# Coding Bootcamp: Unit Testing with JUnit

## Learning objectives

- What is Unit Testing
- What is considered a *Unit* in Java
- How the `JUnit` framework provides Unit Testing support in Java

## Motivating example

- **MyMathSimple** : a class with a simple single method located in the package (main)

```
package main.java;

public class MySimpleMath {
    /**
     * A simple method that takes and input and returns
     * "positive" or "negative" depending on the input number
     */
    public String checkSign(int number) {
        if(number >= 0 ) {
            return "positive";
        } else {
            return "negative";
        }
    }

}
```

## Motivating example (2)

- In order to Test the functionality of our class and methods we used to add some code in a **main** method and compare the output of the execution against the expected one

```
    public static void main(String args[]) {
        MySimpleMath sm = new MySimpleMath();
        // Check a positive
        System.out.println("Input 10: " + sm.checkSign(10));
        // Check a negative
        System.out.println("Input -2: " + sm.checkSign(-2));
        // Check a positive
        System.out.println("Input 0: " + sm.checkSign(0));

    }
```

The output of the execution should be

```
    Input 10: positive
    Input -2: negative
    Input 0: positive
```

# Motivating example (3)

- Are there any other ways to test the execution of the code?
- Use messages ( `System.out.println()` ) in places of interest
- Use the debugger
- Use **Assertions** to verify the value of variables in specific places in your code

# Motivating example (4)

- Do you notice any drawbacks with the aforementioned techniques?
- Testing many classes and methods may produce very long main methods
- Easy to miss some cases in a complex method
- The **main** method is usually part of the **production code**, and should not contain test code
- Error-prone to system-wide changes
- Is there a better solution to test our code?

# *Testing* definition

*Dynamic verification that the program has the expected behavior in a suitably selected final set of test cases*

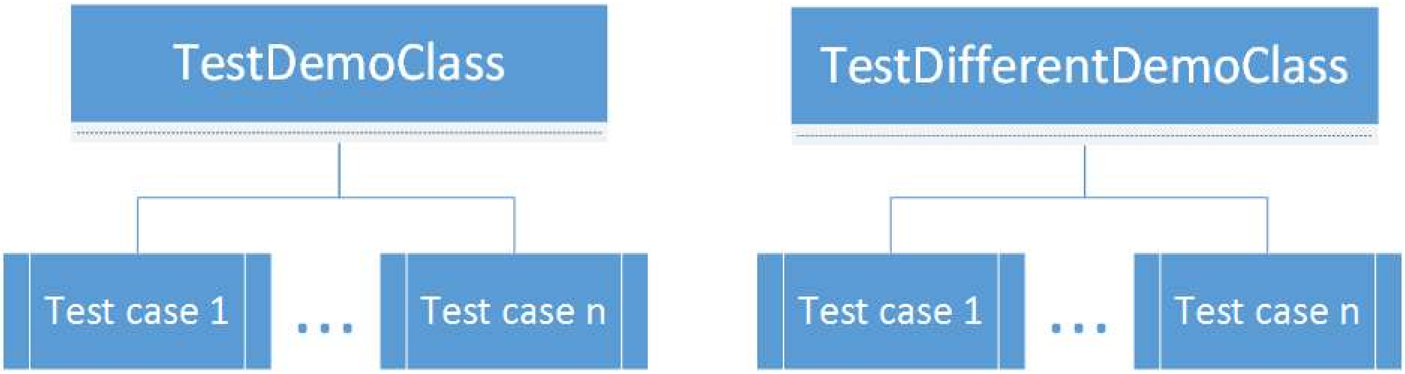
- Unit testing
- Integration testing
- System testing

# *Unit testing* definition

*Unit testing is a method of testing individual units of source code to determine if they are fit for use*

- Unit tests are the smallest testable parts of an application
- Is performed at the development phase of a software’s life cycle
- Detects problems early in the development process
- Unit tests should be independent from one other
- Simplifies Integration
- Not suitable for testing user interface components

# Test Class



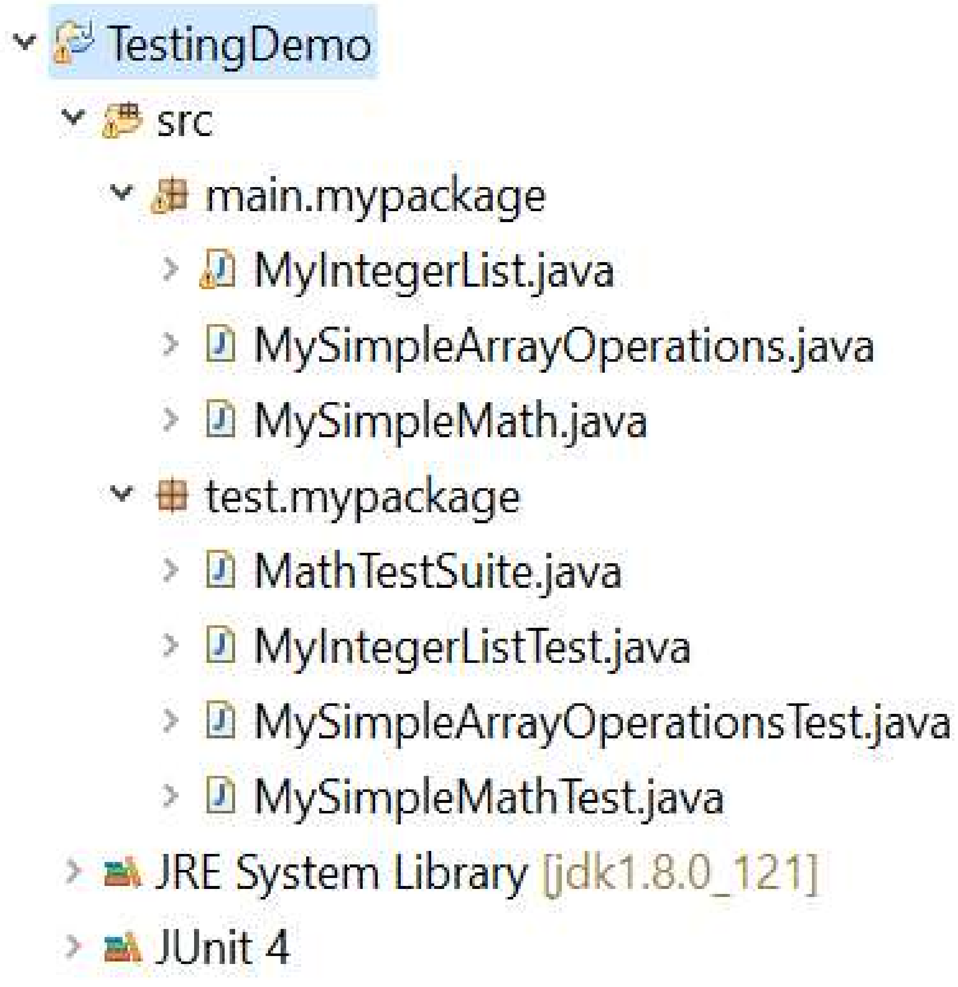
- Test classes are like normal classes, that consist of methods (test cases) that test the functional code of your application

# Test class (2)

- Define test classes and therein implement test methods (known also as “test cases”)
- Good practices:
- Each functional class should have its corresponding test class
- Test classes should have the same name as the functional class but end with an additional ***Test** suffix
- Example:
    - Functional class : MyMathSimple
    - Test class : MyMathSimpleTest

# Test class (3)

Test classes should follow the same package structure with the functional classes



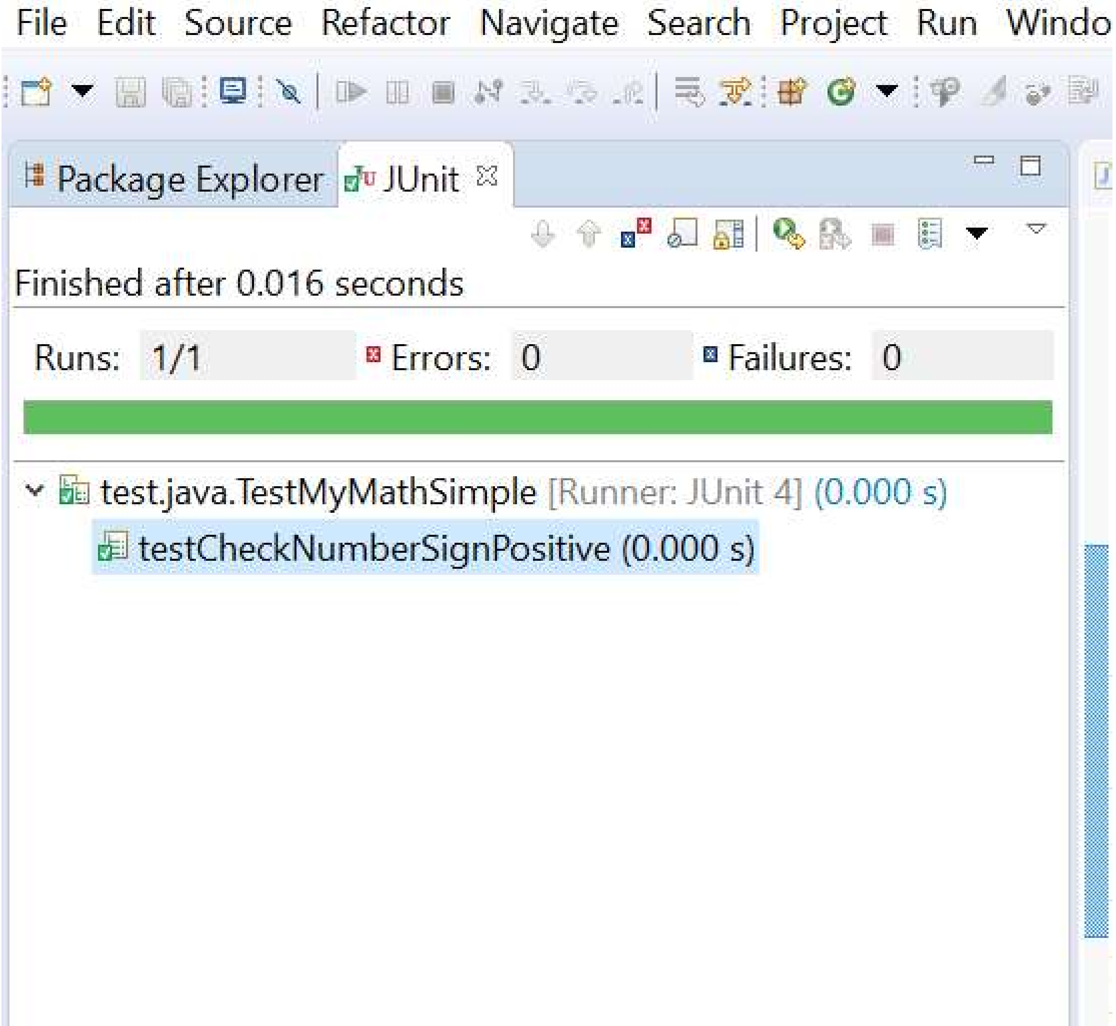
```
TestingDemo
  src
    main.mypackage
      MyIntegerList.java
      MySimpleArrayOperations.java
      MySimpleMath.java
    test.mypackage
      MathTestSuite.java
      MyIntegerListTest.java
      MySimpleArrayOperationsTest.java
      MySimpleMathTest.java
  JRE System Library [jdk1.8.0_121]
  JUnit 4
```

# Proper testing of the "simple" example

- Import the **JUnit 4** library that provides many testing capabilities
- The test class of the the *MyMathSimple* functional class should be created with name **MyMathSimpleTest**

```
package test.main;

import org.junit.*;
import main.java.MySimpleMath;

public class MySimpleMathTest {

    @Test
    public void testCheckSignShouldReturnPositive() {
        MySimpleMath sm = new MySimpleMath();
        Assert.assertEquals("positive", sm.checkSign(5));
        Assert.assertEquals("positive", sm.checkSign(0));
    }

    @Test
    public void testCheckSignShouldReturnNegative() {
        MySimpleMath sm = new MySimpleMath();
        Assert.assertEquals("negative", sm.checkSign(-5));
    }

}
```

# Executing the test cases



Finished after 0.016 seconds

Runs: 1/1 Errors: 0 Failures: 0

```
test.java.TestMyMathSimple [Runner: JUnit 4] (0.000 s)
    testCheckNumberSignPositive (0.000 s)
```

## JUnit good practices

- The *test class* should have the same name with the functional class whose functionalities is testing
- Each `@Test` method is a *test case* that tests a very simple functionality of a method. On method can be tested by many *test cases*
- A *test case*'s name should be self-explanatory of the functionality that is testing and finish the name with the expected result
- *Special cases* should be always tested (zero, edge cases, `null` , etc)

## Test cases annotations `@`

- Annotations `@xxx` are used over the methods in order to depict their functionality
- `@Test` - defines a method as a test method
- `@Test(expected=Exception.class)` – Fails if the method does not throw the expected Exception
- `@Test(timeout=xxxx)` – Fails if the test method requires more time than xxxx milliseconds to execute
- `@Before` & `@After` – This method is executed before (or after) each test method
- `@BeforeClass` & `@AfterClass` – This method is executed once before (or after) the execution of all test methods
- `@Ignore` – Ignores a test method during the execution of the tests

## `Assert`

- Testing of assumptions is done inside each test method by the use of assertions from the Assert class (API) (http://junit.sourceforge.net/javadoc/org/junit/Assert.html)
- The assert methods compare the actual value returned by a test to the expected value, throwing an AssertionException if the comparison fails.

```
assertTrue(“message”,boolean condition)
assertFalse(…)
assertEquals(“message”, expected, actual, tolerance)
assertNotEquals(unexpected, actual);
assertNull(“message”, object)
assertNotNull(…)
assertSame(“message”, expected, actual)
assertNotSame(…)
assertArrayEquals(…)
assertArrayNotEquals(…)
fail()
```

# Testing exceptions

- Lets add a method `double divide(int num, int denom)` in our *MySimpleMath* class

```
package main.java;

public class MySimpleMath {

    /**
     * A simple method that takes and input and returns
     * "positive" or "negative" depending on the input number
     */
    public String checkSign(int number) {
        if(number >= 0 ) {
            return "positive";
        } else {
            return "negative";
        }
    }

    /**
     * Returns the division of numerator by the denominator.
     * If the denominator is zero, it throws an Exception
     */
    public double divide(int num, int denom) {
        if(denom == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        } else {
            return num/(double)denom;
        }

    }

}
```

# Testing exceptions (2)

- We should add our *test cases* in our *TestMyMathSimple* class

```java
package test.main;

import org.junit.*;
import main.java.MySimpleMath;

public class MySimpleMathTest {

    @Test
    public void testCheckSignShouldReturnPositive() {
        MySimpleMath sm = new MySimpleMath();
        Assert.assertEquals("positive", sm.checkSign(5));
        Assert.assertEquals("positive", sm.checkSign(0));
    }

    @Test
    public void testCheckSignShouldReturnNegative() {
        MySimpleMath sm = new MySimpleMath();
        Assert.assertEquals("negative", sm.checkSign(-5));
    }

    @Test
    public void testDivisionShouldReturnPositiveQuotient() {
        MySimpleMath sm = new MySimpleMath();
        Assert.assertEquals(2.0, sm.divide(10, 5), 0);
        Assert.assertEquals(0.0, sm.divide(0, 5), 0);
    }

    @Test
    public void testDivisionShouldReturnNegativeQuotient() {
        MySimpleMath sm = new MySimpleMath();
        Assert.assertEquals(-2.0, sm.divide(10, -5), 0);
    }

    @Test (expected = ArithmeticException.class)
    public void testDivisionShouldThrowArithmeticException() {
        MySimpleMath sm = new MySimpleMath();
        sm.divide(10, 0);
    }

}
```

# Testing `Arrays`

- Assume that we have a class that offers only two functionalities on an array, the `findMin(int[] array)` and the `multiply(int[] array, int factor)` where the first finds the min in an array, and the second multiplies each element of the array with a factor respectivelly

```java
package main.java;

public class MySimpleArrayOperations {

    public int findMin(int[] array) {
        if(!(array.length > 0)) {
            throw new IllegalArgumentException("Input array is empty");
        }

        int min = Integer.MAX_VALUE;
        for(int i=0; i<array.length; i++) {
            if(array[i] <= min)
                min = array[i];
        }

        return min;
    }

    public void multiply(int[] array, int factor) {
        if(!(array.length > 0)) {
            throw new IllegalArgumentException("Input array is empty");
        }

        for( int i=0; i<array.length; i++ ) {
            array[i] = array[i] * factor;
        }
    }

}
```

# Testing Arrays (2)

- For comparing the equality of arrays we use the `assertArrayEquals(...)` method provided by the JUnit library
- The test class should have the following code

```java
package test.main;

import org.junit.*;
import static org.junit.Assert.*;

import main.java.MySimpleArrayOperations;

public class MySimpleArrayOperationsTest {

    @Test
    public void testFindMin() {
        MySimpleArrayOperations msao = new MySimpleArrayOperations();
        int[] array = {10, 2, 3, 10, 1, 0, 2, 3, 16, 0, 2};
        assertEquals(0, msao.findMin(array));
        assertNotEquals(10, msao.findMin(array));
    }

    @Test (expected = IllegalArgumentException.class)
    public void testFindMinShouldThrowException() {
        MySimpleArrayOperations msao = new MySimpleArrayOperations();
        msao.findMin(new int[]{});
    }

    @Test
    public void testMultiply() {
        MySimpleArrayOperations msao = new MySimpleArrayOperations();
        int[] array = {10, 2, 3, 10, 1, 0, 2, 3, 16, 0, 2};
        msao.multiply(array, 10);
        assertArrayEquals(new int[]{100, 20, 30, 100, 10, 0, 20, 30, 160, 0, 20}, array);
    }

    @Test (expected = IllegalArgumentException.class)
    public void testMultiplyShouldThrowException() {
        MySimpleArrayOperations msao = new MySimpleArrayOperations();
        msao.multiply(new int[]{}, 0); //method call with dummy arguments
    }

}
```

# Simplify the test cases

- Make objects that are used in many test cases, instance variables
- Create a `@Before` method if you want to the perform the same operation (ex. initializing instance variables) before the execution of each test case
- After applying the aforementioned changes the class changes to:

# Simplify the test cases (2)

```
package test.main;

import org.junit.*;
import static org.junit.Assert.*;

import main.java.MySimpleArrayOperations;

public class MySimpleArrayOperationsTest {
    private MySimpleArrayOperations msao = new MySimpleArrayOperations();
    private int[] array;

    @Before
    public void initInstanceVariables() {
        System.out.println(this.getClass().getName() + " --> initializing fields");
        this.msao = new MySimpleArrayOperations();
        this.array = new int[] {10, 2, 3, 10, 1, 0, 2, 3, 16, 0, 2};
    }

    @Test
    public void testFindMin() {
        assertEquals(0, msao.findMin(array));
        assertNotEquals(10, msao.findMin(array));
    }

    @Test (expected = IllegalArgumentException.class)
    public void testFindMinShouldThrowException() {
        msao.findMin(new int[]{});
    }

    @Test
    public void testMultiply() {
        msao.multiply(array, 10);
        assertArrayEquals(new int[]{100, 20, 30, 100, 10, 0, 20, 30, 160, 0, 20}, array);
    }

    @Test (expected = IllegalArgumentException.class)
    public void testMultiplyShouldThrowException() {
        msao.multiply(new int[]{}, 0); //method call with dummy arguments
    }

}
```

- Now that we have two test classes (the `MySimpleMathTest` and the `MySimpleArrayOperationsTest` ), is there a way to run both of them in one take?

# Test suites

- Suites allow grouping of Test classes. They are empty classes, annotated with the `@RunWith` and the `@Suite.SuiteClasses` annotations
- When the suite class run all the tests in the included classes will also run

```
package test.java;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({ MyMathTest.class, MyMathSimpleTest.class })

public class MathTestSuite {
    /* the class remains empty, used only as a holder for
     * the above annotations
     */
}
```

# Executing multiple tests

- The `JUnitCore.runClasses` allows the execution of multiple Test classes and suites.
- The results are stored in a `Result` object as defined by the JUnit framework.
- We can execute the `JUnitCore` in (for example) `main` method like the following:

```
package test.java;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(MathTestSuite.class,
              MyArrayOperationsTest.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

# Testing `Lists`

- `Lists, Vectors, Stacks` are `Collections` and collections can be represented as `arrays` by calling the method `aList.toArray();`
- We use this feature to transform a `List` to an `array` and test it as we did in the *testing arrays* slide
- Assume that we have a class that uses an ArrayList of Integers and provides the following functionalities: *add*, *remove*, *clear*, *size*, *get (from an index)*

# Testing `Lists` (2)

```
import java.util.ArrayList;
import java.util.Arrays;

public class MyIntegerList {
    private ArrayList<Integer> list;

    public MyIntegerList(Integer[] array) {
        this.list = new ArrayList<>(Arrays.asList(array));
    }

    public Integer[] getListAsAnArray() {
        return (Integer[]) this.list.toArray();
    }

    public void add(int n) {
        this.list.add(n);
    }

    public void remove() {
        if(!this.list.isEmpty())
            this.list.remove(this.list.size()-1);
    }

    public int get(int index) {
        if(this.list.size() - 1 >= index)
            return this.get(index);
        else
            return (Integer) null;
    }

    public int size() {
        return this.list.size();
    }

    public void clear() {
        this.list.clear();
    }
}
```

# Testing Lists (3)

```
package test.main;

import org.junit.*;
import static org.junit.Assert.*;

import main.java.MyIntegerList;

public class MyIntegerListTest {

    private MyIntegerList myList;

    @Before
    public void initInstanceVariables() {
        System.out.println(this.getClass().getName() + "--> initializing array");
        Integer array[] = {1,2,3,4,5};
        this.myList = new MyIntegerList(array);
    }

    @Test
    public void testSize() {
        assertEquals(5, this.myList.size());
        assertEquals(0, new MyIntegerList(new Integer[]{}).size());
        this.myList.add(6);
        assertEquals(6, this.myList.size());
    }

    @Test
    public void testAdd() {
        this.myList.add(10);
        assertArrayEquals(new Integer[]{1,2,3,4,5,10},this.myList.getListAsAnArray());
    }

    // the other test cases are omitted to simplify the example
}
```

# References

- Github JUnit-team, link (https://github.com/junit-team/junit4/wiki)
- JUnit tutorial by Vogella, link (http://www.vogella.com/tutorials/JUnit/article.html)
- Pragmatic Unit Testing in Java 8 with JUnit, Langr & Hunt, link (https://pragprog.com/book/utj2/pragmatic-unit-testing-in-java-8-with-junit)

# Exercise 1

- Create a project (with a name of your preference) and a package `main.mymath`
- In this package create a class `MyCalculator` that implements a calculator and provides the following functionalities (methods) for any pair of **positive integers** :
- addition
- multiplication
- division

# Exercise 1 (continued)

- Consider checking the input numbers for their illegibility. For example:
- the denominator of the division cannot be zero,
- input numbers should not result to overflow (ex. the result of adding two *int* numbers should fit in a int variable)
- think of more cases if exist..
- In cases that the input values violate your constraints, you should throw an IllegalArgumentException with a corresponding message

# Exercise 1 (continued)

- Create a package `test.mymath` in the same project
- Create a test class `MyCalculatorTest` in the package and implement test cases to challenge the functionality of all methods in the `MyCalculator` class
- Be sure that
- you use appropriate names for the test cases
- edge numbers are tested
- exceptions are also tested

# Exercise 2

- Create a class named `MyAdvancedMath` and implement the following methods
- `int factorial(int n)` : Calculates and returns the factorial of a given non-negative number `n` . If `n < 0` throw an `IllegalArgumentException("n cannot be < 0")` . Also investigate which is the largest factorial that can fit in an integer variable and do not allow the user (throw an `IllegalArgumentException` ) to give an `n` larger than the value that causes an overflow.

# Exercise 2 (continued)

- `double power(int b, int n)` : Calculates and returns the power of a number, where `b` is the base and `n` is the exponent. The exponent (n) can be any integer between [0, 20]. If the input is larger than that, an `IllegalArgumentException("n should be 0 <= n <= 20")` should be thrown
- `int[] reverse(int[] array)` which should return an array which is the reversed of the one you gave as an input

# Exercise 2 (continued)

- Create a **Test class** which will host all **test cases** for your implemented methods in the calculator class

# Exercise 3

- Create a Test Suite which consists of all the Test Classes that you created earlier
- Execute the Test Suite with the `JUnitCore.runClasses` , store the results in a `Result` object and print the failures (if exist)