

## Lab: Introduction to Unit Testing (Java)

This guide gives an introduction into unit testing with the JUnit framework using JUnit 5. Pay attention that JUnit 5 requires Java 8 version of the language, or higher.

In the following sections different aspects of unit testing will be explained in detail:

1. [Configuration for using JUnit 5 for Maven Project](#)
2. [Creating JUnit tests](#)
3. [Creating a test case](#)
4. [Running JUnit tests](#)
5. [Preparing objects for tests](#)

For more information about JUnit5 see:

<https://www.vogella.com/tutorials/JUnit/article.html>

<https://junit.org/junit5/docs/current/user-guide/>

### About the project under testing:

In this lab, we will work with a Maven project which represents a financial calculation system that uses different currencies. Adding and subtracting an amount in the same currency (\$ for example) is implemented in an obvious way.

Let's start by defining the class `Money`, which stores the amount of money as a number, and the currency by representing three letters according to the accepted standard (USD, NIS, GBP, etc.).

The method `add()` of the class `Money` is activated when you want to add two amounts of money of the same currency. This method returns a new object of type `Money` (with the appropriate amount of the considered currency).

## 1. Configuration for using JUnit 5 for Maven Project

To use JUnit 5 you have to make the corresponding libraries available for your test code. If you are using Maven as build system, you need to configure Maven dependencies for JUnit 5. Namely, you need to adjust your pom file, similar to the following:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
</properties>
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Once you have done this, you can start using JUnit5 in your Maven project for writing unit tests.

## 2. Creating JUnit tests

You can write the JUnit tests manually, but Eclipse supports the creation of JUnit tests via wizards.

You can right-click on your class in the *Package Explorer* and select **New -> JUnit Test Case**. Or, you can also use the JUnit wizards available under **File -> New -> Other... -> Java -> JUnit -> JUnit Test Case**.

## 3. Creating a test case

Each test case will be implemented in a separate method. A test method has to be declared by tag `@Test` and annotated with description of the corresponding test case.

Each test method will consist of three parts:

- (1) Creating the objects that will be used for testing;
- (2) Call the method under testing;
- (3) Verifying correctness of the operation (i.e. comparing the results to expectations).

```
public class MoneyTest{
    // ...
    // checking functionality: sum of two positive values
    // input data: 12 CHF + 14CHF
    // expected result: 26CHF
    @Test
    public void testAdd_PositiveValues() {
        Money m12CHF= new Money(12, "CHF");           // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26,"CHF");
        Money result= (Money)m12CHF.add(m14CHF);       // (2)
        assertTrue(expected.equals(result));           // (3)
    }
}
```

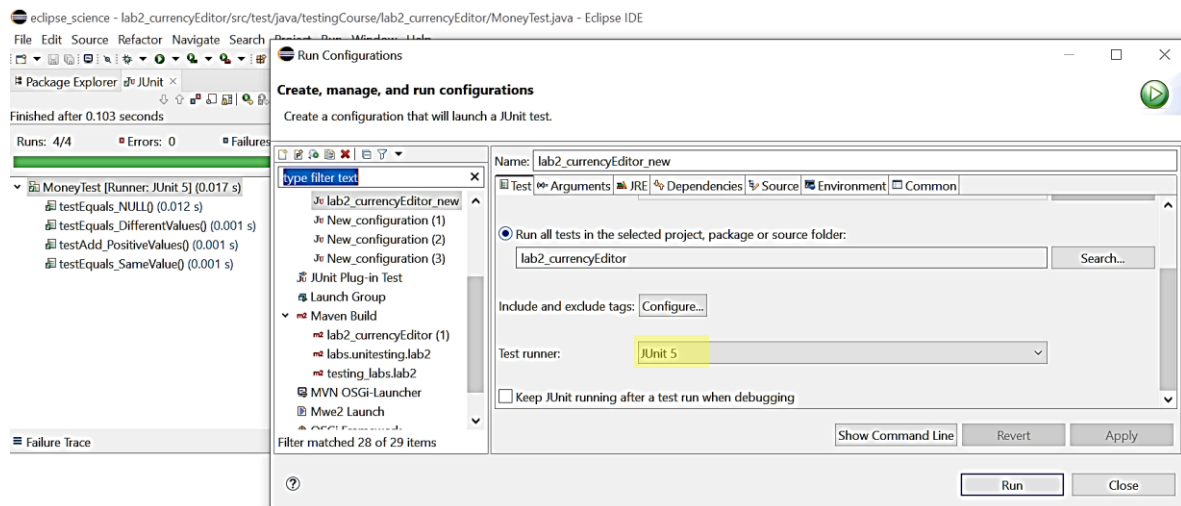
In the example presented above, first we create two objects of the type `Money` to be summed and an object `expected` to save an expected result of this test. Then we call the method under testing `add()` and save its result in the variable `result`. Finally we compare `expected` and real results of the calculation using the standard method `assertTrue()`.




Running the test, `assert` declaration is checked, and if a boolean expression inside `assertTrue()` is false, the test fails, otherwise it passes.

## 4. Running JUnit tests

Now we can run the test. To do so, select the test class, right-click on it and select **Run as JUnit Test**. This starts JUnit and executes all test methods in this class.

Alternatively, one can choose **Run->Run configuration** and create a new configuration of type JUnit. Pay attention that we use JUnit 5 and the corresponding information has to be chosen as shown below.



As you can see on the screen above, test results are represented in JUnit view. Each test case is marked by green , that means ‘test passed’, or by blue  that means ‘test failed’. Pay attention that red  is not the correct result of the test, it means exception in its execution, so the test was built incorrectly.

## 5. Preparing objects for tests

It can be useful to build a collection of objects that we need for testing, and use it in several test cases. In this case, it will be more efficient, if we define the collection of objects in advance before starting the tests. For this purpose we can create the method `setUp()` as in the example below with the required objects and rewrite the test case from section 2 using these objects.

```
public class MoneyTest{
...
    private Money f12CHF;
    private Money f14CHF;

    @BeforeEach
    protected void setUp() throws Exception {
        m12CHF= new Money(12,"CHF");
        m14CHF= new Money(14,"CHF");
    }
...
}
```

In this case any changing of defined objects inside test methods will be actual only for this test case.

## 6. Further explanations about project under test

In this lab you have to create tests for checking `MoneyBag` class.

```
public class MoneyBag implements IMoney {
    private Vector fMonies= new Vector(5);
    ...
    MoneyBag(Money m1, Money m2) {
        appendMoney(m1);
    }
}
```

```
        appendMoney(m2);  
    }  
}
```

The object of this class contains sums in more than one currency. For example, an object <5USD, 10NIS> contains 5 dollars and 10 shekels and can be created as follows:

```
Money m5USD= new Money(5, "USD");  
Money m10NIS= new Money(10, "NIS");  
MoneyBag mb= new MoneyBag(m5USD, m10NIS);
```

All operations are possible only with objects of the same currency. In our example, <5USD, 10NIS> + 5NIS = <5USD, 15NIS>.

Both classes `Money` and `MoneyBag` implement the interface `IMoney` for one currency and several currencies, correspondingly. Pay attention that in the following example: <5USD, 10NIS> - 10NIS = 5USD, the code is written in a way that in a moment when we remove one of two currencies from `MoneyBag` object, the object will be transferred to the corresponding `Money` object.