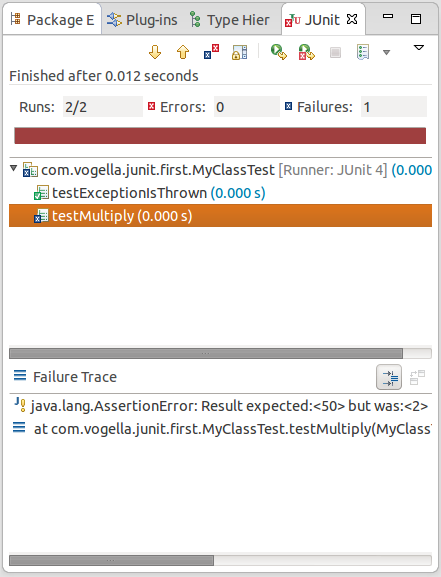
}

**4.4. Run your test in Eclipse**

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

Run JUnit test in Eclipse

The result of the tests will be displayed in the JUnit *view*. In our example one test should be succesful and one test should show an error. This error is indicated by a red bar.



The test is failing 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.

**5. Advanced JUnit options**

**5.1. Parameterized test**

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 @Parameters that generates and returns a Collection of Arrays. Each item in this collection is used as the parameters for the test method.

You need also to create 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.

The following code shows an example for a parameterized test. It assume that you test themultiply() method of the MyClass class which was used in an example earlier.

**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** MyParameterizedClassTest {

**private** **int** multiplier;

**public** MyParameterizedClassTest(**int** testParameter) {

**this**.multiplier = testParameter;

}

*// Creates the test data*

*@Parameters*

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

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

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

}

*@Test*

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

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

assertEquals("Result", multiplier \* multiplier,

tester.multiply(multiplier, multiplier));

}

}

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.

**5.2. Rules**

Via 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 your expect during 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 provides already several useful implementations of rules. For example theTemporaryFolder class allows to setup files and folders which are automatically removed after a test.

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());

}

}

To write your own rule you need to implement the TestRule interface.

**6. 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)**

**7. Thank you**

Please help me to support this article: