Unit testing with JUnit

As you already know, a unit is a piece of code that performs a single task or a unit of work. In Java, in most cases, a unit corresponds to a class.

You can write and execute tests to check if the public methods of that unit work correctly.

Note that we use ***unit*** tests to test our application without its ***external*** dependencies such as databases, web services, and so on, which falls into the responsibility of ***integration*** tests.

In Java, methods may return values or change the internal state of objects. So to verify the correctness of any method you may compare the value returned by that method with the expected output or compare the internal state of an object modified by that method with the expected internal state.

If you write a bunch of tests covering all execution paths of such a method you can be sure that the method works correctly.

Doing unit testing manually is tedious and time-consuming, especially if it involves manual input from the user.

If you have ever tried it, you know it very well. That's why some frameworks provide convenient tools for automated unit testing. The most popular of them is JUnit.

Getting started

We are going to use JUnit 5 in this topic since it is the most recent version of the JUnit framework and it supports all features introduced by Java 1.8.

This means that JUnit 5 requires Java 8 (or higher) at runtime but also can be used to test code compiled with previous versions of the JDK.

To integrate the JUnit 5 framework into our project, you need to add the required dependencies. If you use **Gradle** as your project build tool, add the following lines to the build.gradle file:

```
dependencies {
   implementation 'org.junit.jupiter:junit-jupiter:5.7.1'
}
```

where 5.7.1 is the current most recent version of the framework. You always may check maven central for the most current version of the framework.

Also, to correctly use Gradle with JUnit 5, tell Gradle to use JUnitPlatform to run the tests:

```
test {
    useJUnitPlatform()
}
```

Otherwise, Gradle will not be able to see your tests and run them.

If you use **Maven**, add the following dependency to pom.xml:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.7.1</version>
</dependency>
```

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    public int divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Divisor cannot be zero!");
        }
        return a / b;
    }
}
```

Writing tests

Let's create a class in our project's test/java folder and name it CalculatorTests. You may do it manually or, if you are using IntelliJ IDEA, you may right-click on the class name and select **Generate...** and then **Test...** in the drop-down menu to let the IDE create the test class for you.

Inside the class, add a new method testAddition and annotate it with @Test.

This annotation tells the JUnit framework that the method is a unit test method. Note, that in JUnit5 you don't need the test class or any of the test methods to be public to work properly, so they may be package-private.

```
class CalculatorTests {

   @Test
   void testAddition() {
      Calculator calculator = new Calculator();
      int result = calculator.add(1, 2);

      assertEquals(3, result);
   }
}
```

Inside the test method, we declared an instance of our Calculator class and calculated a result of the execution of its add method with arguments 1 and 2.

After that we check if the expected result is equal to the actual result, using the assertEquals method. It compares the expected value (first argument) and the actual value (second argument) and throws AssertionFailedError if they are not equal.

It's a good idea to give your test classes and methods meaningful and descriptive names. If you don't like to use long and hard to read names, you may use @DisplayName annotation to declare custom names that will be shown in the test's output:

```
@Test
@DisplayName("Add 1 and 2, result should be 3")
void testAddition() {
   Calculator calculator = new Calculator();
   int result = calculator.add(1, 2);

   assertEquals(3, result);
}
```

Assertions

The Assertions class of JUnit framework has a lot of overloaded assertion methods that allow you to test different conditions. Here are some useful assertions:

assertEquals	tests if the arguments are equal
assertTrue	tests if the argument is true
assertFalse	tests if the argument is false

assertNull	tests if the argument is null
assertNotNull	tests if the argument is not null
assertThrows	tests if the argument throws a certain exception

All of them have overloaded versions which accept a message of the String type that will be displayed if the test fails.

Running tests
Now we have a few unit tests which we may run. You may run unit tests by opening the CalculatorTests file and click the green **Run** button in the gutter or by right-clicking on this file in the project view panel and selecting **Run 'CalculatorTests'** in the drop-down menu.

Another way to run the tests is to run them using Gradle. First, let's add the following lines to the build.gradle file so we can see more detailed output for executed tests:

```
test {

   testLogging {
      events "passed", "skipped", "failed"
   }
}
```

After that, run the following command in the terminal in the project root directory:

```
./gradlew test
```

Once the tests are executed, you will see this is the output:

```
> Task :test
CalculatorTests > testMultiplication() PASSED
CalculatorTests > Add 1 and 2, result should be 3 PASSED
CalculatorTests > testDivisionByZero() PASSED
CalculatorTests > testDivision() PASSED
BUILD SUCCESSFUL in 1s
```

The output shows the task that has been executed, the names of all tests, and the status of their execution.

Note that if a test method has an empty body it will be counted as PASSED. If you want to force your test to fail, you have to invoke the fail method inside it.

Test outcomes
Let's create a unit test for another method. But first, we are going to change our multiply method to introduce a bug in its implementation and make it return zero for any arguments:

```
public int multiply(int a, int b) {
   return 0;
}
```

Here is the code of the respective test:

```
@Test
void testMultiplication() {
   Calculator calculator = new Calculator();
   int result =  calculator.multiply(2, 3);

   assertEquals(6, result);
}
```

Run the tests again using the terminal:

```
./gradlew test
```

The new test fails as expected:

```
> Task :test FAILED
CalculatorTests > testDivision() PASSED
CalculatorTests > Add 1 and 2, result should be 3 PASSED
CalculatorTests > testMultiplication() FAILED
   org.opentest4j.AssertionFailedError at CalculatorTests.java:38
CalculatorTests > testDivisionByZero() PASSED
4 tests completed, 1 failed
```

The output shows that testMultiplication failed with AssertionFailedError at line 38 in CalculatorTests.java. Now let's go to the faulty method, fix the bug by making multiply to return a product of its arguments instead of zero, and re-run the tests:

```
> Task :test
CalculatorTests > testMultiplication() PASSED
CalculatorTests > Add 1 and 2, result should be 3 PASSED
CalculatorTests > testDivisionByZero() PASSED
CalculatorTests > testDivision() PASSED
```

Please note that gradle test runs each test only one time per change. If you execute this command again, there will be no output of test results. If you want to run all tests, including successfully passed and unchanged, use the following command: gradle cleanTest test.