JUnit Testing Part I: Setup With Simple Example

By **Dusan Odalovic**, dzone.com
March 25th, 2016

JUnit is one of available *Java* libraries we can use to test our application code. Let's get started by buliding simple project using *Maven*.

Generating Project With JUnit Support

Go generate simple *Java* artifact using *Maven, which* can be done quite easilly using command line.

```
mvn archetype:generate -DgroupId={GROUP_ID} -DartifactId=
{ARTIFACT_ID} -DarchetypeArtifactId=maven-archetype-
quickstart -DinteractiveMode=false
```

Just execute upper line in your console, giving it some **`{GROUP_ID}`** and **`{ARTIFACT_ID}`**. Make sure you haven *Maven* installed and on your path, so that you can run it as described. This is not related to JUnit testing, just give it any values, I will use **`com.mydomain`**as **`{GROUP_ID}`**, and **`junit-demo`**as **`{ARTIFACT_ID}`**. That should generate new java project ready to demo *JUnit* tests. You can import generated application into IDE of your choice.

To confirm we have all we need, open generated **`pom.xml`**file in the root of your project and verify there's dependency to *JUnit* library in there (I'm using 4.12 version which is the latest stable version of *JUnit* at the time of writing this article).

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Once we confirm there's given dependency, we will proceed with some theory alongside example code. We will create simple calculator class which can do some basic operations as adding, subtracting, multiplying, dividing numbers. We want to create *JUnit* tests to make sure the calculator class is working as expected and has production ready quality.

Creating System Under Test (SUT) and Test case

JUnit testing takes care of testing our application at low level, meaning we want to make sure our classe methods are tested against different conditions, and they provide expected results in such a cases. Our methods we want to test can, during application runtime, receive different values for input parameters. We want to make sure our method is tested against these. Of course - not against all possible values, since that's not possible, but against some meaningful values, as well as some edge cases where our methods can receive, let's say *null* as parameter, or some other value that could potentially crush our application. Let's start writing **`com.mydomain.Calculator.java`**and *JUnit* tests for it, **`com.mydomain.CalculatorTest.java`**. Test classes are also known as ***Test cases.***

It's a common practice to name test class after class we're testing (sometimes referred to as ***System Under Test - SUT***), by appending ***Test*** suffix to it. In our case, our *SUT* is **`Calculator.java`***,* so our test class should be **`CalculatorTest`**.java*.* Also, common practice is that test class should reside in same package as *SUT* one (in our case, both are in **`com.mydomain`**package, but test class is in **`src/test/java`**tree, and *SUT* is in **`src/main/java`***.* These are maven defaults which state that application code should go to X*,* whereas test code should be inside **`src/test`**packages. Our *Calculator* class should look like:

```
package com.mydomain;

public class Calculator {

    public long add(long first, long second) {
        return first + second;
    }

    public long subtract(long first, long second) {
        return first - second;
    }

    public long multiply(long first, long second) {
        return first * second;
    }

    public long divide(long first, long second) {
        return first / second;
    }
}
```

Without writing *JUnit* tests for given piece of code, we can only *HOPE* our code will work as expected. We should be able to exercise this piece of code before deploying it to production so that we can be confedent code is working as expected.

Our application should have layout such as given below (target directory is incomplete here, but that's irrelevant):

```
├── pom.xml
├── src
│   ├── main
│   │   └── java
│   │       └── com
│   │           └── mydomain
│   │               ├── App.java
│   │               └── Calculator.java
│   └── test
│       └── java
│           └── com
│               └── mydomain
│                   ├── AppTest.java
│                   └── CalculatorTest.java
└── target
```

Let's now write some *JUnit* tests in order to exercise our application. Our **`CalculatorTest`**class could look like:

```
package com.mydomain;

import org.junit.Test;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

public class CalculatorTest {
    @Test
    public void twoAndThreeIsFive() throws Exception {
        final long result = new Calculator().add(2, 3);
        assertThat(result, is(5L));
    }

    @Test
    public void twoAndZeroIsTwo() throws Exception {
        final long result = new Calculator().add(2, 0);
        assertThat(result, is(2L));
    }

    @Test
    public void twoAndMinusTwoIsZero() throws Exception {
        final long result = new Calculator().add(2, -2);
        assertThat(result, is(0L));
    }

    @Test
    public void threeMinusTwoIsOne() throws Exception {
        final long result = new Calculator().subtract(3, 2);
        assertThat(result, is(1L));
    }

    @Test
    public void threeMinusThreeIsZero() throws Exception {
        final long result = new Calculator().subtract(3, 3);
        assertThat(result, is(0L));
    }

    @Test
    public void threeMinusMinusThreeIsSix() throws Exception {
        final long result = new Calculator().subtract(3, -3);
        assertThat(result, is(6L));
    }

    @Test
    public void threeXThreeIsNine() throws Exception {
        final long result = new Calculator().multiply(3, 3);
        assertThat(result, is(9L));
    }

    @Test
    public void threeXZeroIsZero() throws Exception {
        final long result = new Calculator().multiply(3, 0);
        assertThat(result, is(0L));
    }

    @Test
    public void threeXMinusThreeIsMinusNine() throws
Exception {
        final long result = new Calculator().multiply(3, -3);
        assertThat(result, is(-9L));
    }

}
```

Idea is to exercise our public methods present in **`Calculator`**class. We should give it various inputs, and *assert* that results are as expected. In order to create *JUnit* test, we should just mark our method with **`@Test`***JUnit* annotation (**`org.junit.Test`***)*. When we execute this set of tests we wrote, *JUnit* will executed all methods marked with **`@Test`** annotation.

In each test method we have two phases:

1. Call *SUT* method for particular input (parameters)
2. Assert that result matches our expectancy (like we assert that 3 x 3 should be 9)

Running Tests and Verifying Results

The easy way to run all the tests is to package our application, using *Maven* command in our comand line:

```
mvn clean package
```

In console output, after our application code was compiled, we can see that *JUnit* tests are executed:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.mydomain.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.013 sec - in com.mydomain.AppTest
Running com.mydomain.CalculatorTest
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.004 sec - in com.mydomain.CalculatorTest
```

```
Results :
Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
```

We can see that there's total of 9 tests being run in **`CalculatorTest`**and that there were no failures (Failures: 0 in log output). If some tests were failing, we would be presented with an error message, and *Maven* build would have failed. Let's change implementation of **`threeMinusMinusThreeIsSix`**method to wrongly assert that 3 multiplied by -3 should be 2.

```
@Test
public void threeMinusMinusThreeIsSix() throws Exception {
    final long result = new Calculator().subtract(3, -3);
    assertThat(result, is(2L));
}
```

When repeating upper maven command to package app, we will get output:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.mydomain.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.013 sec - in com.mydomain.AppTest
Running com.mydomain.CalculatorTest
Tests run: 9, Failures: 1, Errors: 0, Skipped: 0, Time
elapsed: 0.027 sec <<< FAILURE! - in
com.mydomain.CalculatorTest
threeMinusMinusThreeIsSix(com.mydomain.CalculatorTest) Time
elapsed: 0.005 sec <<< FAILURE!
java.lang.AssertionError:
Expected: is <2L>
 but: was <6L>

....

Tests run: 10, Failures: 1, Errors: 0, Skipped: 0

[INFO]
--------------------------------------------------------------
----------
[INFO] BUILD FAILURE
```

We clearly see that build failed, and what's causing build to fail. It's quite handy we can quickly execute tests and easilly see if something is failing and why. Test can fail if *SUT* has issues, or *JUnit* tests are not properly written (such as we did above - just for demo purpose).

Bare in mind that *JUnit* test should have quality standards same as application code, so make sure you apply all the good practices you already apply to application code. Also, *JUnit* tests should be independent of each other, and must be able to run in any order given. They should also be quick to execute and easy to verify results.

Key Takeaways

- *JUnit* is library for automated unit testing
- Idea is to excercise the application code for various cases before application goes live, thus preventing issues that might happen during application runtime.
- Each *JUnit* test should be independent of other tests, and should be able to run fast, so that we can execute them frequently.

The next post will be related to extending *JUnit* test using *Mockito* library.

Hope you liked the post and feel free to download application sources at GitHub.

Stay tuned and please - don't forget to subscribe in case you're eager to find out what's coming next in upcoming posts.