

Объектно- ориентированное программирование на языке Java

Часть 3. TDD и JUnit

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
<http://www.berkut.mk.ua>

Tetyana Smykodub, NUoS



История

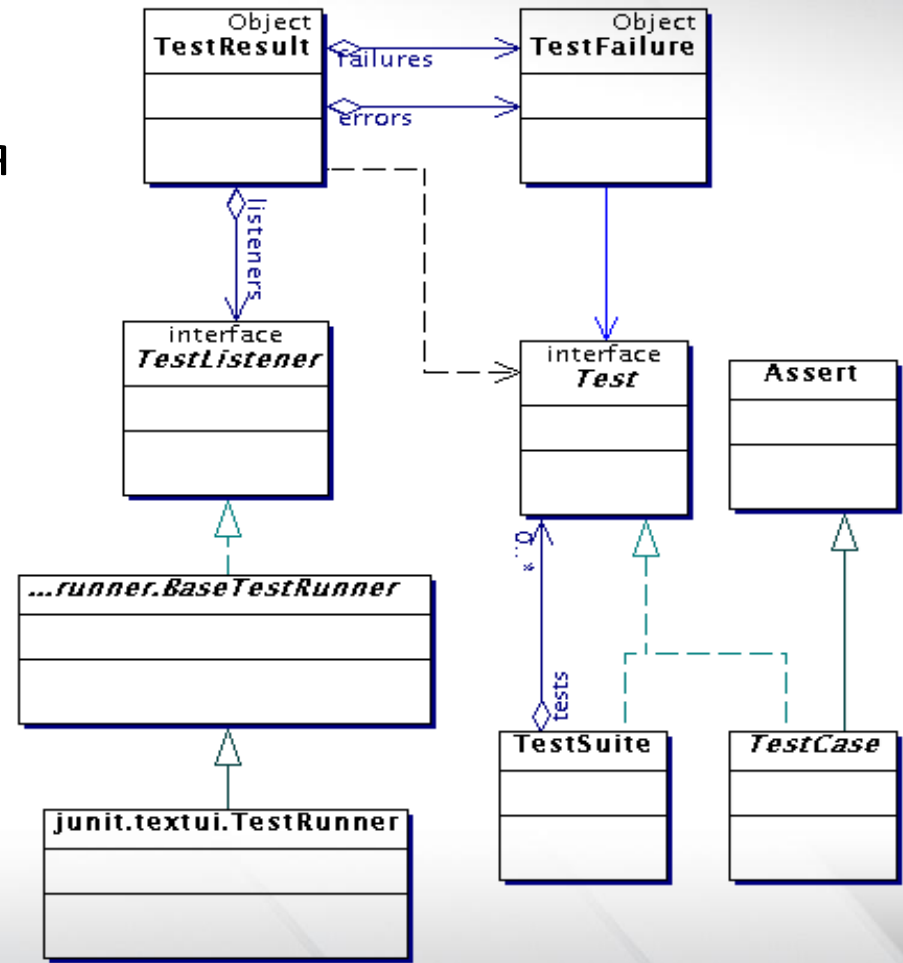
- В середине 90-х годов Кент Бек разработал первый автоматизированный тестовый инструмент xUnit для Smalltalk
- Бек и Гамма разработали JUnit на рейсе из Цюриха в Вашингтон, округ Колумбия.
- Мартин Фаулер: «Никогда в области разработки программного обеспечения не было так много строк кода».
- JUnit стал стандартным инструментом для тестовой разработки на Java (см. junit.org)
- Генераторы тестов JUnit теперь являются частью многих Java IDE (IntelliJ IDEA, NetBeans, Eclipse, BlueJ, ...)

Зачем создавать тестовый набор?

- Очевидно, вам нужно проверить свой код?
 - Вы можете проводить специальное тестирование (выполняя тесты, которые необходимы на данный момент), или
 - Вы можете создать набор тестов (полный набор тестов, которые можно запустить в любое время)
- Недостатки набора тестов
 - Много дополнительного программирования
 - Правда, использование хорошей тестовой платформы может немного помочь
 - У вас нет времени на выполнение этой дополнительной работы
 - Ложь! Практика показывает, что тестовые комплекты сокращают время отладки на количество часов больше, чем было бы, потрачено на создание набора тестов
- Преимущества тестового набора
 - Снижает общее количество ошибок в поставляемом коде
 - Делает код гораздо более рефакторизуемым

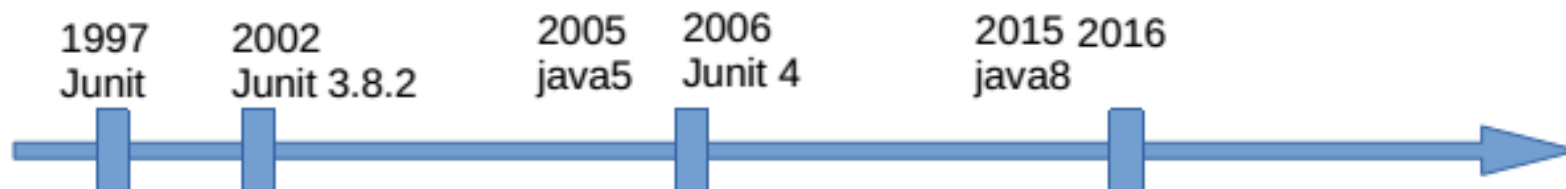
Диаграмма классов junit.framework

- JUnit test framework - это пакет классов, который позволяет писать тесты для каждого метода, а затем легко выполнять эти тесты
- TestRunner запускает тесты и отчеты TestResults
- Вы тестируете свой класс, расширяя абстрактный класс? TestCase (необязательно)
- Чтобы написать тестовые примеры, вам нужно знать и понимать класс Assert



История Junit

Код не менялся годами (аннотация с Java 5)



Junit 3

```
class MyTest extends
    TestCase
{
    public void testFoo() {
        Assert.assertEquals(...
    );
}
```

Junit 4

supports Java 5
Annotations
@Test
public void foo() {
}
No extends TestCase

Junit5 in alpha version

Пишем TestCase

- Чтобы начать использовать JUnit3, раньше нужно было создавать подкласс TestCase, к которому вы планируете добавлять методы тестирования. Сейчас (JUnit 4 и JUnit 5) этого делать не нужно.
- Название класса важно - должно быть в форме MyClassTest
- Это соглашение об именах позволяет TestRunner автоматически находить ваши тестовые классы

```
import org.junit.jupiter.api.BeforeEach;
```

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

```
class MainTest {  
    @BeforeEach  
    void setUp() {  
        .....  
    }  
}
```

Пишем методы в TestCase

- Настроить предварительные условия
- Проверить функционал с помощью тестов
- Проверка постусловий
- Пример:

```
public void testEmptyList() {  
    Bowl emptyBowl = new Bowl();  
    assertEquals("Size of an empty list should be zero.",  
        0, emptyList.size());  
    assertTrue("An empty bowl should report empty.",  
        emptyBowl.isEmpty());  
}
```

Обратить внимание:

- Специфическая подпись метода - public void **test**Whatever () была обязательной до появления аннотаций.
- Кодировать следует по схеме
- Обратите внимание на вызовы **assert**-type ...

Пишем методы в TestCase

Следующий код показывает тест JUnit с использованием версии JUnit 5. В этом тесте предполагается, что класс MyClass существует и имеет метод multiply (int, int).

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.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");}
}
```

В JUnit 3 и JUnit 4 :

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


Assert методы

- У каждого assert метода есть параметры, такие как: *message*, *expected-value*, *actual-value*
- Assert методы, касающиеся действительных чисел, получают дополнительный аргумент
- Каждый метод assert имеет эквивалентную версию, которая не принимает сообщение, однако это использование не рекомендуется, потому что:
 - сообщения помогают документировать тесты
 - сообщения предоставляют дополнительную информацию при чтении журналов ошибок

Assert методы

- JUnit предоставляет статические методы для проверки определенных условий с помощью класса Assert.
- Эти методы обычно начинаются с `assert`. Они позволяют указать сообщение об ошибке, ожидаемый и фактический результат, `assert`-метод сравнивает фактическое значение, возвращаемое тестом, с ожидаемым значением. Срабатывает `AssertionException`, если сравнение не выполняется.

В приведенной ниже таблице дается обзор этих методов. Параметры в скобках `[]` являются необязательными и имеют тип `String`.

Assert методы (JUnit 3)

метод	Описание
<code>fail([message])</code>	Пусть метод терпит неудачу. Может быть использован для проверки того, что определенная часть кода не была достигнута, или для выполнения теста на отказ до того, как будет реализован тестовый код. Параметр сообщения является необязательным.
<code>assertTrue([message,] boolean condition)</code>	Проверяет, что условие <code>boolean</code> истинно
<code>assertFalse([message,] boolean condition)</code>	Проверяет, что условие <code>boolean</code> ложно.
<code>assertEquals([message,] expected, actual)</code>	Проверяет, на равенство два значения. Примечание: для массивов ссылка проверяется не на содержимое массивов.

Assert методы (JUnit 3)

метод	Описание
<code>assertEquals([message,] expected, actual, tolerance)</code>	Проверяет, совпадают ли значения float или double. tolerance - это точность для десятичных знаков, которые должны совпадать.
<code>assertNull([message,] object)</code>	Проверяет, объект имеет значение null.
<code>assertNotNull([message,] object)</code>	Проверяет, объект имеет значение не null.
<code>assertSame([message,] expected, actual)</code>	Проверяет, относятся ли обе переменные к одному и тому же объекту.
<code>assertNotSame([message,] expected, actual)</code>	Проверяет, что обе переменные относятся к разным объектам.

Определение методов тестирования

JUnit 4	Описание
<code>import org.junit.*</code>	Импорт junit для использования следующих аннотаций.
<code>@Test</code>	Определяет метод в качестве метода тестирования.
<code>@Before</code>	Выполняется перед каждым тестом. Он используется для подготовки тестовой среды (например, чтение входных данных, инициализация класса).
<code>@After</code>	Выполняется после каждого теста. Он используется для очистки тестовой среды (например, удаление временных данных, восстановление значений по умолчанию). Он также может экономить память, очищая дорогие структуры памяти.

Определение методов тестирования

JUnit 4	Описание
@BeforeClass	Выполняется один раз, перед началом всех тестов. Он используется для выполнения действий, требующих много времени, например, для подключения к базе данных. Методы, отмеченные этой аннотацией, должны быть определены как статические для работы с JUnit.
@AfterClass	Выполняется один раз, после завершения всех тестов. Он используется для выполнения действий по очистке, например, для отключения от базы данных. Методы, аннотированные этой аннотацией, должны быть определены как статические для работы с JUnit.

Определение методов тестирования

JUnit 4	Описание
@Ignore or @Ignore("Why disabled")	Отмечает, что тест должен быть отключен. Это полезно, когда базовый код был изменен, и тестовый пример еще не адаптирован. Или если время выполнения этого теста слишком велико для включения. Лучше всего предоставить необязательное описание, почему тест отключен.
@Test (expected = Exception.class)	Ошибка, если метод не выбрасывает именованное исключение.
@Test(timeout=100)	Ошибка, если метод выполняется больше 100 миллисекунд.

JUnit 5

Итак, официальный сайт начинается с того, что сообщает нам о новом строении JUnit:

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage (← [офф.сайт](#)).

JUnit Platform — фундаментальная основа для запуска на JVM фреймворков для тестирования. Платформа предоставляет [TestEngine API](#), для разработки фреймворков (для тестирования), которые могут быть запущены на платформе. Кроме этого, в платформе имеется [Console Launcher](#) для запуска платформы из командной строки а также для запуска любого JUnit 4 Runner'a на платформе.

JUnit Jupiter — сердце JUnit 5. Этот проект предоставляет новые возможности для написания тестов и создания собственных расширений. В проекте реализован специальный TestEngine для запуска тестов на ранее описанной платформе. (не поддерживает Rules и Runners)

JUnit Vintage — поддержка легаси. Определяется TestEngine для запуска тестов ориентированных на JUnit 3 и JUnit 4.

JUnit 5. Обзор нововведений

1. JUnit больше не требует, чтобы методы были публичными.
2. **Продвинутый assert**
 - 2.1 Опциональное сообщение сделали последним аргументом.
 - 2.2 Добавили интересный метод для сравнения набора строк. Поддерживаются регулярные выражения!

```
Assertions.assertLinesMatch(  
    asList("можно сравнивать строки", "а можно по  
        regex: \\d{2}\\.\\.\\d{2}\\.\\.\\d{4}"),  
    asList("можно сравнивать строки", "а можно по  
        regex: 12.09.2017")  
);
```

и др.

Ссылки на более подробную информацию:

[официальный сайт JUnit 5](#) и [очень дружелюбное руководство](#).

Тестовые классы

- Предположим, вы хотите протестировать класс **Counter**
- **public class CounterTest {**
 - Это тестовый класс Counter
- **public CounterTest() { }** // Конструктор по умолчанию
- **protected void setUp()**
 - Тест fixture создает и инициализирует переменные экземпляра и т. д.
- **protected void tearDown()**
 - Освобождает любые системные ресурсы, используемые тестовым fixture
- **public void testIncrement(), public void testDecrement()**
 - Эти методы содержат тесты для методов **increment()**, **decrement()** и т.д. класса **Counter**.

Тестовые классы

fixture - это фрагмент кода, который вы хотите запустить перед каждым тестированием

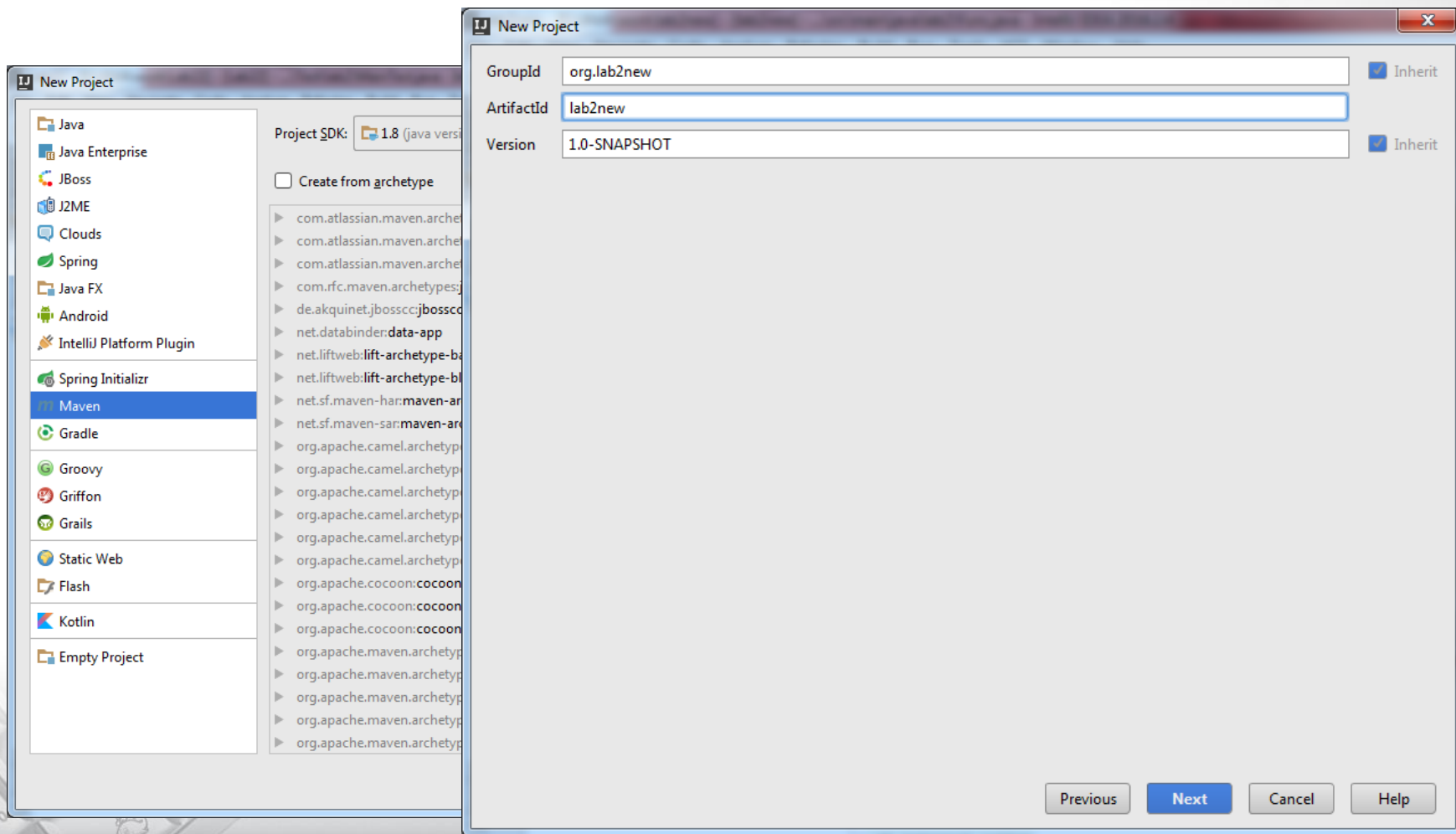
- Вы получаете инструмент , переопределяя метод
`protected void setUp () {...}`
- Общее правило для запуска теста:
`protected void runTest () {
 setUp(); <запустить тест> tearDown ();
}`
- поэтому мы можем переопределить setUp и / или tearDown, и этот код будет выполняться до или после каждого теста

setUp ()

- Переопределяйте setUp () для инициализации переменных и объектов
- Поскольку setUp () является вашим кодом, вы можете модифицировать его любым способом (например, создавая в нем новые объекты)
- Уменьшает дублирование кода

JUnit4 в IntelliJ IDEA

Создадим **Maven Project** в IntelliJ IDEA.



JUnit4 в IntelliJ IDEA

В Maven каждый проект идентифицируется парой `groupId` `artifactId`. Во избежание конфликта имён, **groupId** - **наименование организации или подразделения** и обычно действуют такие же правила как и при именовании пакетов в Java - записывают доменное имя организации или сайта проекта. **artifactId** - **название проекта**.

Внутри тэга **version**, как можно догадаться хранится **версия проекта**. Если состояние кода для проекта не зафиксировано, то в конце к имени версии добавляется "-SNAPSHOT" что обозначает, что версия в разработке и результирующий jar файл может меняться.

Давайте лучше рассмотрим на примере проекта

Lab2new **groupId** - org.lab2new, **artifactId** - lab2new , **version** - 1.4.6

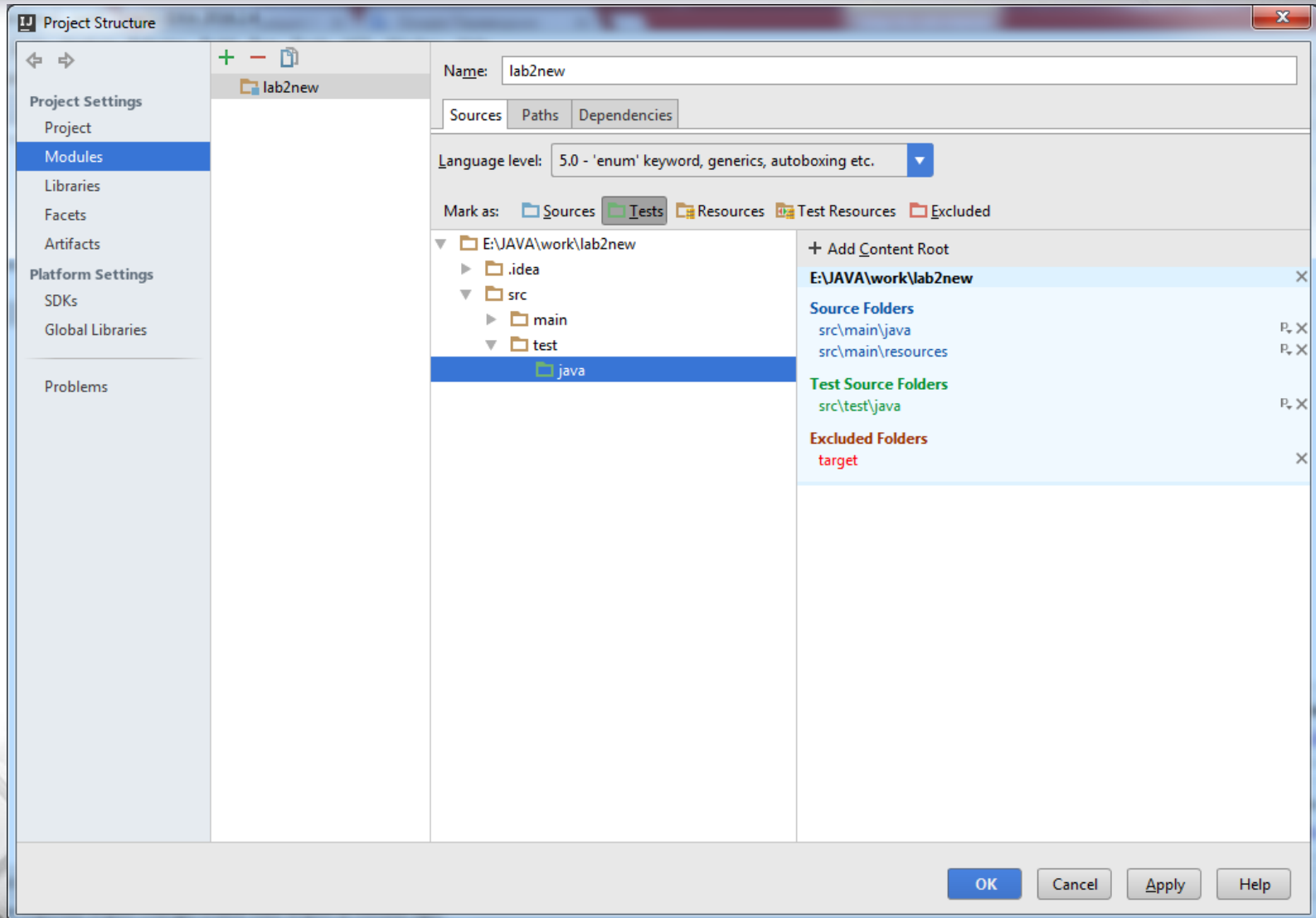
JUnit4 в IntelliJ IDEA

Обязательно проверьте: папка, которая лежит в **test/java** должна быть зеленого цвета - это будет обозначать, что в данной папке лежат тестовые классы и при сборке проекта они не будут собираться в проект.

Если же она не зеленая, то заходим в **Project Structure(Ctrl+Alt+Shift+S)**, далее выбираем слева Modules->Sources и указываем, что папка **test/java** будет тестовым ресурсом. Пример на картинке ниже.

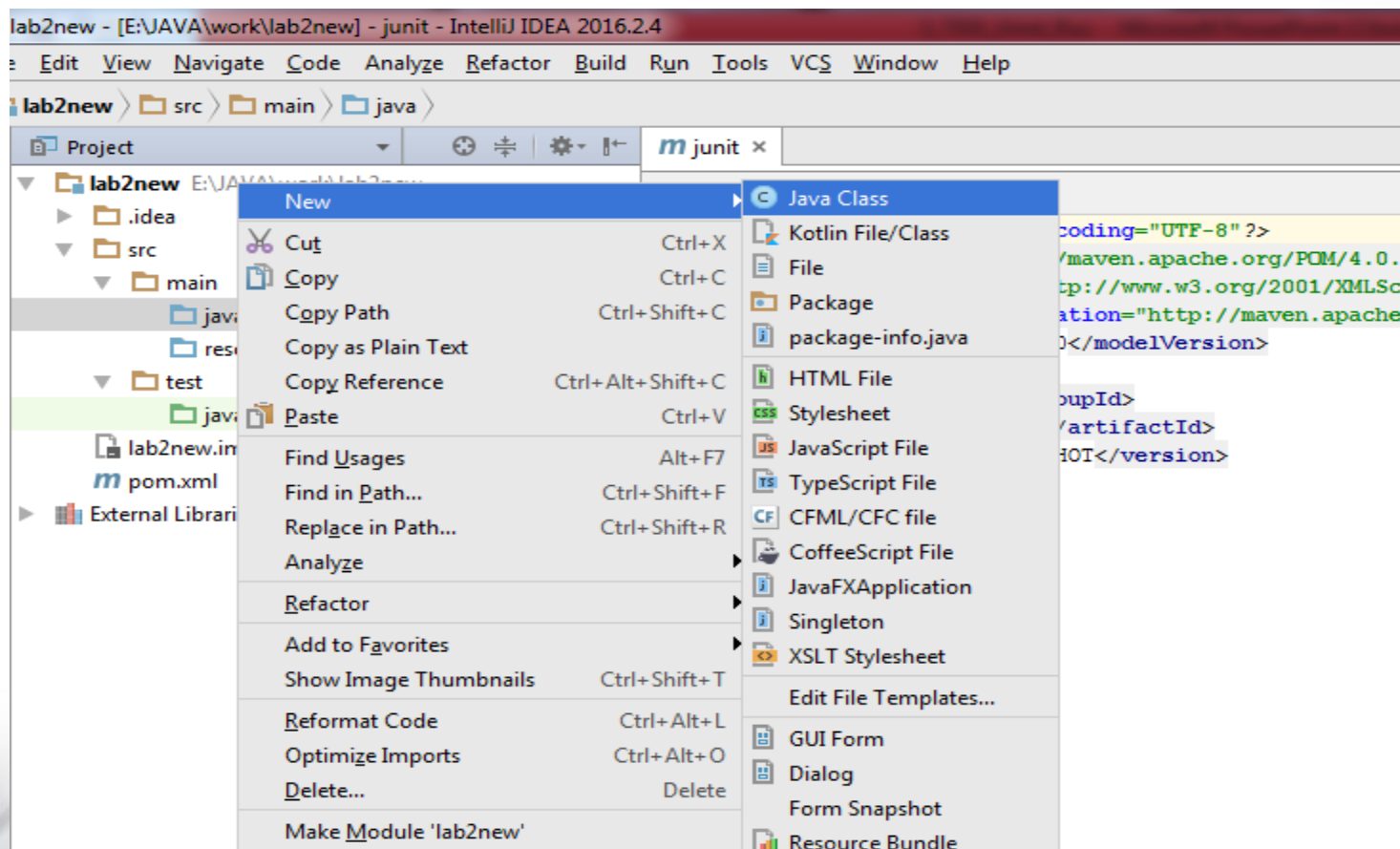


JUnit4 в IntelliJ IDEA



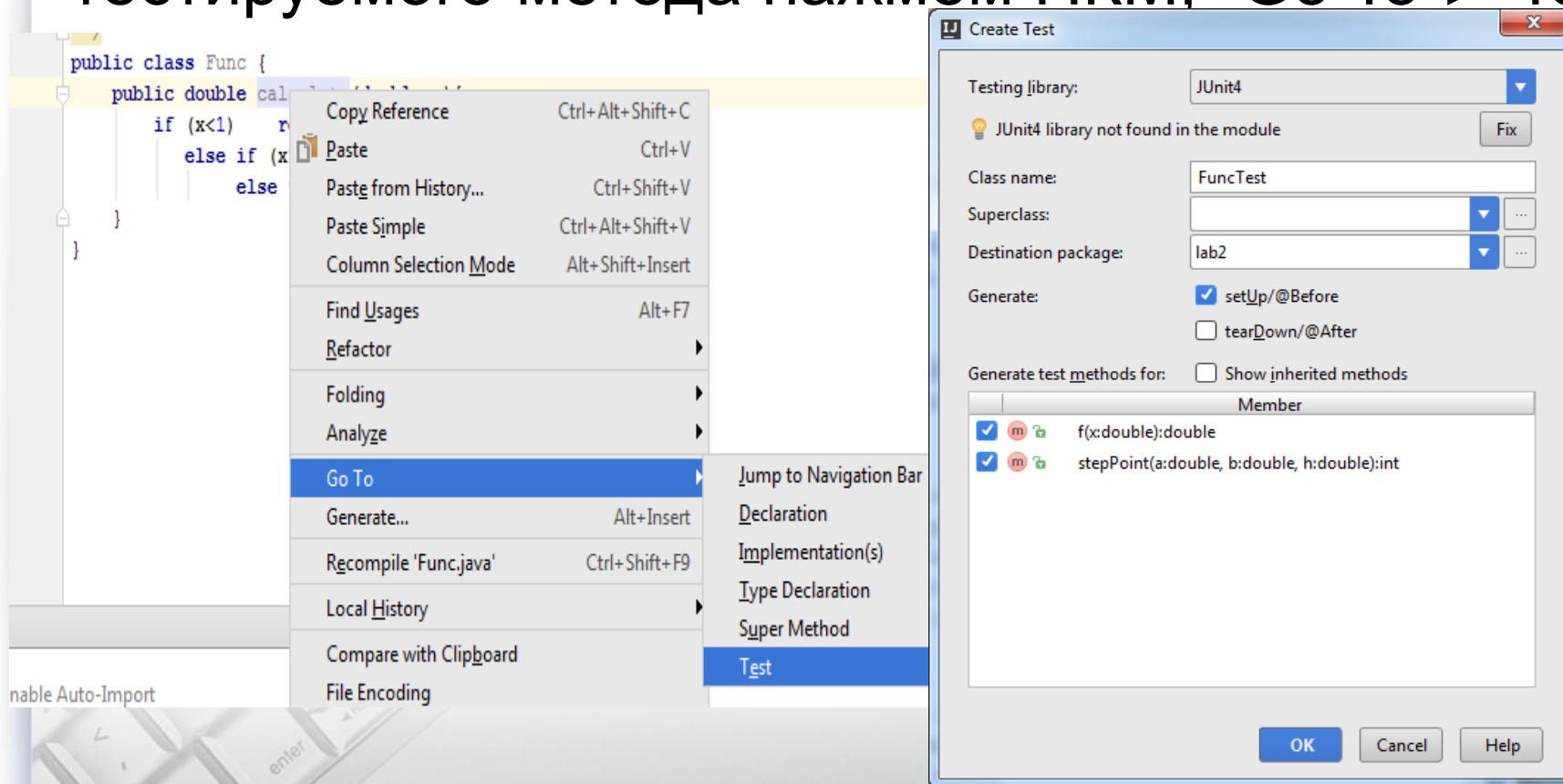
JUnit4 в IntelliJ IDEA

Создадим тестируемый класс.



JUnit4 в IntelliJ IDEA

Создадим тестовый класс, для этого на имени тестируемого метода нажмем ПКМ, “Go To ► Test



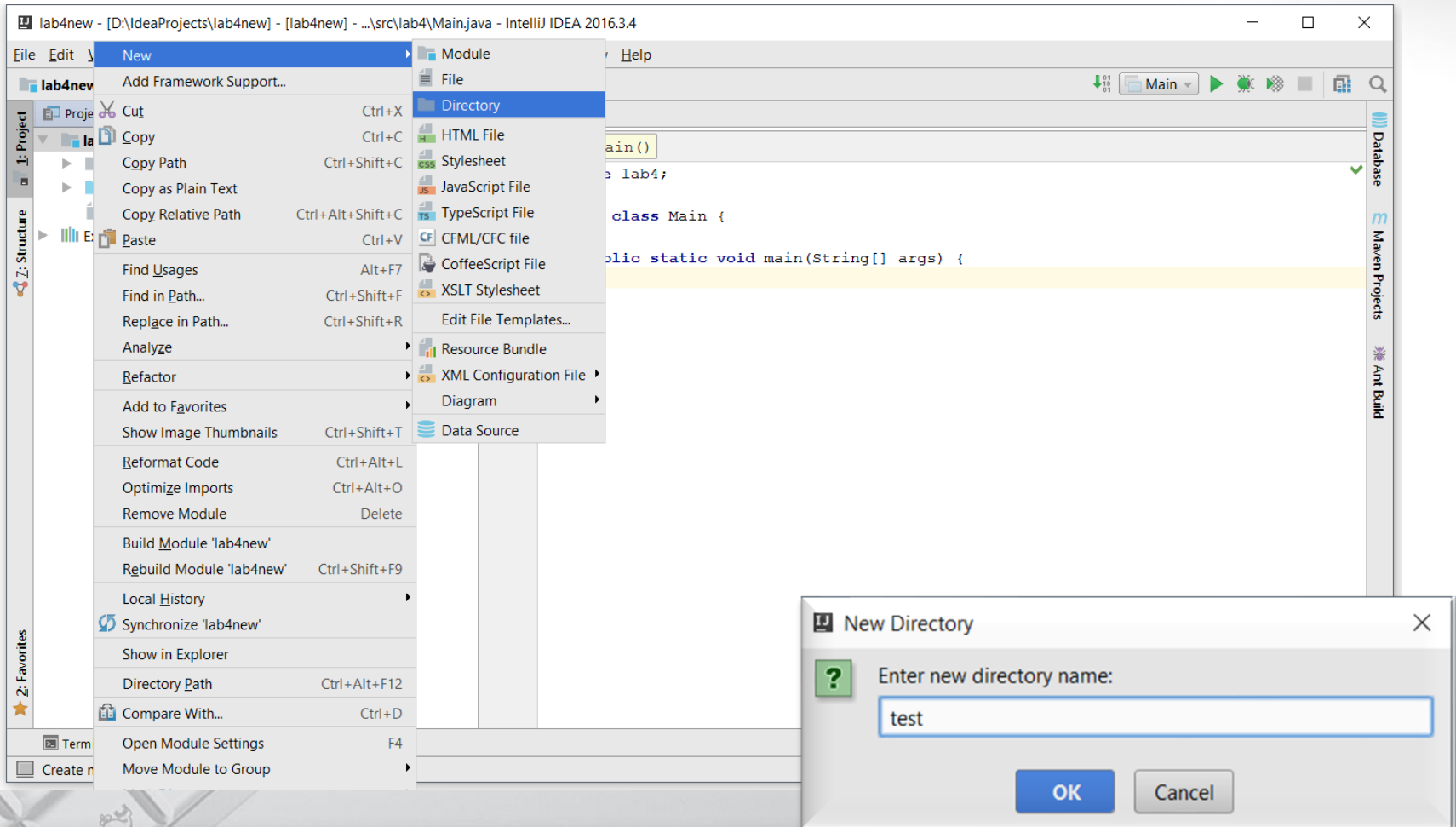
JUnit в IntelliJ IDEA

Примечание: если ваш тестируемый класс не лежит в пакете, необходимо создать пакет, например *Lab2*, потом перенести в него класс *Func*.



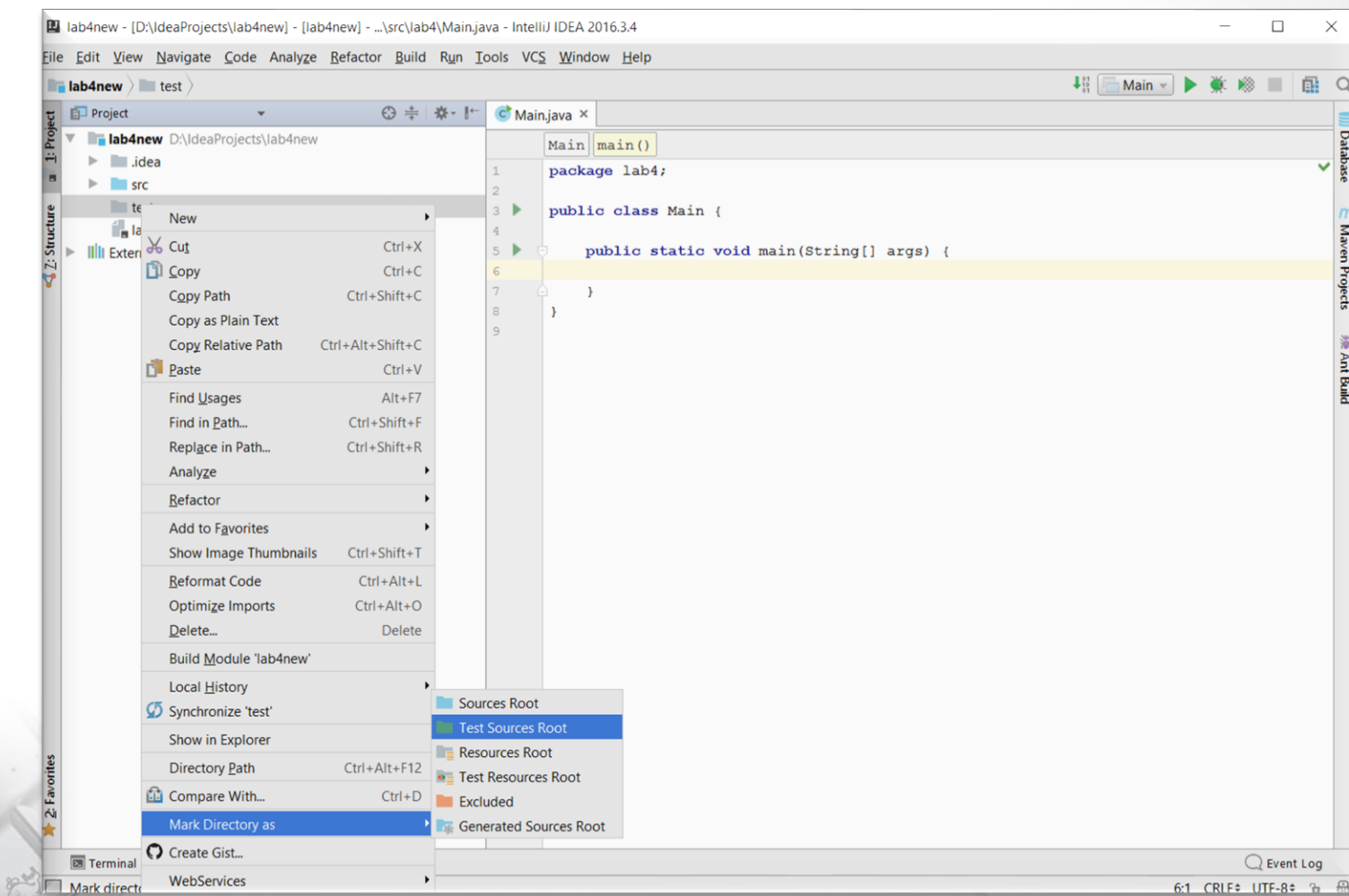
JUnit in IntelliJ IDEA

At first you have to create a directory for your tests



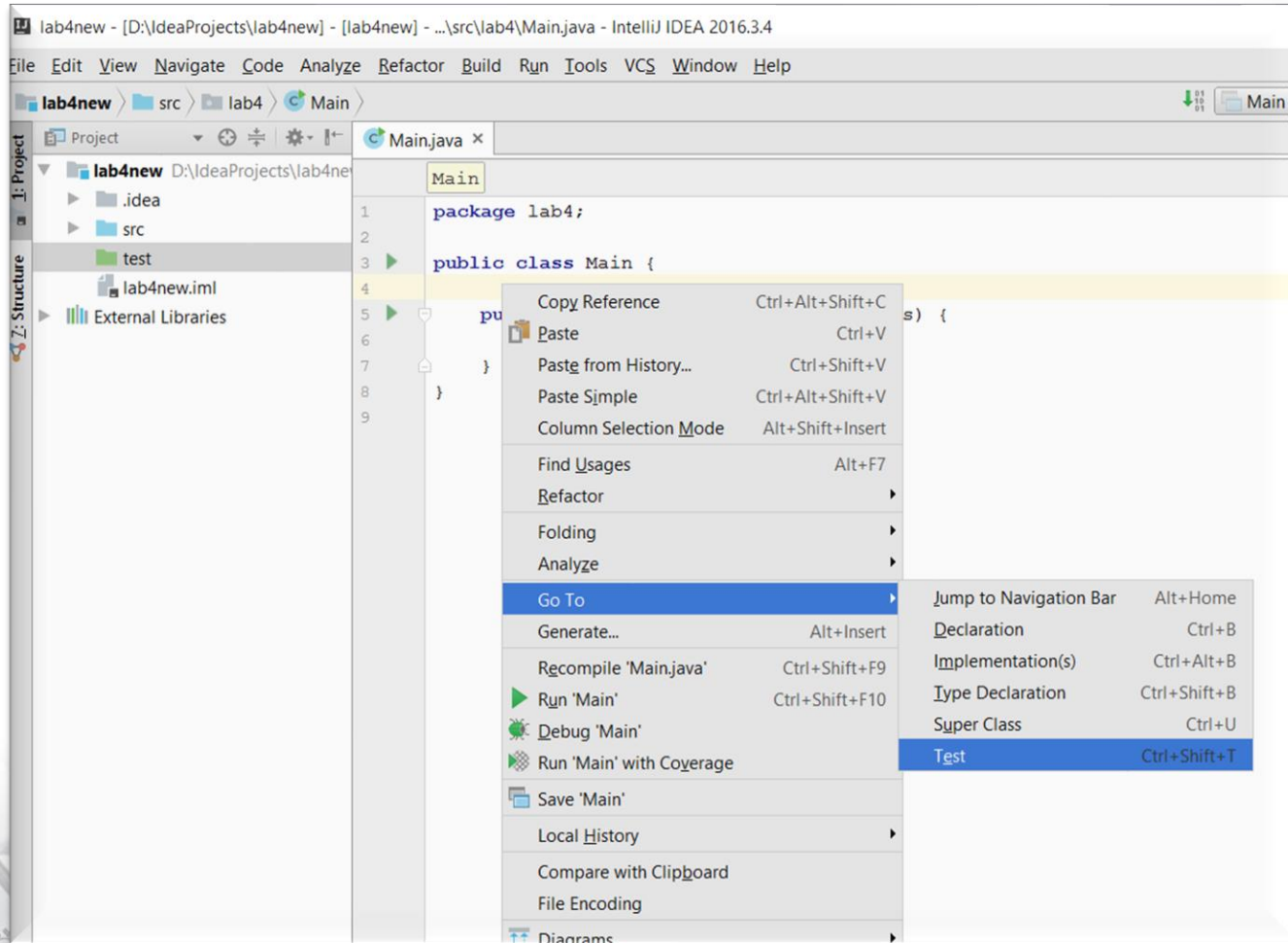
JUnit in IntelliJ IDEA

Then, mark it as Test Sources Root

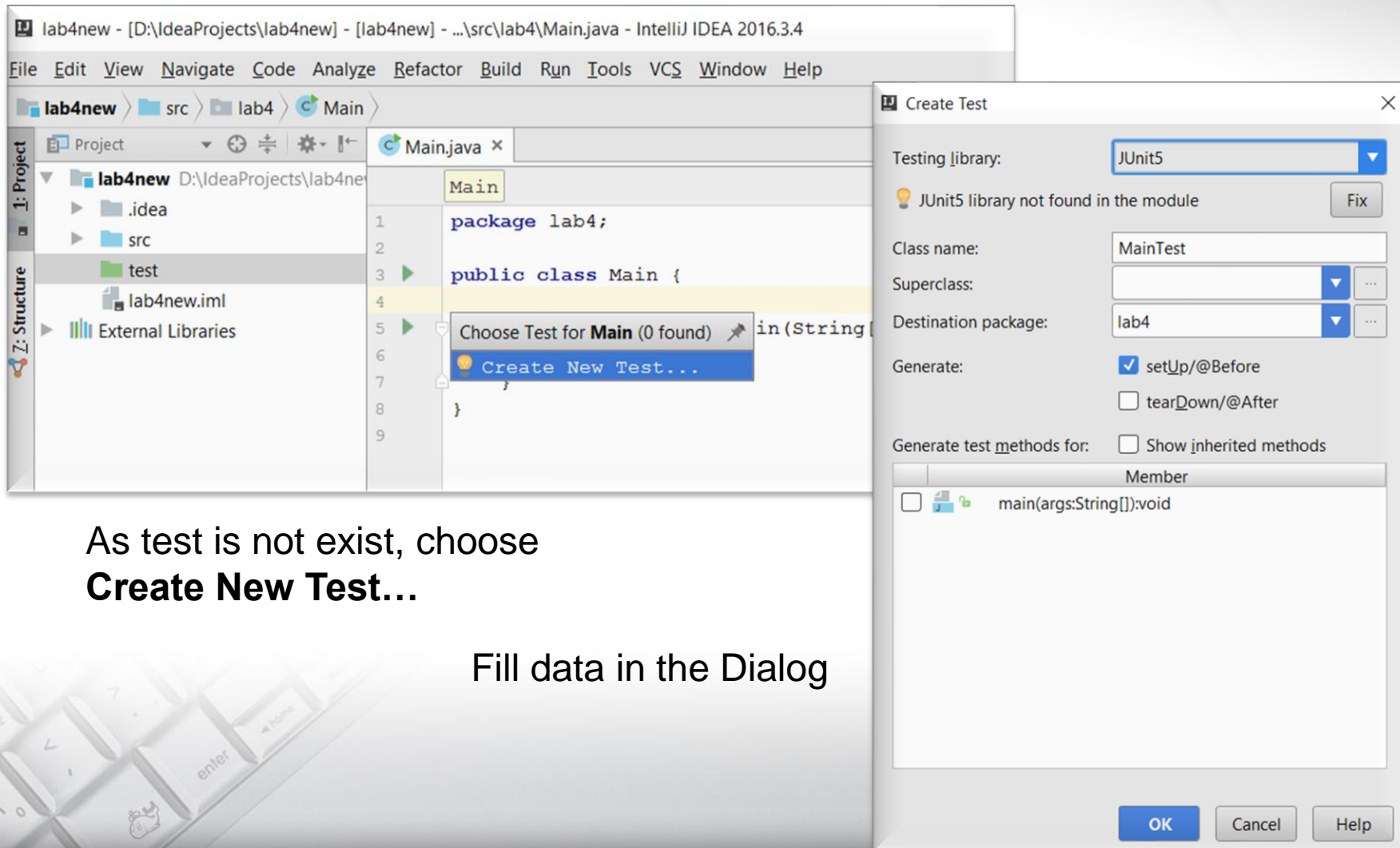


JUnit in IntelliJ IDEA

In your class choose “Go To ► Test



JUnit in IntelliJ IDEA



The screenshot shows the IntelliJ IDEA interface with a project named 'lab4new'. The 'Main.java' file is open, showing the following code:

```
1 package lab4;  
2  
3 public class Main {  
4  
5     in (String  
6  
7  
8  
9 }
```

A context menu is open over the 'Main' class, showing the option 'Create New Test...'. The 'Create Test' dialog is also open, with the following settings:

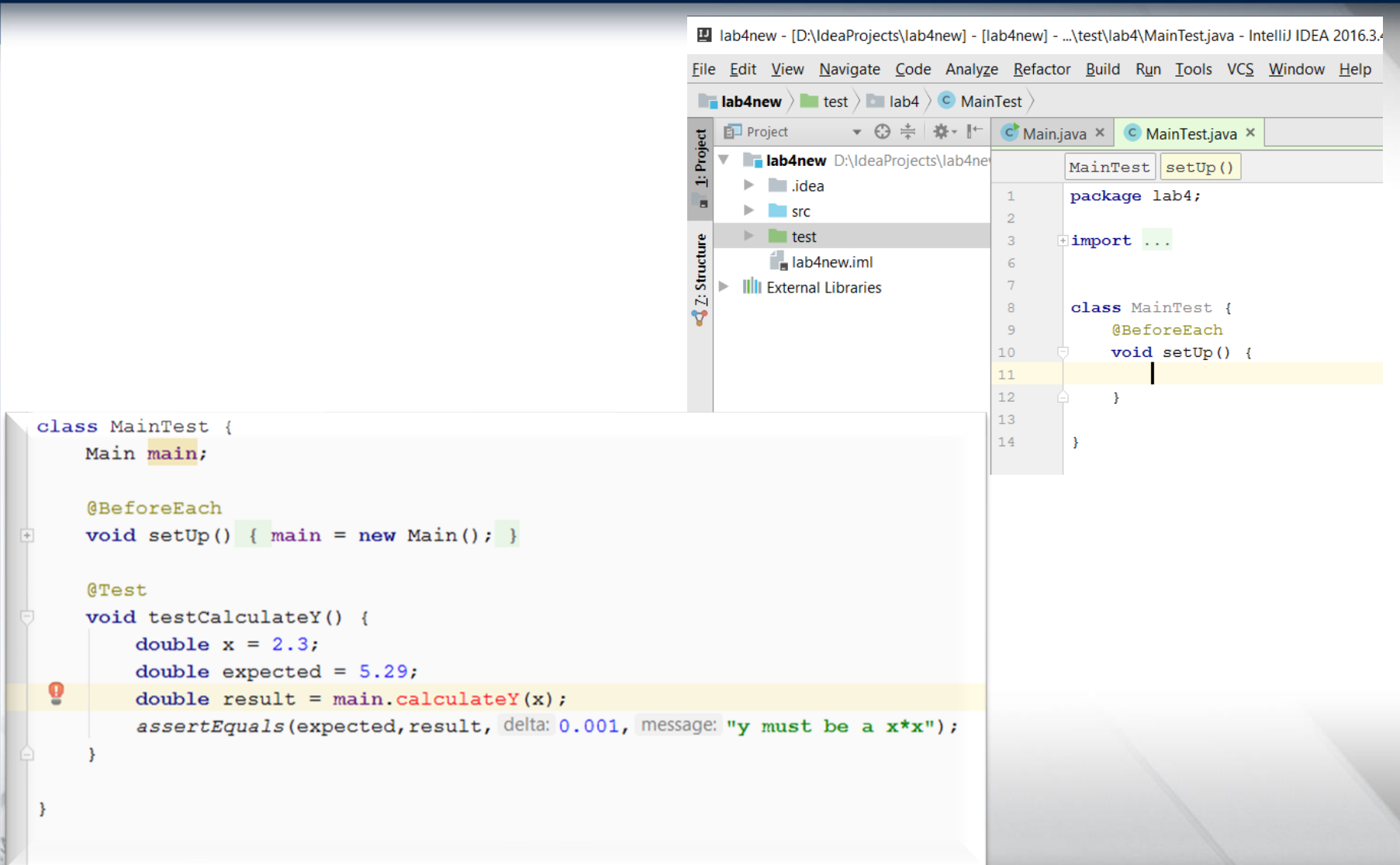
- Testing library: JUnit5
- JUnit5 library not found in the module (Fix button)
- Class name: MainTest
- Superclass: (empty)
- Destination package: lab4
- Generate: ☒ setUp/@Before, ☐ tearDown/@After
- Generate test methods for: ☐ Show inherited methods

The 'Member' list shows the method 'main(args:String[]):void'.

As test is not exist, choose **Create New Test...**

Fill data in the Dialog

JUnit in IntelliJ IDEA



lab4new - [D:\IdeaProjects\lab4new] - [lab4new] - ...test\lab4\MainTest.java - IntelliJ IDEA 2016.3.4

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

lab4new > test > lab4 > MainTest

Project D:\IdeaProjects\lab4new

- lab4new
 - .idea
 - src
 - test
 - lab4new.iml
 - External Libraries

MainTest setUp()

```
1 package lab4;
2
3 import ...
4
5
6
7 class MainTest {
8     @BeforeEach
9     void setUp() {
10
11
12     }
13
14 }
```

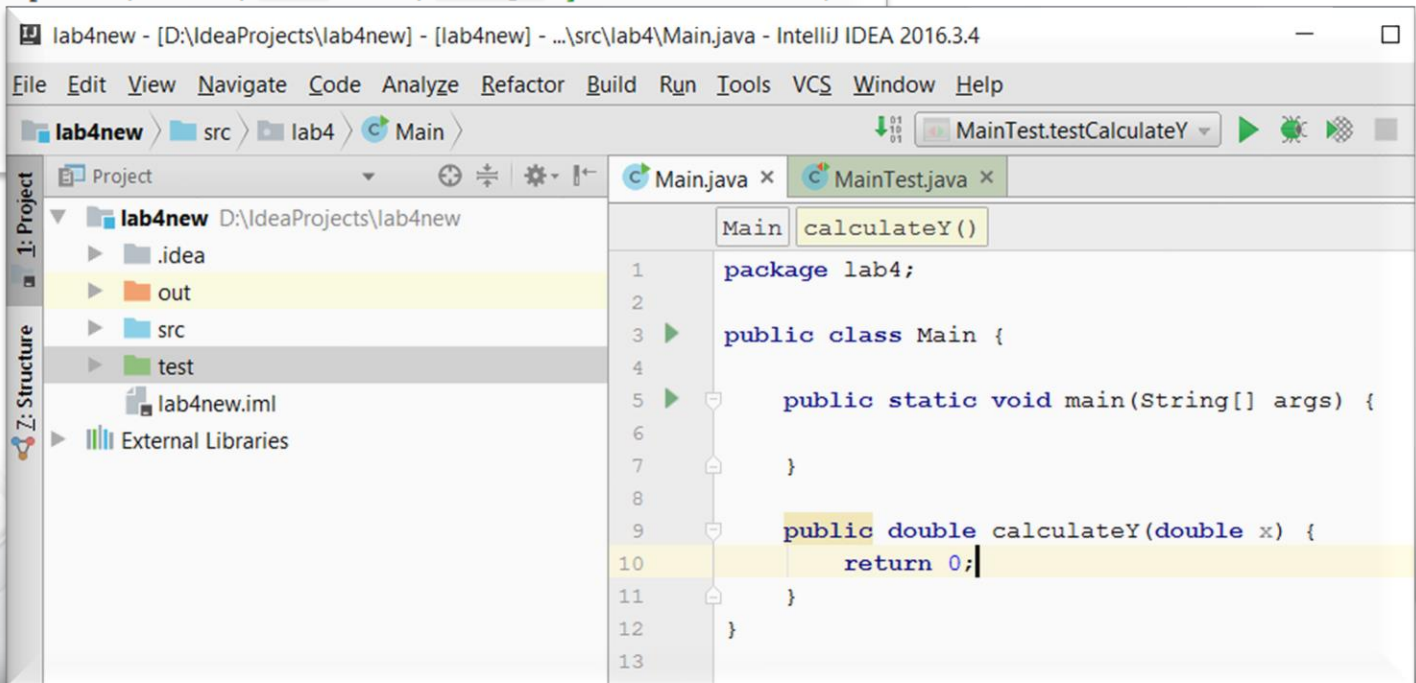
```
class MainTest {
    Main main;

    @BeforeEach
    void setUp() { main = new Main(); }

    @Test
    void testCalculateY() {
        double x = 2.3;
        double expected = 5.29;
        double result = main.calculateY(x);
        assertEquals(expected, result, delta: 0.001, message: "y must be a x*x");
    }
}
```

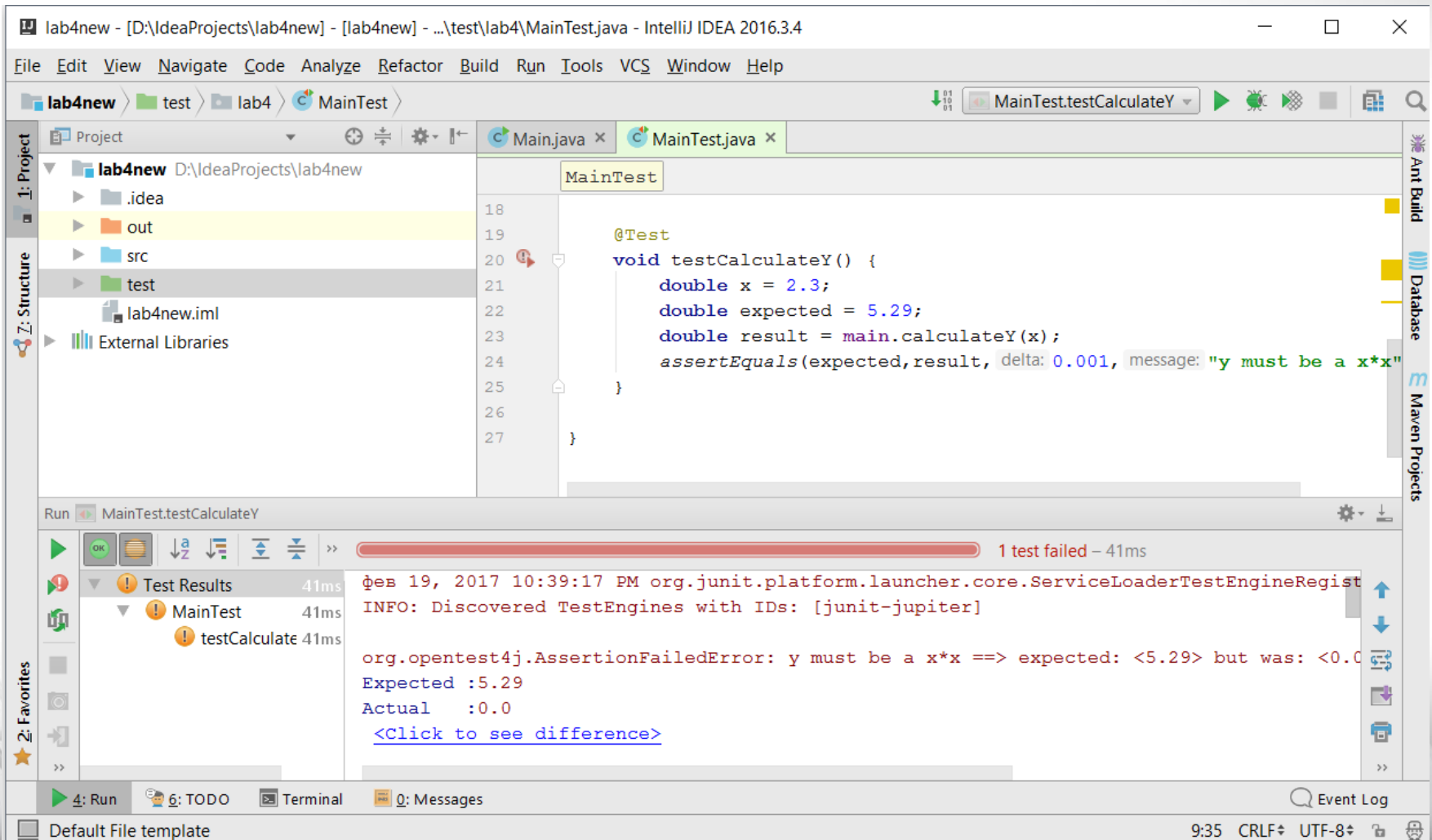

JUnit in IntelliJ IDEA

```
class MainTest {  
    Main main;  
  
    @BeforeEach  
    void setUp() { main = new Main(); }  
  
    @Test  
    void testCalculateY() {  
        double x = 2.3;  
        double expected = 5.29;  
        double result = main.calculateY(x);  
        assertEquals(expected, result, delta: 0.001, message: "y must be a x*x");  
    }  
}
```



JUnit in IntelliJ IDEA

Run test of the generated method. It fails



The screenshot shows the IntelliJ IDEA interface with a project named 'lab4new'. The 'test' directory is selected in the Project view. The 'MainTest.java' file is open, showing a JUnit test method 'testCalculateY()' that calls 'main.calculateY(x)' and asserts the result is 5.29. The Run toolbar shows the test failed. The Run Results window shows the test 'testCalculateY()' failed with an 'AssertionFailedError' message: 'y must be a x*x ==> expected: <5.29> but was: <0.0>'. The terminal window shows the output of the test run, including the assertion failure message.

lab4new - [D:\IdeaProjects\lab4new] - [lab4new] - ...test\lab4\MainTest.java - IntelliJ IDEA 2016.3.4

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

lab4new > test > lab4 > MainTest

Project

- lab4new D:\IdeaProjects\lab4new
 - .idea
 - out
 - src
 - test
 - lab4new.iml
 - External Libraries

Structure

Ant Build Database Maven Projects

MainTest

```
18
19
20 @Test
21 void testCalculateY() {
22     double x = 2.3;
23     double expected = 5.29;
24     double result = main.calculateY(x);
25     assertEquals(expected, result, delta: 0.001, message: "y must be a x*x")
26 }
27 }
```

Run MainTest.testCalculateY

1 test failed - 41ms

Test Results 41ms

- MainTest 41ms
 - testCalculate 41ms

org.opentest4j.AssertionFailedError: y must be a x*x ==> expected: <5.29> but was: <0.0>
Expected :5.29
Actual :0.0
[<Click to see difference>](#)

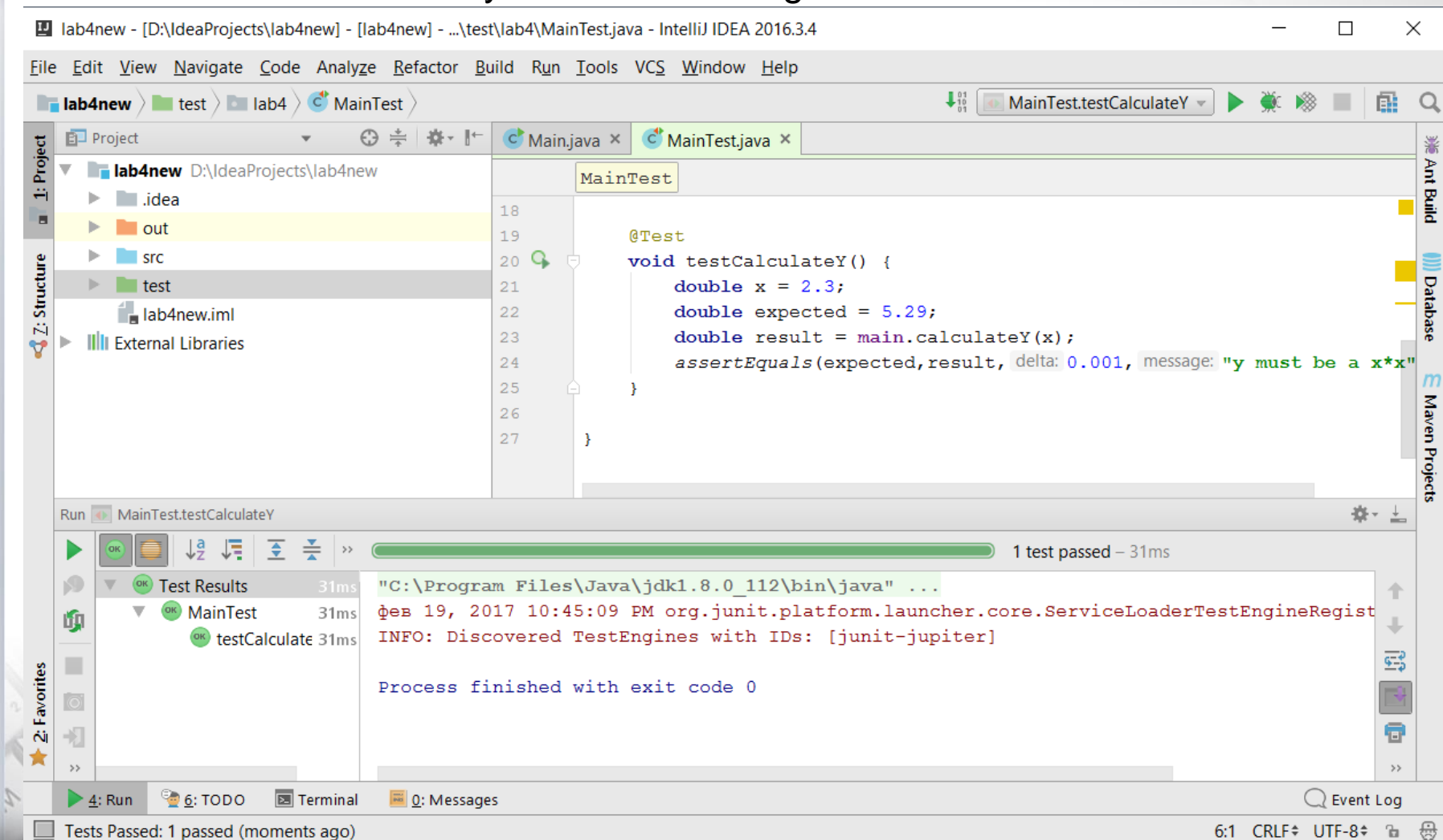
4: Run 6: TODO Terminal 0: Messages

Event Log

9:35 CRLF UTF-8

JUnit in IntelliJ IDEA

Write correct method body. Run test of the generated method. It should be OK



The screenshot shows the IntelliJ IDEA 2016.3.4 interface. The top toolbar has a green play button icon. The main editor displays the `MainTest.java` file with the following code:

```
18
19
20 @Test
21 void testCalculateY() {
22     double x = 2.3;
23     double expected = 5.29;
24     double result = main.calculateY(x);
25     assertEquals(expected, result, delta: 0.001, message: "y must be a x*x")
26 }
27
```

The left sidebar shows the project structure for `lab4new`, with the `test` directory selected. The bottom panel shows the Run configuration for `MainTest.testCalculateY`. The Run toolbar shows a green play button and a progress bar indicating "1 test passed - 31ms". The Test Results pane shows the test `testCalculate` passed in 31ms. The Run output pane shows the following log:

```
"C:\Program Files\Java\jdk1.8.0_112\bin\java" ...
фев 19, 2017 10:45:09 PM org.junit.platform.launcher.core.ServiceLoaderTestEngineRegist
INFO: Discovered TestEngines with IDs: [junit-jupiter]

Process finished with exit code 0
```

The bottom status bar shows "Tests Passed: 1 passed (moments ago)" and the encoding "UTF-8".

More Information

- <http://www.junit.org>
 - Download of JUnit
 - Lots of information on using Junit
- <https://junit.org/junit5/docs/current/user-guide/>