

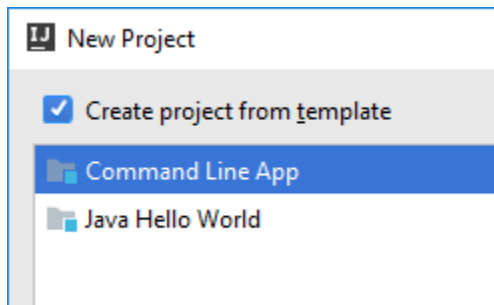
Exercises 1 – Setting up a new project with tests in IntelliJ

2017 © Edument AB

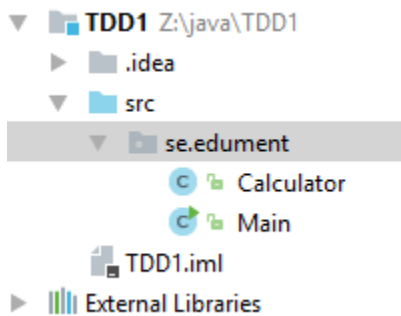
Exercises 1.1 – Hello World TDD

Let's create a simple application with tests.

1. Create a new **Command Line App** in IntelliJ



2. Create a **new class** named **Calculator** in the same folder as your Main class.



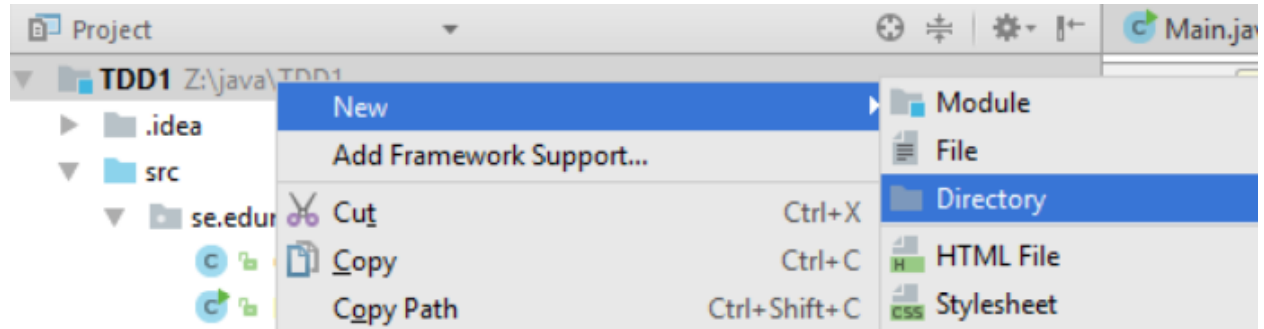
(the exact path is not important)

3. Add the following **add method** to this class:

```
public class Calculator {  
  
    public int add(int x, int y) {  
        return x+y;  
    }  
}
```

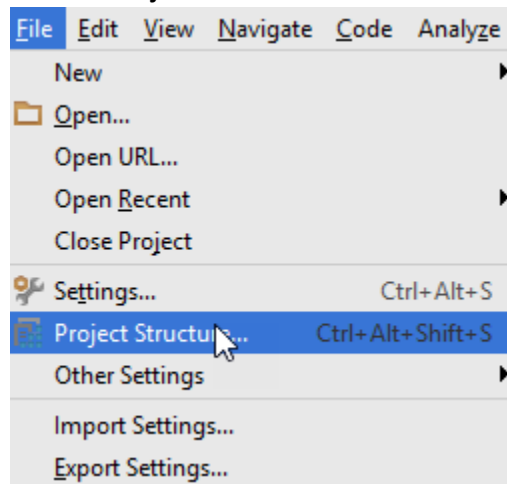
4. We now want to create a test class where we can write our tests in. We can either put the test class in the same folder as our Calculator or in a separate Tests directory. We choose to have it in a separate directory to better separate the two things.

Right-click on the project root and create a new **directory** named **tests**

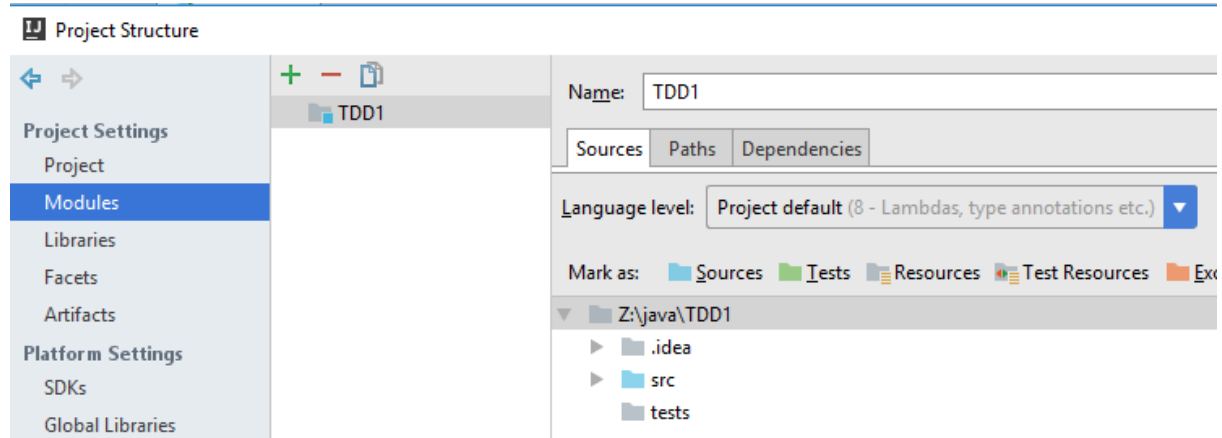


5. To tell IntelliJ that we want to put our tests in this specific **tests** folder, we need to do the following:

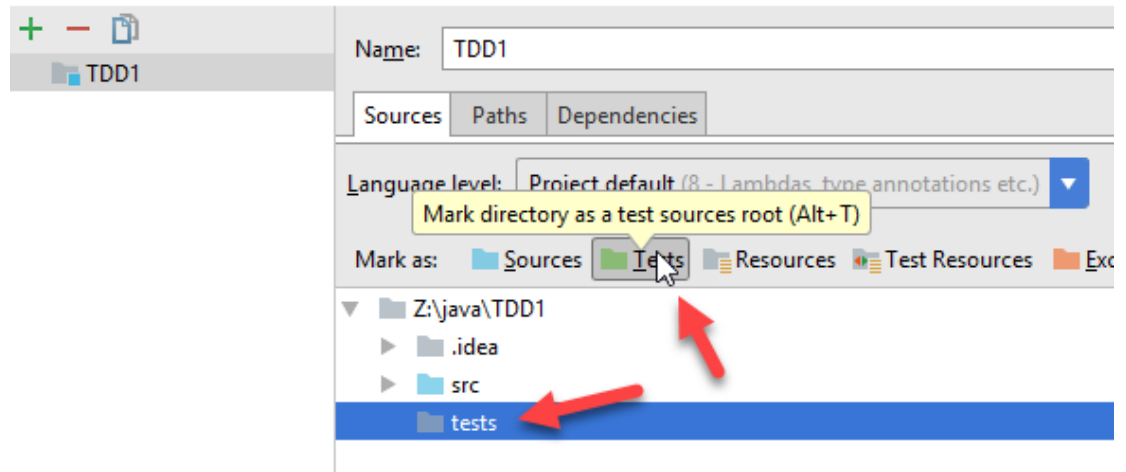
- a. Click on **Project Structure...**



- b. Then click on the **Modules** tab:

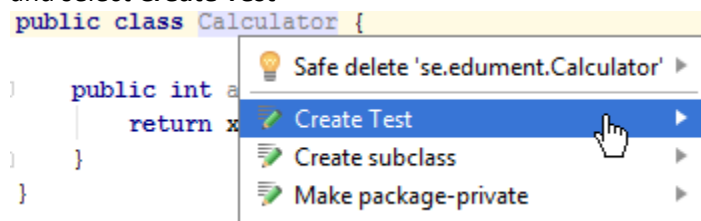


- c. Then click on the **tests folder** and then click on the **Tests icon** to mark the directory as your tests folder.

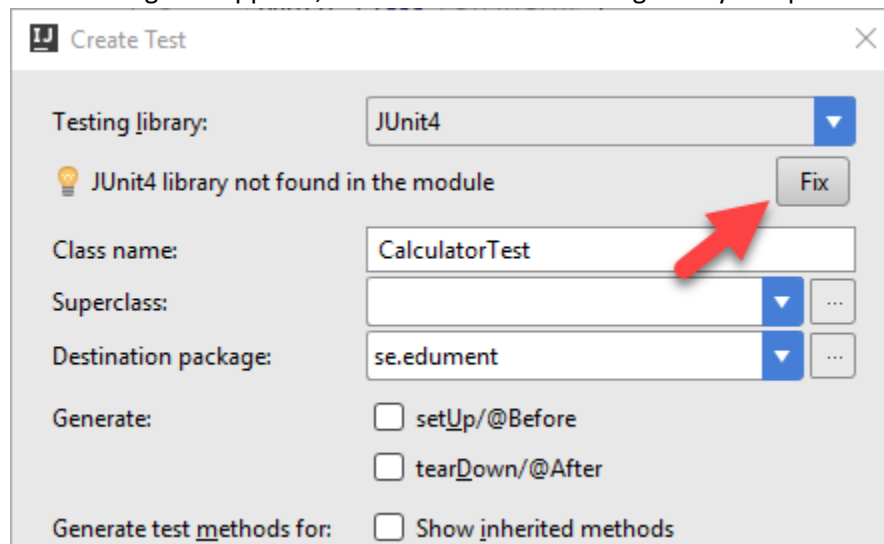


- d. Press OK to close the Project Structure dialog.

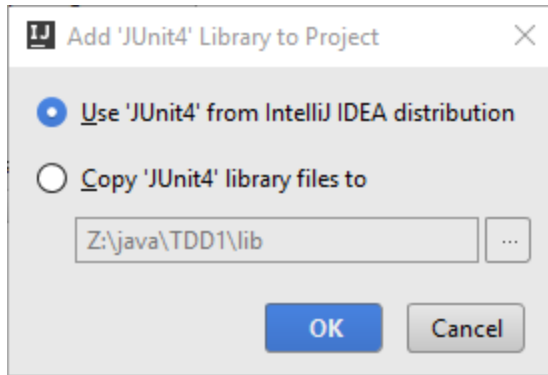
6. Go back to the **Calculator class** and put the cursor on the **class name** and then press **Alt+Enter** and select **Create Test**



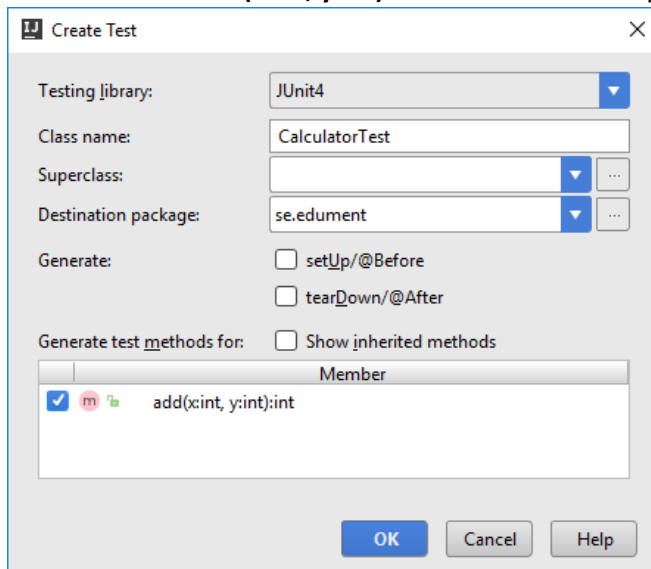
7. In the dialog that appears, select **JUnit4** as the testing library and press the fix **Fix** button



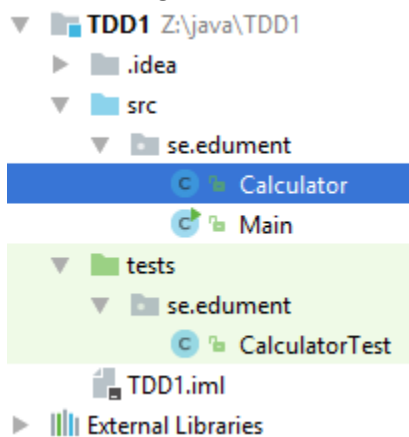
When pressing the Fix button the following dialog appear and select the **Use 'JUnit4' from IntelliJ** option.



Then select the **add(x:int, y:int):int** method and then press OK



8. A new testing class is now created in the Tests folder named **CalculatorTest**:



Notice the different colors on the folders! Blue for your source code and green for the tests. To read more about the definition for all the icons and colors visit:

<https://www.jetbrains.com/help/idea/2016.3/symbols.html>

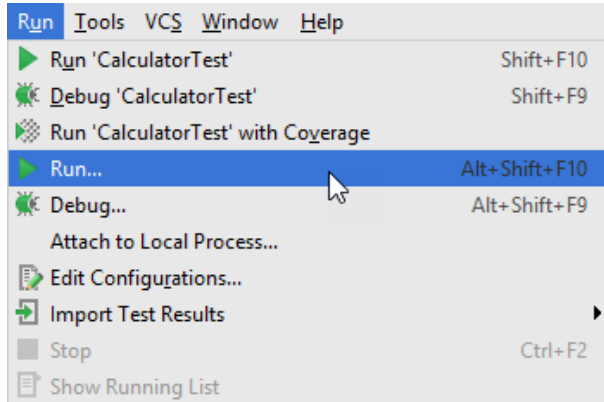
9. In our test-class remove any existing methods and add this new test:

```
@Test
public void whenAddingTwoPlusThreeShouldReturnFive() {
    Calculator sut = new Calculator();

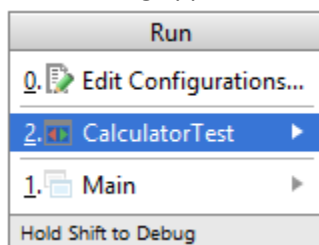
    int result = sut.add(2, 3);

    Assert.assertEquals( result, 5);
}
```

10. To run the test we can click on **Run -> Run**

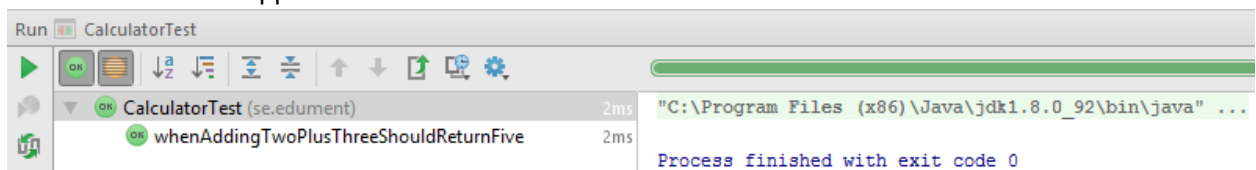


A new dialog appears:



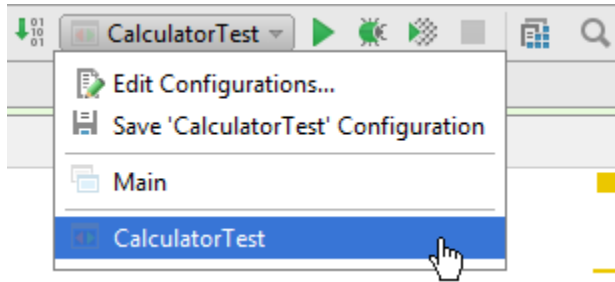
(Click on **CalculatorTest**)

A new test-window appears with the test results:



That's it! We have now executed our first unit test!

11. Using the configuration chooser in the upper right corner we can choose if we should run our **application** or **test** when we press the **run button**:



12. Read more about the test-runner window here:
<https://www.jetbrains.com/help/idea/2016.3/test-runner-tab.html>
and explore buttons and features available in the test-runner.

Exercises 1.2 – Adding a second

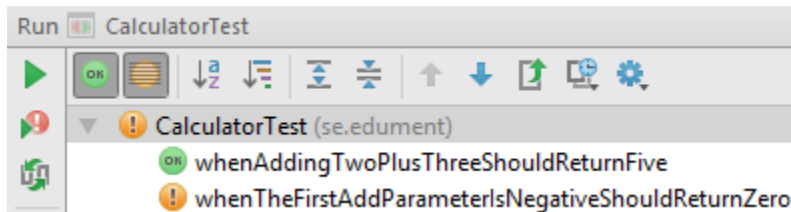
Let's try TDD and add more test to our class!

1. Continue on the previous exercise
2. This time we have a new requirement that says that we don't accept negative input numbers to our **add** method. If we provide negative numbers we should always return zero from our method.
3. Add a new test method named **whenTheFirstAddParameterIsNegativeShouldReturnZero**

Yes, it's a long name but long and descriptive names are important! Find your own naming convention and be consistent! It will make the test much easier to understand! Long names are good here!

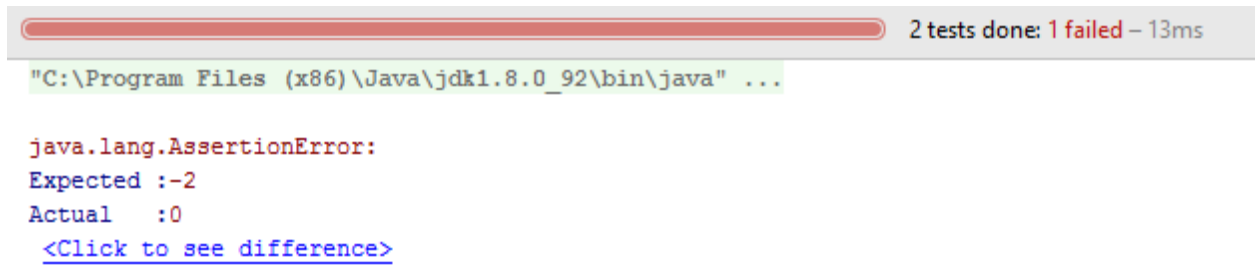
In the method copy the code from the previous test and this time:

- a. call **add(-5,3);**
 - b. In the **assert** verify that the result is **zero**.
4. Run the test and this time the new test should fail!



The test should fail because we haven't implemented the logic yet to handle negative numbers. We let the test drive what to do next and we should only implement enough code to make sure the test pass!

5. Before we make the test pass, we see the following error in the right part of the test-runner window:

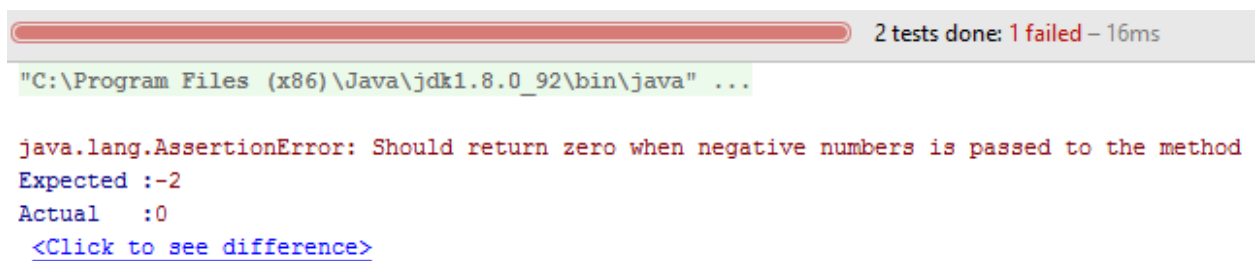


The screenshot shows a test runner window with a red progress bar at the top indicating '2 tests done: 1 failed - 13ms'. Below the progress bar, the command prompt shows the execution of 'C:\Program Files (x86)\Java\jdk1.8.0_92\bin\java'. The error message is 'java.lang.AssertionError: Expected :-2 Actual :0' with a link to 'Click to see difference'.

When troubleshooting testing errors in the future it can be hard to know why it failed or how to fix it. To fix this we can add a **message** that is displayed each time an assertion fails, like:

```
Assert.assertEquals( "Should return zero when negative numbers is  
passed to the method", result, 0);
```

Now when the test fails, we will see this result instead:



The screenshot shows a test runner window with a red progress bar at the top indicating '2 tests done: 1 failed - 16ms'. Below the progress bar, the command prompt shows the execution of 'C:\Program Files (x86)\Java\jdk1.8.0_92\bin\java'. The error message is 'java.lang.AssertionError: Should return zero when negative numbers is passed to the method' followed by 'Expected :-2' and 'Actual :0', with a link to 'Click to see difference'.

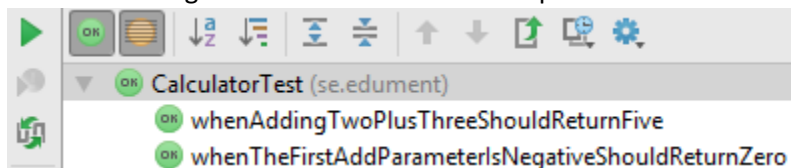
This message clearly will help future developers to understand why it failed! Add messages where it helps. But remember that you don't have to add it to every Assert statement.

6. Let's make this test by adding the necessary logic in the **add** method in the **Calculator** class to make it pass!

IMPORTANT! Only add enough code to make this specific test pass, don't think ahead!

Only check if the first parameter is zero, then return zero. Don't check the second parameter.

Run the tests again and make sure both test pass!



Exercises 1.3 – Refactoring

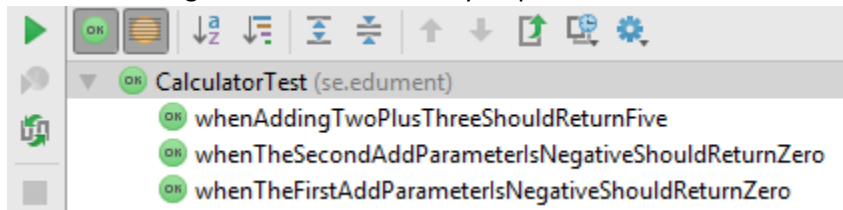
1. Write a new test named **whenTheSecondAddParameterIsNegativeShouldReturnZero**

In the test call the Add method with the values `sut.add(3, -6);`

Run the tests and make sure it fails!

2. Implement the logic to make this test pass **by adding a second if statement** that checks if the second parameter is zero and if so **return zero**.

Run the tests again and make sure they all pass!



3. Let's refactor!

After the test pass we have the opportunity to refactor our code to improve the code without changing the functionality.

This time you might find that you have two if-statements in the Add method, let's refactor it down to one if-statement.

Run the tests again to make sure everything still works!

4. A second refactor that we can do is in our tests. Here we notice that all of our tests contains this line of code:

```
Calculator sut = new Calculator();
```

5. Let's add a new method named **setup** in the test class with the **@Before** annotation:

```
@Before
public void setup() {
}
```

The **@Before** annotation cause that method to be executed before every **@Test** method. You must make sure **import org.junit.Before;** is added at the top of the class.

6. Remove the code `Calculator sut = new Calculator();` from every method and add this code instead in the class:

```
public class CalculatorTest {  
  
    private Calculator sut;  
  
    @Before  
    public void setup() {  
        sut = new Calculator();  
    }  
  
    //...
```

Run the tests again and make sure everything still works!

Exercises 1.4 – More tests

1. Try to add more tests and functionality to the code on your own. For example add support for multiply and Division.
2. Watch this short introduction video to unit testing <https://www.youtube.com/watch?v=BlD3644bIAo> and this one: <https://www.youtube.com/watch?v=xHk9yGZ1z3k&feature=youtu.be>
3. Read the Junit FAQ at: <http://junit.org/junit4/faq.html>
4. Junit contains many assertions and you can read about them here: <https://github.com/junit-team/junit4/wiki/Assertions>

Try some of them by adding a few additional tests to the previous exercise!

5. Many different naming conventions exists for JUnit and TDD, here's a few resources about that: <https://dzone.com/articles/7-popular-unit-test-naming>