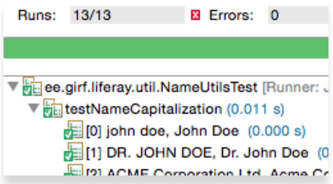

Runs: 13/13 Errors: 0

ee.girf.liferay.util.NameUtilsTest [Runner: J

testNameCapitalization (0.011 s)

[0] john doe, John Doe (0.000 s)

[1] DR. JOHN DOE, Dr. John Doe (0

[2] ACME Corporation Ltd, Acme Co

Testing

1 Unit testing

JUnit is a Java package used to automate testing processes. By creating Tests, JUnit will perform tests in the code indicated by the user. Whenever we are going to develop some kind of software, we will have to take into account the tests to be done. With JUnit we will make sure that our program works correctly in different situations.

1.1 Creation of a Test Case

Using JUnit in Eclipse you can create JUnit Test Case classes. To do this, press the right mouse button on a package and then *New* > *Other* and select within the folder *Java* > *JUnit* > *JUnit TestCase*. You can select the class to test in the field "*Class under test*". In this case we want to test the class Point from the first laboratory session on Object Orientation.

Test cases are classes with methods annotated using the @Test annotation from org.junit. Each test can *pass* or *fail*. As a unit test case, the test case must check a specific functionality of a class.

Complete the following example test case by writing a test to verify the performance of the Point class, particularly of the method which calculates the distance between a point and the origin. The test should check that for a given point (x = 4, y = 6), the distance to the origin is 7.211102550927978.

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

public class PointTest {
       @Test
       public void testFail() {
               fail("Not yet implemented");
       }

       @Test
       public void testOk() {
               assertEquals(2,2);
       }

       @Test
       public void testError() {
               int i = 1/0;
       }

       @Test
       public void testDistance() {
               //to be completed
       }

 }
```

1.2 Test-driven development

It may be advisable to implement the tests before writing the code to be tested. There is a development methodology called *Test-Driven Development*, which is based on implementing the tests first and then developing the code. A series of test cases are designed and implemented from the requirements of the application. At first, the tests will not even compile, since there will be neither the class nor the method to test. With the help of Eclipse we can generate automatically the components we need; to do so, we must click on the error marked by Eclipse, and then choose among the proposed solutions.

Once the code compiles, you can run the test to see if it fails. This way we can make sure that the tests are actually checking something.

Now, you are requested to implement the necessary code to be able to correctly verify the following Test. The necessary code is a new class, called FullProfessor, which inherits from Professor (second laboratory session on Object Orientation). A Full Professor will have an ID, salary, department and benefits. The net salary will be half the sum of both salary and benefits.

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

public class FullProfessorTest {

        @Test
        public void testNetSalary() {
                FullProfessor c = new FullProfessor("00030000-C", 2000, 500, "math");
                float result = c.calculateNetSalary();
                float resultExpected = 1250.0f;
                assertEquals(resultExpected, result, 0.01);
        }
}
```

1.3 Testing exceptions

In addition to testing that the performance of a code is correct (when it is expected to be correct), it is convenient to test that the code also fails when it is expected to fail. The following example shows how to indicate that the expected correct behavior of a code is the throwing of a particular exception. In the example, two approaches to do this test are presented. The first option, which is the recommended by JUnit 5, makes uses of the method `assertThrows`. The second option makes uses of the try/catch block. In this case, you can put the code you expect to throw an exception under a try block and catch the exception with a catch block. Nevertheless, it is important to put `fail()` after the line which should throw the exception in the try block to avoid false positives (if this line is not included and if the exception is not thrown, the test would pass).

Complete the JUnit test (using both options) to verify that the exception that is thrown when attempting to create a Rectangle (this class was defined in the third laboratory session on Object Orientation) given erroneous vertices is correct.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class RectangleTest {


        @Test
        public void testDivisionWithException() {
                assertThrows(ArithmeticException.class, ()->{int a=1/0;});
        }

        @Test
        public void testDivisionWithException2() {
                try {
                        int i = 1 / 0;
                        fail("Failed test");
                }catch(ArithmeticException e) {
                }
        }

        @Test
        public void testIsRegular(){
                //to be completed
        }

        @Test
        public void testIsRegular2(){
                //to be completed
        }
}
```

1.4 Test parameterization

Given the following class which calculates the square of a number:

```
public class MyClass {

       MyClass() {
       }

       public int square (int i) {
               return i * i;
       }
}
```

You are requested to run the following test cases:

Input	Output
2	4
5	25
11	121

To do this you must use JUnit annotations that allow running tests with different input values (Parameterized). You can use the following template for this exercise:

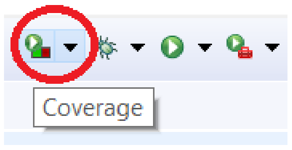
```
import static org.junit.jupiter.api.Assertions.*;

import java.util.stream.Stream;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

public class MyClassTest {

       private static Stream<Arguments> _______________() {
           return Stream.of(
             Arguments.of(__, __),
             Arguments.of(__, __),
             Arguments.of(__, __));
         }


       @ParameterizedTest
       @MethodSource(____________)
       public void testSquare(_________, __________) {
               ______________________________________
               ______________________________________
       }

}
```

For more information, check the documentation available at https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources-MethodSource

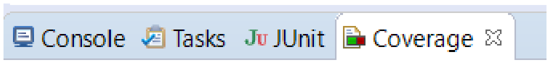
1.5 Full code coverage

Given the following code which represents a Chessboard, you must design White Box tests that are able to execute all the lines of code of the `displayZone()` method. Once you design the Test class, you must run the code using the Code Coverage tool, EclEmma, which is installed as an Eclipse plugin (newer versions of Eclipse already include EclEmma as a built-in tool). This tool shows the areas covered by the tests in green.

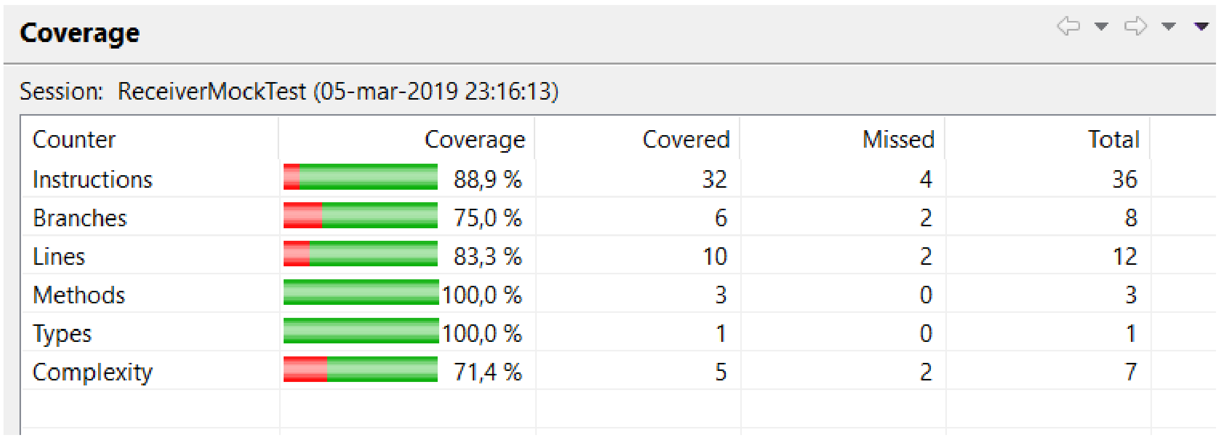
If you want to execute the Code Coverage tool, you need to click on the Coverage button (see Picture)



Afterwards, your code will be colored depending if lines have been covered in your tests or not. If you want to discover your coverage, you can click on a Tab called "Coverage".



When you click in this tab, a tree with the class hierarchy will be displayed. You can select the class you are testing and click with the right botton of your mouse in the class or a specific method. Then, you click on "Properties" and a new window will be opened. You can click on "Coverage" in the left panel and the coverage of your class/method will be shown. The format of the coverage will be as follows:



Coverage

Session: ReceiverMockTest (05-mar-2019 23:16:13)

Counter	Coverage	Covered	Missed	Total
Instructions	88,9 %	32	4	36
Branches	75,0 %	6	2	8
Lines	83,3 %	10	2	12
Methods	100,0 %	3	0	3
Types	100,0 %	1	0	1
Complexity	71,4 %	5	2	7

```
public class SquareGrid {
    private int size;

    // precondition n%2 == 0
    SquareGrid(int n) {
        size = n;
    }

    // precondition 0 <= x <= n
    // precondition 0 <= y <= n
    public String getZone(int x, int y) {
        String s = "";
        if (y >= size / 2) {
            s = s+"N";
        } else {
            s = s+"S";
        }
        if (x >= size / 2) {
            s = s+"E";
        } else {
            s = s+"W";
        }
        return s;
    }}
```

1.6 Making use of fixtures

It is common for several of the implemented tests to use the same input or to have the same expected output data. In order not to have repeated code in the different test methods, we can use the so-called fixtures, which are fixed elements that will be created before each proof. JUnit has the following annotations for fixtures:

1. @BeforeEach The method is run before each test.

2. @AfterEach The method is run after each test.

You are requested to make a new Test, called TestPoint_Fixtures to test the methods quadrant and nearest of class Point (Session 2 – Object Orientation) with the same input data:

```
p = new Point(4, 6);
otherPoints = new Point[] {new Point(1, 1), new Point(5, 3),
            new Point(10, 10), new Point(-3, 2), new Point(-4, -5)};
result = new Point(5,3);
```

Note: Comparing two objects in Java can be complicated, so testing the method nearest will need some tweaks (Hint: override equals() in the class Point).

1.7 Combining multiple Test Cases in a Test Suite

Test Suites allow running several test classes at a time. Now, you are asked to create a TestSuite by combining some of the TestCase from previous exercises (e.g., PointTest and MyClassTest). You can use the following template for this exercise:

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses( { ____________, _______________ } )
public class AllTests {
}
```

2 Integration testing

In the tests done so far, the methods to be tested received all necessary input data through parameters, but in a real application the behavior of the methods will also depend on other classes, such as classes that extract data stored in a database, or classes that access services offered through the network.

The problem is: how can we test / implement methods that depend on other classes that may not yet be written or tested? The solution is to create fake objects from fake classes (mock objects) that simulate the minimum possible behavior to test.

You must create a mock class that replaces the original class with the impostor. So you must use interfaces (which will be later on implemented by the classes) in the Test, instead of using the original classes, and make use of the polymorphism to run the mock instead of the original class.

2.1 Abstracting behavior through interfaces

Let's assume that we have a database of students with the grade obtained in their most recent exam. For simplicity, first:

1. Create a simple Java class to represent a Student (name: String, grade: float) by encapsulating its fields.
2. Create a class StudentDAOMock with the following public methods :
    a. Default constructor that generates an array of 2 objects of type Student with any name and random grade (between 0.0f and 10.0f) and stores them in its internal state. See the Math.random() documentation for help.
    b. findAll() -> Student[], which returns the array of students created in the constructor
    c. minGrade() -> float, which returns the minimum grade of the students
    d. maxGrade() -> float, which returns the maximum grade of the students
    e. avgGrade() -> float, which returns the average grade of the students
3. Create an interface StudentDAOApi, which represents the behavior of StudentDAOMock, documenting it.
4. Modify the signature of StudentDAOMock to indicate that it implements the interface.

2.2 Testing the Mock

Write a test case that tests the correct performance of any class that implements StudentDAOApi. In particular, check that the values returned by minGrade() maxGrade() and avgGrade () are consistent with each other, and that they are contained in the range [0.0f, 10.0f].

In the future we would write the full code of a real class StudentDAO, which consults the database management system to perform the operations. But the unit test of this class is already done, and we would just need to modify one line, replacing the StudentDAOMock object by StudentDAO.

This way we can continue with the development of the complete application and begin to carry out the integration of modules based on the fact that the Student module which queries the database is not yet implemented but is already tested, thanks to the mock.