JUnit

Unit Testing with JUnit - Tutorial

Quelle: Lars Vogel http://www.vogella.com/tutorials/JUnit/article.html#installation

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. Testing terminology
- 1.1. Unit tests and unit testing
- 1.2. Test fixture
- 1.3. Functional and integration tests
- 1.4. Performance tests
- 1.5. Behavior vs. state testing

2. Test organization
- 2.1. Test organization for Java projects
- 2.2. What should you test?
- 2.3. Introducing tests in legacy code

3. Testing frameworks for Java

4. Using JUnit
- 4.1. The JUnit framework
- 4.2. How to define a test in JUnit?
- 4.3. Example JUnit test
- 4.4. JUnit naming conventions
- 4.5. JUnit test suites
- 4.6. Run your test from the command line

5. JUnit code constructs
- 5.1. Available JUnit annotations

- 5.2. Assert statements
- 5.3. Test execution order

6. Installation of JUnit
- 6.1. Using JUnit integrated into Eclipse
- 6.2. Downloading the JUnit library

7. Eclipse support for JUnit
- 7.1. Creating JUnit tests
- 7.2. Running JUnit tests
- 7.3. JUnit static imports
- 7.4. Wizard for creating test suites
- 7.5. Testing exception

8. Exercise: Setting Eclipse up for using JUnit
- 8.1. Target
- 8.2. Configure Favorites in the preferences

9. Exercise: Using JUnit
- 9.1. Project preparation
- 9.2. Create a Java class
- 9.3. Create a JUnit test
- 9.4. Run your test in Eclipse

10. Advanced JUnit options
- 10.1. Parameterized test
- 10.2. Rules
- 10.3. Categories

11. Mocking

12. Support this website
- 12.1. Thank you
- 12.2. Questions and Discussion

13. Links and Literature
- 13.1. JUnit Resources
- 13.2. vogella Resources

Get the book

1. Testing terminology

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 to be tested. 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, (local tests).

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.

1.2. Test fixture

The *test fixture* is a fixed state of the software under test used as a baseline for running tests.

1.3. Functional and integration tests

An *integration test* has the target 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.

This kind of tests allow you to translate your user stories into a test suite, i.e., the test would resemble an expected user interaction with the application.

1.4. Performance tests

Performance tests are used to benchmark software components in a repeatable way.

1.5. Behavior vs. state testing

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

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

If you are testing algorithms or system functionality, you you want to test in most cases state and not interactions. A typical test setup uses mocks or stubs of related classes to abstract the interactions with these other classes away and tests state in the object which is tested.

2. Test organization

2.1. Test organization for Java projects

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

2.2. What should you test?

What should be tested is a hot topic for discussion. Some developers believe every statement in your code should be tested.

In general it is safe to ignore trivial code as, for example, 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 and you are safe to assume that field assignment works in Java if you are developing end user applications.

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

2.3. Introducing tests in legacy code

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

3. Testing frameworks for Java

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

This description focuses at JUnit.

4. Using JUnit

4.1. The JUnit framework

JUnit in version 4.x is a test framework which uses annotations to identify methods that specify a test.

The main websites for JUnit are the JUnit homepage and the Github project page.

4.2. How to define a test in JUnit?

Typically a JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*.

To write a test with the JUnit 4.x framework you annotate a method with the `@org.junit.Test` annotation.

In this method you use a method provided by the JUnit framework to check the expected result of the code execution versus the actual result.

4.3. Example JUnit test

The following code shows a JUnit test method.

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {

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

   // Tests
   assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
   assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
   assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
 }
```

4.4. JUnit naming conventions

There are several potential naming conventions for JUnit tests. In widespread use is to use the name of the class under test and to add the "Test" suffix to the test class.

Tip

You should prefer the "Test" suffix over "Tests" as the Maven build system (via its surfire plug-in) automatically includes such classes in its test scope.

For the test method names it is frequently recommend to use the word "should" in the test method name, as for example "ordersShouldBeCreated" or "menuShouldGetActive" as this gives a good hint what should happen if the test method is executed.

As a general rule, a test name should explain what the test does so that it can be avoided to read the actual implementation.

4.5. JUnit test suites

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 in the specified order.

The following example code shows a test suite which defines that two test classes (MYClassTest and MySecondClassTest) should be executed. If you want to add another test class you can add it to `@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 {

}
```

4.6. Run your test from the command line

You can also run your JUnit tests outside Eclipse via standard Java code. Build frameworks like Apache Ant or Apache Maven are typically used to execute tests automatically on a regular basis.

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.

The following class demonstrates how to run the MyClassTest. 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.

5. JUnit code constructs

5.1. Available JUnit annotations

JUnit 4.x uses annotations to mark methods and to configure the test run. The following table gives an overview of the most important available annotations.

Table 1. Annotations

Annotation	Description
@Test public void method()	The `@Test` annotation identifies a method as a test method.
@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.
@Before public void method()	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After public void method()	This method is 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 public static void method()	This method is 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 public static void method()	This method is 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.

Annotation	Description
@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.

5.2. Assert statements

JUnit provides static methods in the `Assert` class to test for certain conditions. These *assertion methods* typically start with `assert` 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. The String 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([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([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.

Statement	**Description**
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 helps in fixing the issue, especially if someone looks at the problem, who did not write the code under test or the test code.

5.3. Test execution order

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 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 `@FixMethodOrder(MethodSorters.NAME_ASCENDING)`.

Note

The default in JUnit 4.11 is to uses a deterministic, but not predictable, order which can also be explicitly specified via the `MethodSorters.DEFAULT` parameter in the above annotation. You can also use `MethodSorters.JVM` which uses the JVM defaults, which may vary from run to run.

6. Installation of JUnit

6.1. 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.

6.2. 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/`

7. Eclipse support for JUnit

7.1. 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.

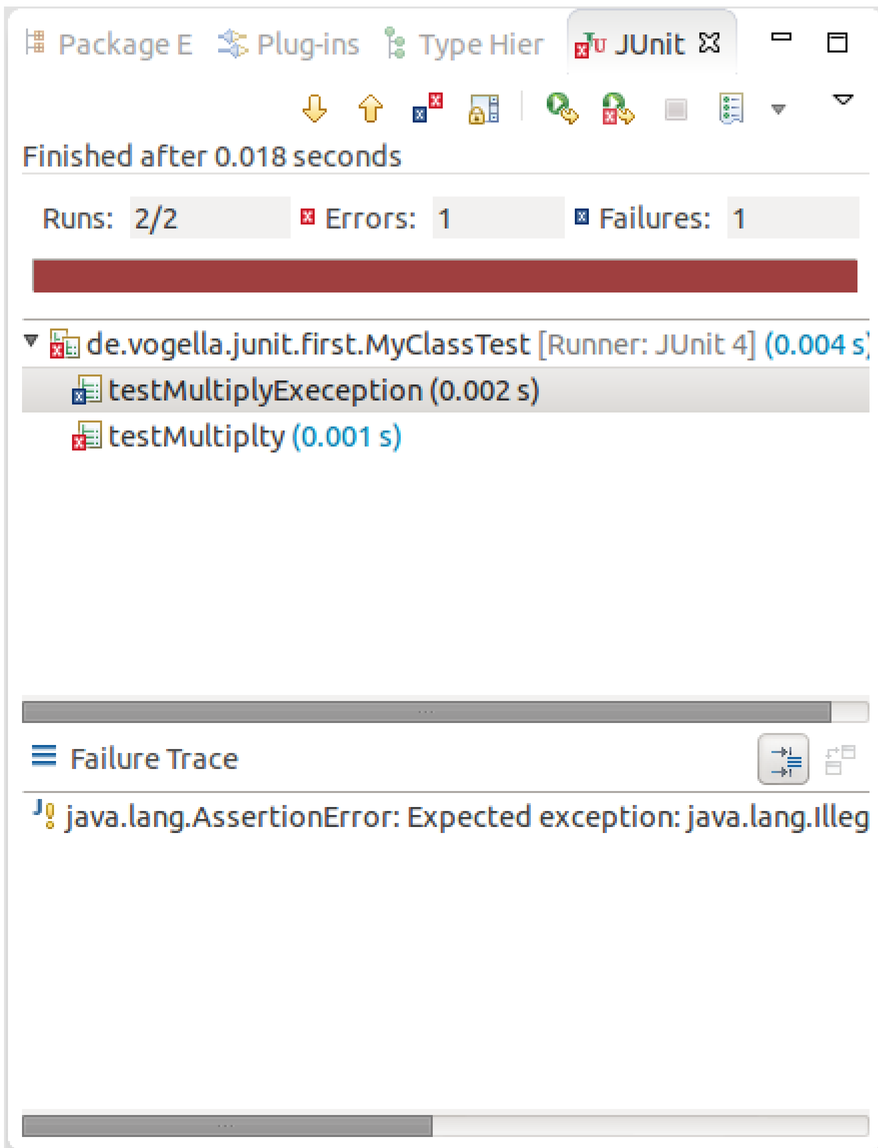
7.2. Running JUnit tests

The Eclipse IDE also provides support for executing your tests interactively.

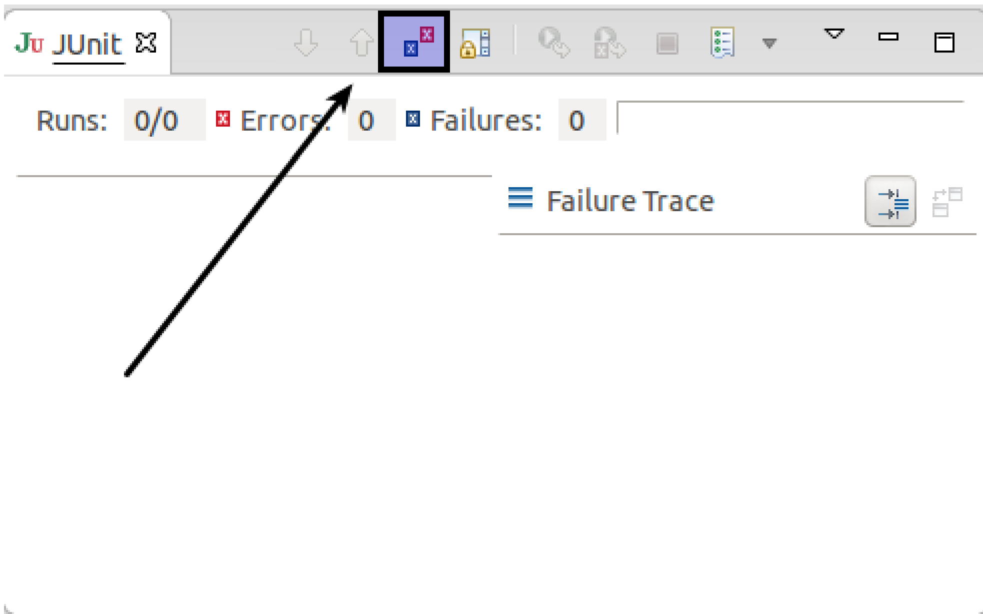
To run a test, select the class which contains the tests, right-click on it and select Run-as → JUnit Test. This starts JUnit and executes all test methods in this class.

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

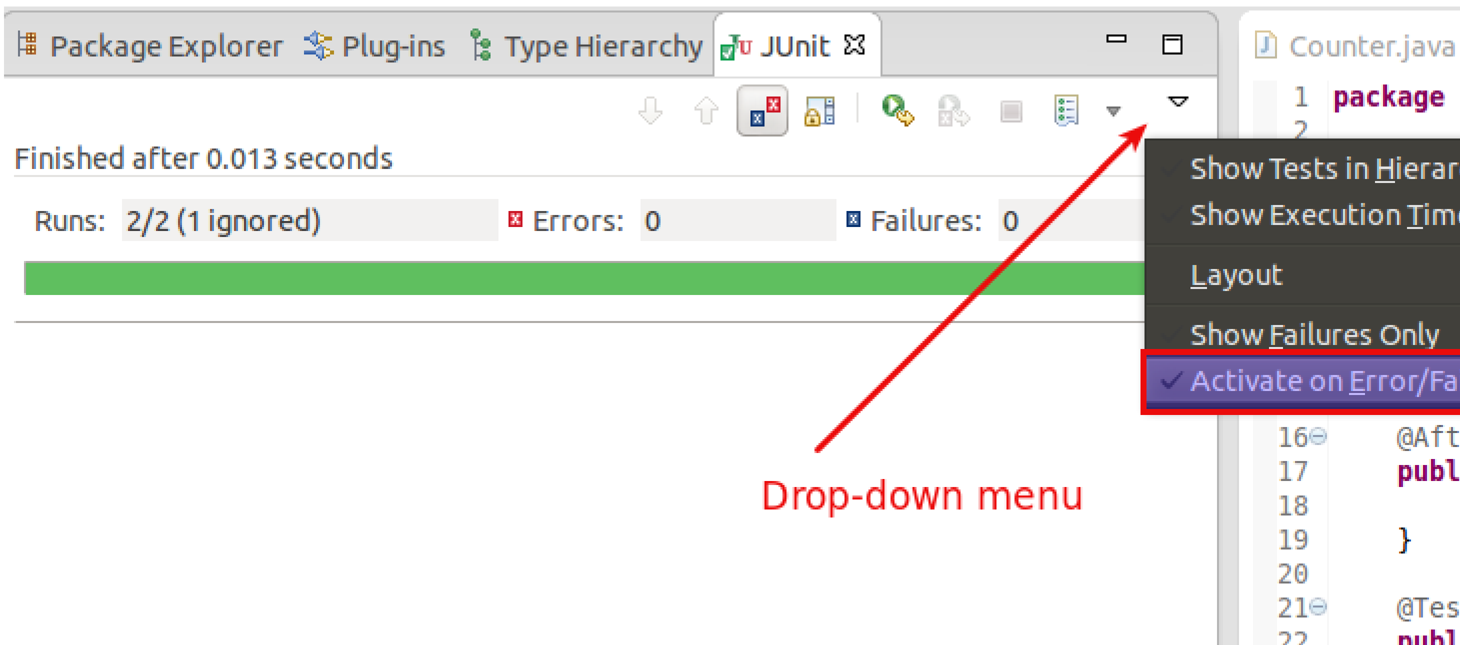
To see the result of an JUnit test, Eclipse uses the JUnit view which shows the results of the tests. You can also select individual unit tests in this view , right-click on them and select Run to execute them again.



By default this view shows all tests. You can also configure, that it only shows 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.

7.3. JUnit static imports

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

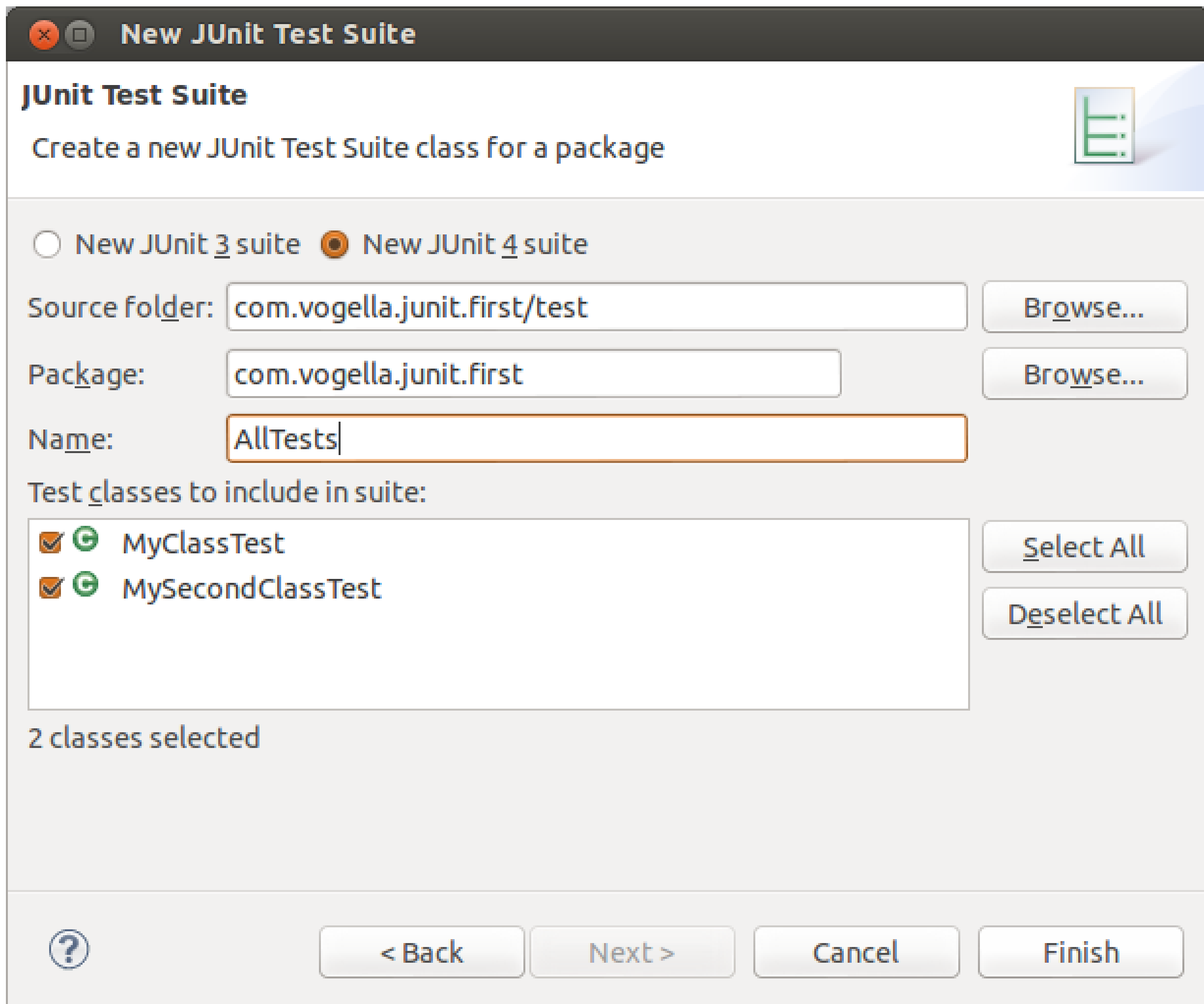
JUnit assert statement are typically defined as `public static` to allow the developer to write short test statements. The following snippet demonstrates an assert statement with and without static imports.

```
// without static imports you have to write the following statement
Assert.assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));


// alteratively define assertEquals as static import
import static org.junit.Assert.assertEquals;
// more code
// use it without the prefix
assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
```

The Eclipse IDE cannot always create the corresponding `static import` statements 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. See Section 8.2, “Configure Favorites in the preferences” for the required setup.

7.4. 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.



7.5. 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.
}
```

8. Exercise: Setting Eclipse up for using JUnit

8.1. Target

In this exercise you want to configure Eclipse to allow you to use code completion to insert typical JUnit method calls.

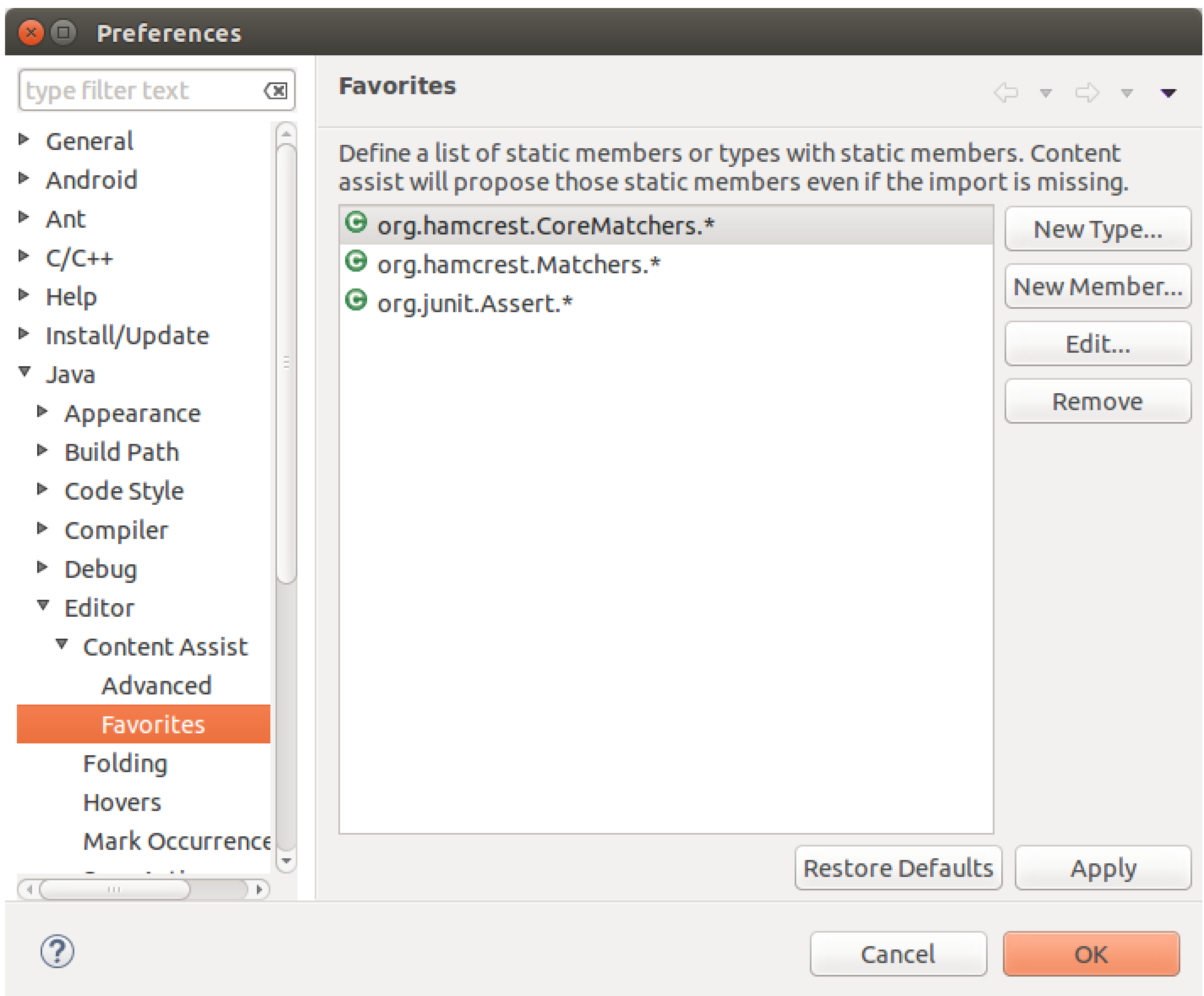
8.2. Configure Favorites in the preferences

Open the Preferences via Window → Preferences and 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*.



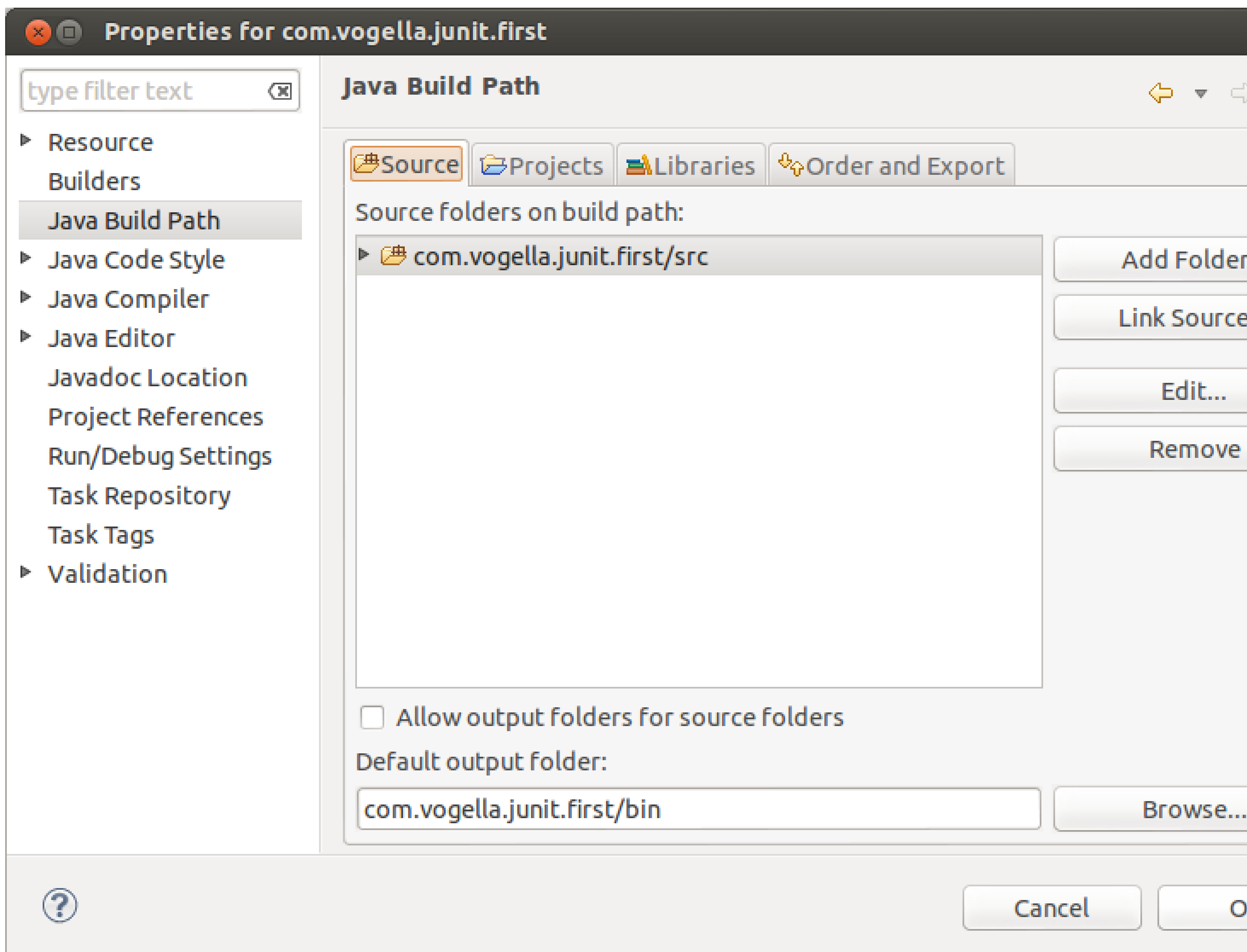
You can now use *Content Assists* (shortcut: **Ctrl+Space**) to add the method and the import.

9. Exercise: Using JUnit

9.1. Project preparation

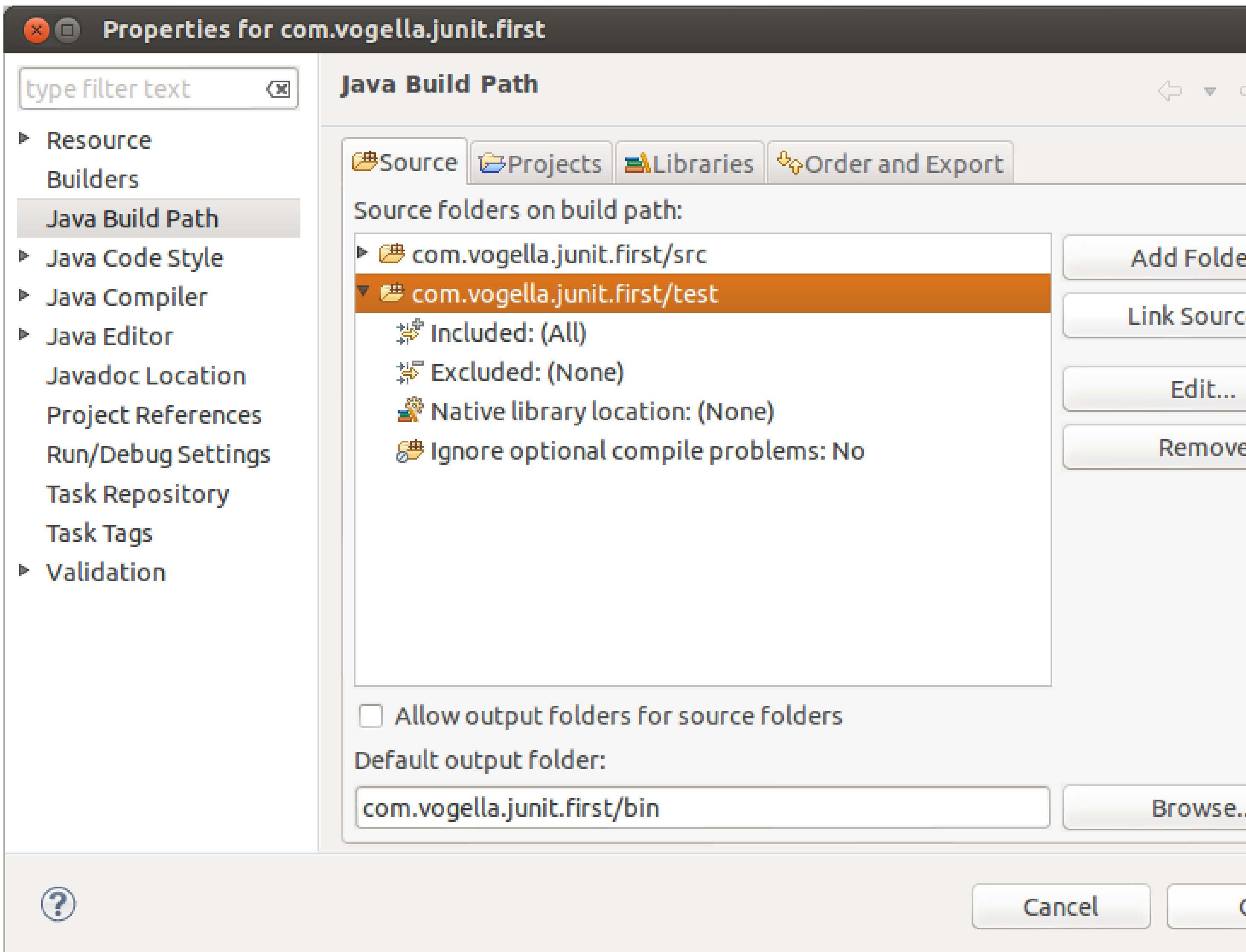
Create a new project called *com.vogella.junit.first*.

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



Press the Add Folder button. Afterwards, press the Create New Folder button. Enter `test` as folder name.

The result is depicted in the following screenshot.



Tip

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

9.2. Create a Java class

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

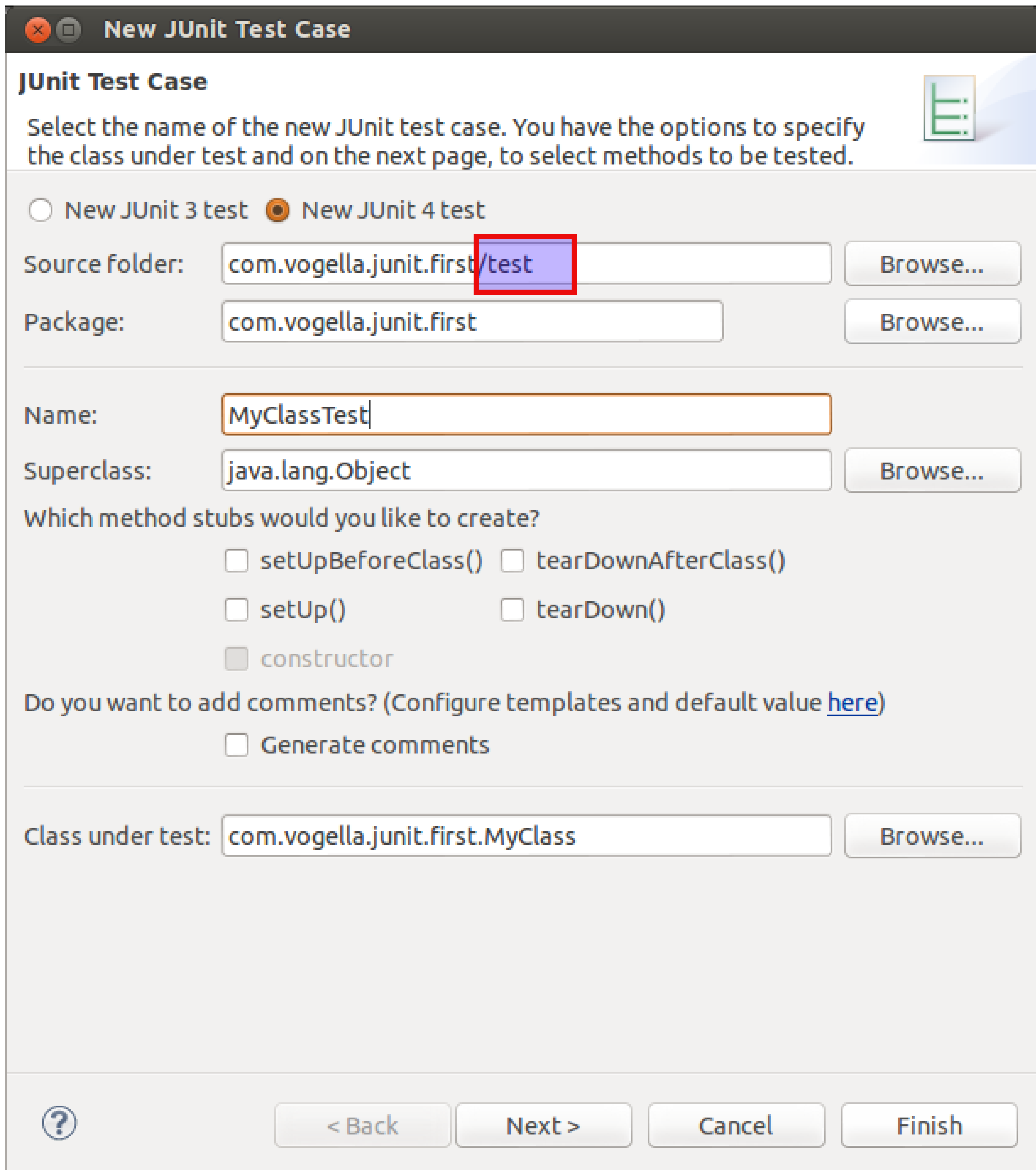
```
package com.vogella.junit.first;

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;
  }
}
```

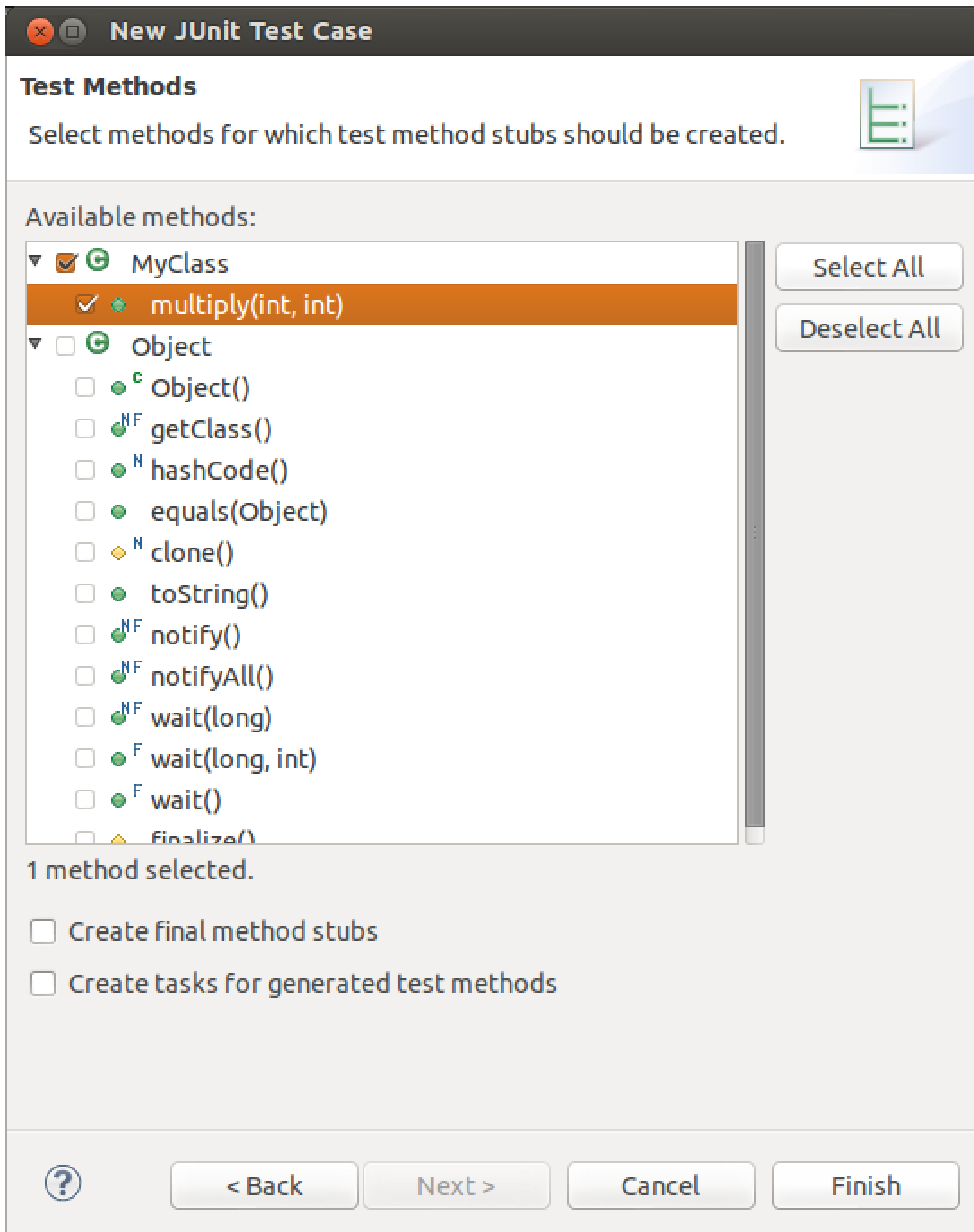
9.3. Create a JUnit test

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

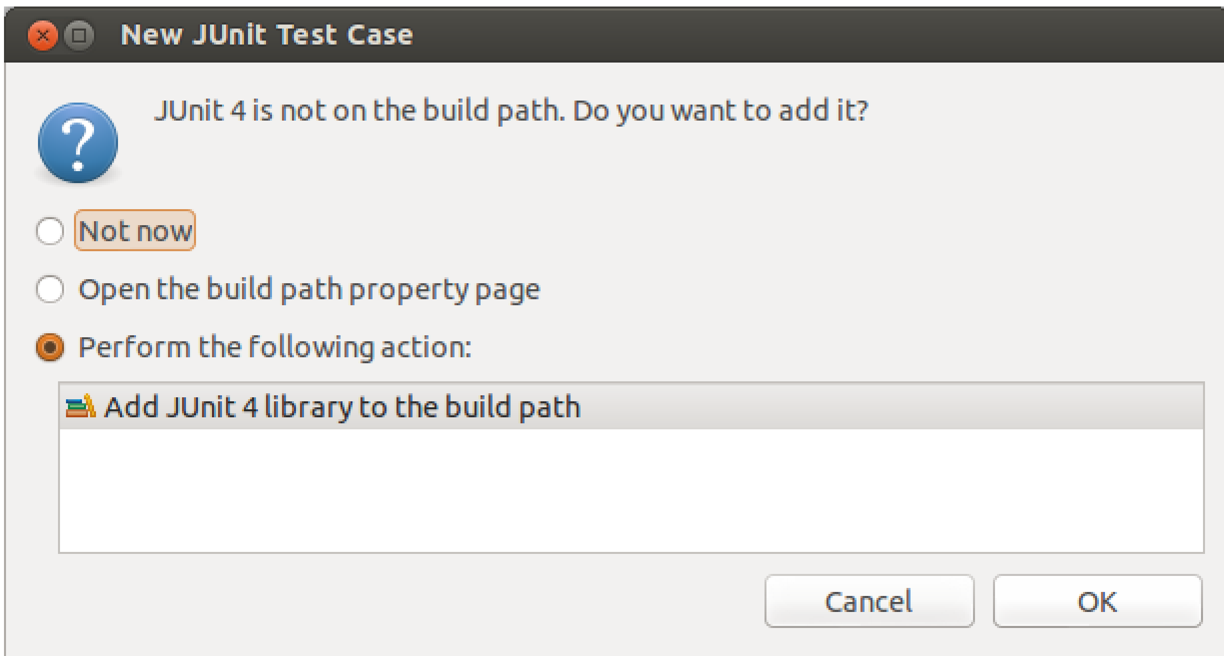
In the following wizard ensure that the New JUnit 4 test flag is selected and set the source folder to `test`, so that your test class gets created in this folder.



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.

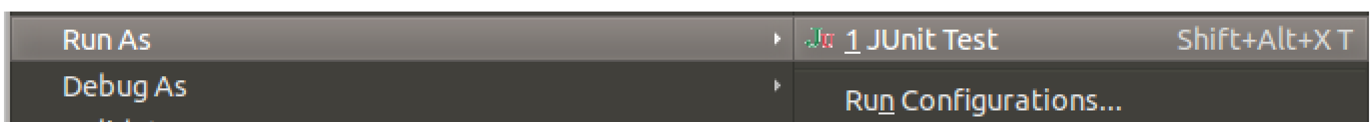


Create a test with the following code.

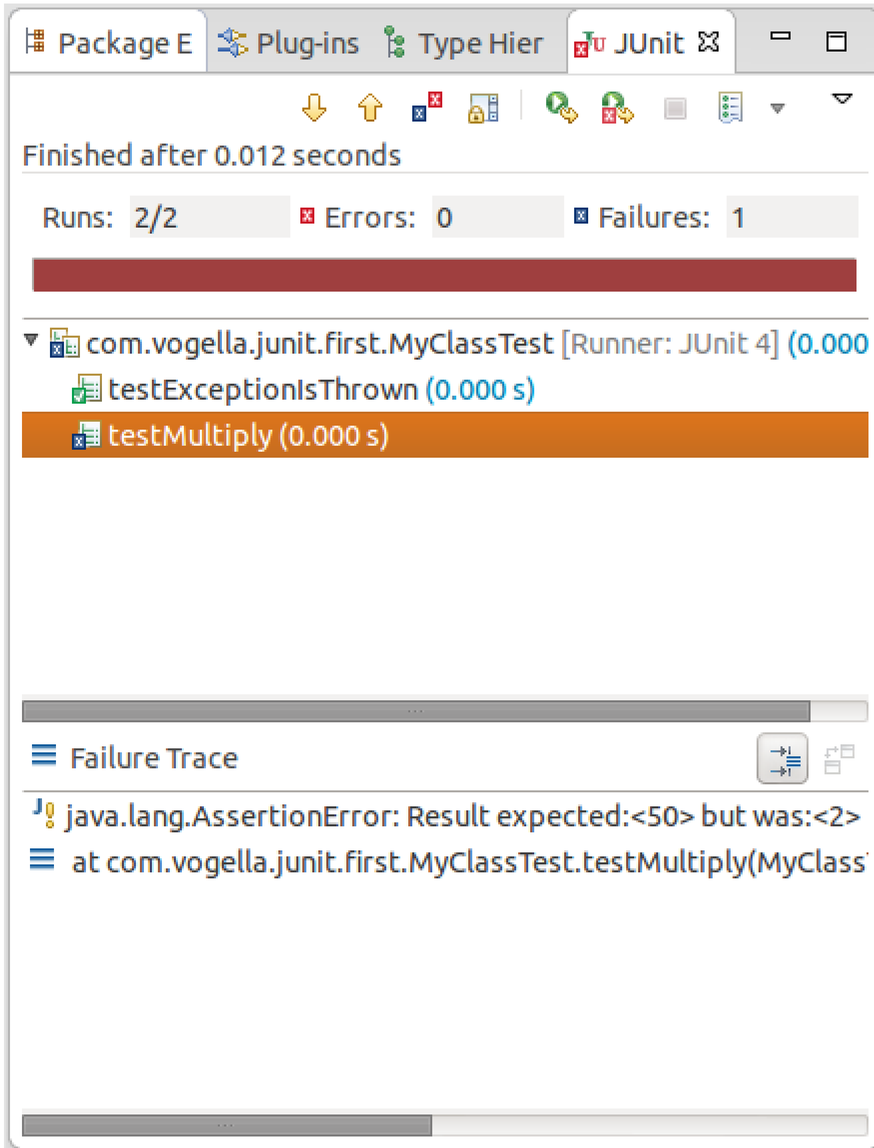
```
package com.vogella.junit.first;

import static org.junit.Assert.assertEquals;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

public class MyClassTest {

  @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));
  }
}
```

9.4. Run your test in 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 . 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 the test to get a green bar.

10. Advanced JUnit options

10.1. 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 parameter for the test method.

You also need 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 assumes that you test the `multiply()` 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.

10.2. 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 you 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 already provides several useful implementations of rules. For example, the `TemporaryFolder` 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 custom rule, you need to implement the `TestRule` interface.

10.3. 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.

```
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
}
```

11. Mocking

Unit testing also makes use of object mocking. In this case the real object is exchanged by a replacement which has a predefined behavior for the test.

There are several frameworks available for mocking. To learn more about mock frameworks please see the Mockito tutorial and the EasyMock tutorial

12. Support this website

This tutorial is Open Content under the CC BY-NC-SA 3.0 DE license. Source code in this tutorial is distributed under the Eclipse Public License. See the vogella License page for details on the terms of reuse.

Writing and updating these tutorials is a lot of work. If this free community service was helpful, you can support the cause by giving a tip as well as reporting typos and factual errors.

12.1. Thank you

Please consider a contribution if this article helped you. It will help to maintain our content and our Open Source activities.

12.2. Questions and Discussion

If you find errors in this tutorial, please notify me (see the top of the page). Please note that due to the high volume of feedback I receive, I cannot answer questions to your implementation. Ensure you have read the vogella FAQ as I don't respond to questions already answered there.

13. Links and Literature

13.1. JUnit Resources

Eclipse IDE book from Lars Vogel

JUnit Homepage

13.2. vogella Resources

vogella Training Android and Eclipse Training from the vogella team

Android Tutorial Introduction to Android Programming

GWT Tutorial Program in Java, compile to JavaScript and HTML

Eclipse RCP Tutorial Create native applications in Java

JUnit Tutorial Test your application

Git Tutorial Put all your files in a distributed version control system

Quellen:

http://www.mm.informatik.tu-darmstadt.de/courses/helpdesk/junit4.html

http://www.vogella.de/articles/JUnit/article.html#installation

http://www.torsten-horn.de/techdocs/java-junit.htm

D. Waldvogel, 13. Dezember 2014
M226_JUnit_Eclipse_1.0.docx