



Australian  
National  
University

# Software Testing

Lab

## Task for this week

- 1) Implement test cases using Junit
- 2) Implement a minimum number of Junit test cases for the given method to achieve path complete.
- The code structure/skeleton is available on Wattle
- **This lab contains assessable items!**
- **Submission Guidelines**
  - The last slide contains information about the submission
  - Read it carefully to avoid losing marks!

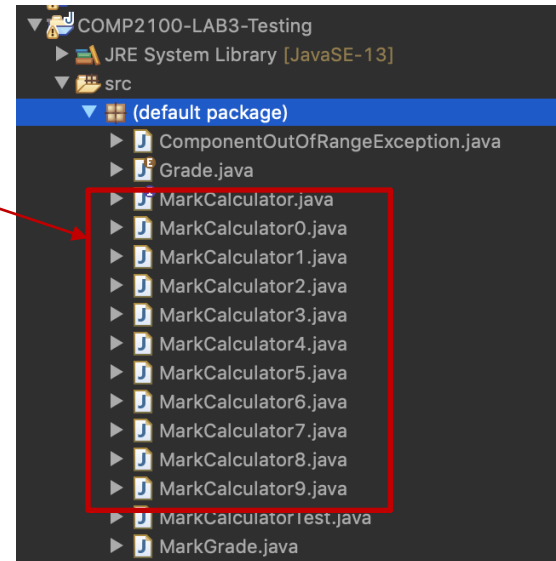
# Task 1 – Finding the correct implementation!

*The goal of this task is to give you an idea on how assessable items will be evaluated throughout the semester. Most of the lab-related activities and final exam will be auto-graded via test cases. We will use JUnit4, which provides a simple way of creating unit tests for an application you are developing.*

## Task:

10 versions of a `calculateMark` method has been written, but **ONLY ONE** is correct!

The task is to write JUnit tests to determine which of these implementations are incorrect.



# Task 1 – Finding the correct implementation!

Follow the steps below:

1) Download a copy of the code **but do not read** the code for the different implementations (i.e. the code in [MarkCalculator0.java](#), [MarkCalculator1.java](#), ..., [MarkCalculator9.java](#)). Your tests should be driven by the specification only. (It will be more difficult to try to understand all implementations than writing the tests following the specification! Believe me! 😊).

2) Read the [MarkCalculator.java](#) file. This file includes an interface and a comment that provides an exact specification of what the **calculateMark** method does. Identify test cases from the interface specification.

Hint: before start implementing anything, think about the cases: boundary cases, cases you expect to fail, and maybe some random cases that you expect to succeed.

```
/**
 * The calculateMark method must return the MarkGrade based on the following:
 *
 * + The "lab", "Assignment1" and "Assignment2" marks are between 0 and 10 (inclusive).
 * + The "FinalExam" is between 0 and 100 (inclusive).
 *
 * If any of these components are not within the expected range
 * then a ComponentOutOfRangeException is thrown.
 *
 * If a student does not attend the final exam (attendedFinal will be false)
 * then a grade of NCN should be given and
 * the final mark must be set to null.
 *
 * The lab mark is worth 10% of the final grade and marked out of 10,
 * the assignments are worth 15% each of the final grade and marked out of 10, and
 * the final exam is worth 60% and marked out of 100. These marks are combined EXACTLY
 * and then rounded to produce an integer out of 100 (round up for values exactly
 * half way between integer values). Once this is done
 * the grade can be determined using the following ranges:
 *
 * + 0 ~ 44 : N
 * + 45 ~ 49 : PX
 * + 50 ~ 59 : P
 * + 60 ~ 69 : C
 * + 70 ~ 79 : D
 * + 80 ~ 100 : HD
 *
 * This calculateMark does not represent the way you will be assessed in this course.
 * This is just a simple task to understand how software testing works.
 */
public interface MarkCalculator {
    MarkGrade calculateMark(int lab, int assignment1, int assignment2, int finalexam, boolean attendedFinal)
        throws ComponentOutOfRangeException;
}
```

# Task 1 – Finding the correct implementation!

3) Read [MarkCalculatorTest.java](#). This file implements base JUnit testing framework with `ParameterizedTest`.

- There is an ArrayList that create instances of **MarkCalculator** (each array position has a **MarkCalculator** instance with different implementations).
- Note that for each test, a different instance of the **MarkCalculator** will be executed.

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

@RunWith(Parameterized.class)
public class MarkCalculatorTest {
    /**
     * Return a list of parameters which are different implementation of
     * interface {@link plain MarkCalculator}.
     * Do NOT Modify this part
     */
    @Parameters
    public static Iterable<? extends Object> getImplementations() {
        return Arrays.asList(
            new MarkCalculator0(),
            new MarkCalculator1(),
            new MarkCalculator2(),
            new MarkCalculator3(),
            new MarkCalculator4(),
            new MarkCalculator5(),
            new MarkCalculator6(),
            new MarkCalculator7(),
            new MarkCalculator8(),
            new MarkCalculator9()
        );
    }

    @Parameter
    public MarkCalculator calculator;

    // ##### YOUR CODE STARTS HERE #####

    /* EXAMPLE Test case 1 */
    @Test(expected = ComponentOutOfRangeException.class)
    public void testException() throws ComponentOutOfRangeException {
        this.calculator.calculateMark(-1, 0, 0, 0, true);
    }
}
```

# Task 1 – Finding the correct implementation!

4) Implement a set of test methods and cases, identified in the previous step, that aims to uncover errors in the implementation (black box testing) into `MarkCalculatorTest.java`.

```
// ##### YOUR CODE STARTS HERE #####

/* EXAMPLE Test case 1 */
@Test(expected = ComponentOutOfRangeException.class)
public void testException() throws ComponentOutOfRangeException {
    this.calculator.calculateMark(-1, 0, 0, 0, true);
}

/* EXAMPLE Test case 2 */
@Test
public void testGradeN() throws ComponentOutOfRangeException {
    assertEquals(new MarkGrade(0, Grade.N), this.calculator.calculateMark(0, 0, 0, 0, true));
}

//TODO: write other test cases

// ##### YOUR CODE ENDS HERE #####
```

# Task 1 – Finding the correct implementation!

Which implementations passed your tests?

**NOW** read over the code and see if you can spot any errors (code review). Create some other test cases based on being able to view the code (**white box testing**).

Additionally, think about what would be the minimum number of tests case that are statement/branch/path complete for the different implementations?

Which implementation now passes your tests?

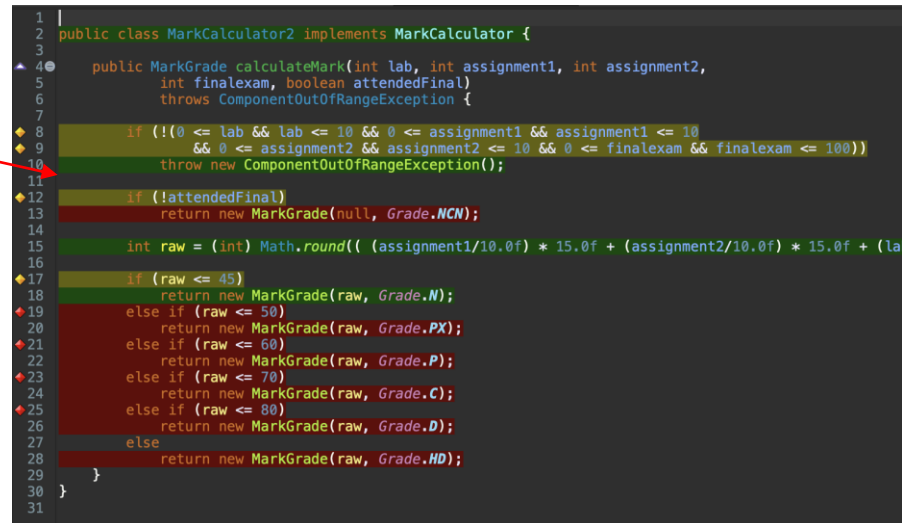
Did your code review spot any errors?

Now which implementations passed your tests?

Which implementation is exactly correct?

Do you think JUnit tests can check code correctness? (Y/N, if No, why?)

These questions are for your reflection only.




```
1 |
2 | public class MarkCalculator2 implements MarkCalculator {
3 |
4 |     public MarkGrade calculateMark(int lab, int assignment1, int assignment2,
5 |                                     int finalexam, boolean attendedFinal)
6 |                                     throws ComponentOutOfRangeException {
7 |
8 |         if (!(0 <= lab && lab <= 10 && 0 <= assignment1 && assignment1 <= 10
9 |             && 0 <= assignment2 && assignment2 <= 10 && 0 <= finalexam && finalexam <= 100))
10 |             throw new ComponentOutOfRangeException();
11 |
12 |         if (!attendedFinal)
13 |             return new MarkGrade(null, Grade.NCN);
14 |
15 |         int raw = (int) Math.round(( (assignment1/10.0f) * 15.0f + (assignment2/10.0f) * 15.0f + (la
16 |
17 |         if (raw <= 45)
18 |             return new MarkGrade(raw, Grade.N);
19 |         else if (raw <= 50)
20 |             return new MarkGrade(raw, Grade.PX);
21 |         else if (raw <= 60)
22 |             return new MarkGrade(raw, Grade.P);
23 |         else if (raw <= 70)
24 |             return new MarkGrade(raw, Grade.C);
25 |         else if (raw <= 80)
26 |             return new MarkGrade(raw, Grade.D);
27 |         else
28 |             return new MarkGrade(raw, Grade.HD);
29 |
30 |     }
31 | }
```

## Task 2 – Path Complete

*The goal of this task is to give you an idea on how to create a minimum number of tests to achieve Path Complete.*

### Task:

Implement the minimum number of JUnit test cases for **findSomething** that is path complete.



```
public static int findSomething(int a, int b, int c) {  
    if(a > b) {  
        if(a > c) {  
            System.out.println("a");  
            return a;  
        }else {  
            System.out.println("c1");  
            return c;  
        }  
    }else if(b > c) {  
        System.out.println("b");  
        return b;  
    }else {  
        System.out.println("c2");  
        return c;  
    }  
}
```



- **Assignment deadline:** see the deadline on Wattle (always!)
- Submission mode: via Wattle (Lab Testing)
- Each task worth 1 mark (total 2 marks)
  - Task 1: For each calculator identified as incorrect implementation, 1/10 marks (the correct implementation is worth 1/10, but only if all incorrect ones are found). Any test case that fails all implementations will be discarded.
  - Task 2: The minimum number of paths to achieve path complete was found: 1 mark. If more test cases than the minimum required were used: 0.5 marks; if test cases don't cover all paths (only part of them): 0.2 marks. If no path is covered, 0 marks.

Submission format (**IMPORTANT**):

- Upload **only** your final version of `MarkCalculatorTest.java` and `PathCompleteTest.java` to Wattle
- **Do not** change the file names
- **Do not** upload any other files (only the specified files are needed)
- **Do not** upload a folder (your submission should be only **the two java files**).
- The answers will be marked by an automated marker.
  - **Do not** change the structure of the source code including class name, package structure, etc.
  - **You are only allowed to edit the designated code segment indicated in the comments.**
- **Do not** import packages outside of the standard java SE package. The list of available packages can be found here: <https://docs.oracle.com/en/java/javase/12/docs/api/index.html>
- Any violation of the submission format will result in zero marks