Week 5 Lab

Unit Tests and Javadoc

Objectives

In this week's lab, you will:

- use assertions and exceptions for defensive programming
- document code using Javadoc
- use unit tests

No class diagram this week
You won't be adding any new functionality to the JavaUniversity system this week, so there's no class diagram to submit. We suggest that you prepare by making sure that you have completed all functionality from previous labs, and that you understand and can document the structure of the existing system.

You will be adding JUnit test classes to the system, but you are not expected to show these on a class diagram.

Before you begin this week's lab, make sure you have submitted the homework tasks from last week.

Task 1. Experiment with assertions and exceptions

At Java University, valid student IDs consist of precisely eight digits. Multiple leading zeroes are permitted.

Modify the constructor of `Student` to throw an exception when an attempt is made to construct a new `Student` where this property does not hold. Catch the exception in `University` and print a stack trace, using a similar approach to the `readstring()` method provided in the Week 3 lab.

Verify that this works as expected before proceeding to the next task.

Task 2. A unit test for `Exams`

At the moment, the `Assessment` class hierarchy has no input validation. You can create an `Exam` that is -50 minutes long, or an `Assignment` that makes up 200% of your final mark.

One way to improve the reliability of software is to have robust and comprehensive checks to ensure only valid data enters the system. Therefore, we will add some checking and exceptions to the `Assessment` class hierarchy, to ensure that invalid `Assessments` do not exist within the system.

Furthermore, we will use automated unit tests to give sufficient confidence that the code in the `Assessment` classes is implemented correctly, including the condition-checking and exceptions.

The first step is to implement a unit test for existing functionality within the `Exam` class.

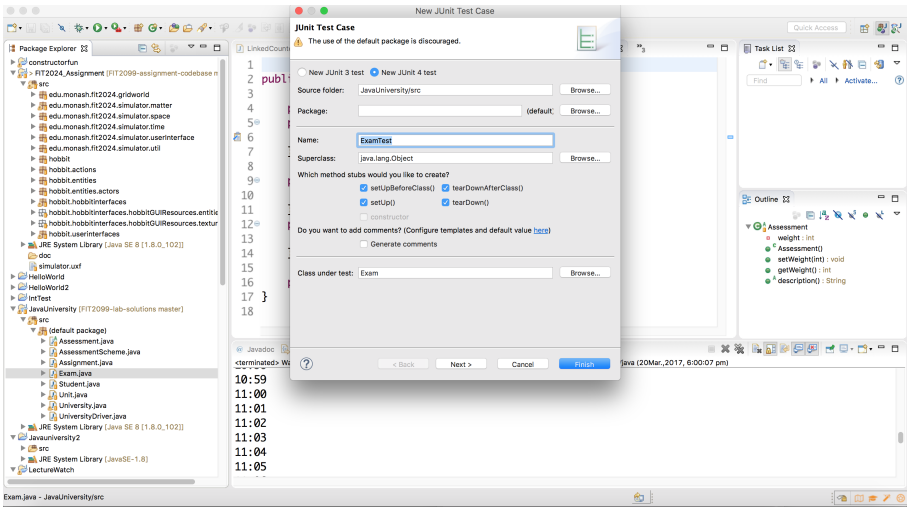
The most common unit test framework used for Java is JUnit [1]. While it is not part of the Java language or standard class libraries, it is widely used within the Java programming community.

We will use JUnit version 4. JUnit 5 is about to be released, but we will stick with version 4 for now as all our tools support it by default. It is open source software[2].

Creating JUnit tests is easy, and Eclipse can do a lot of the typing for you.

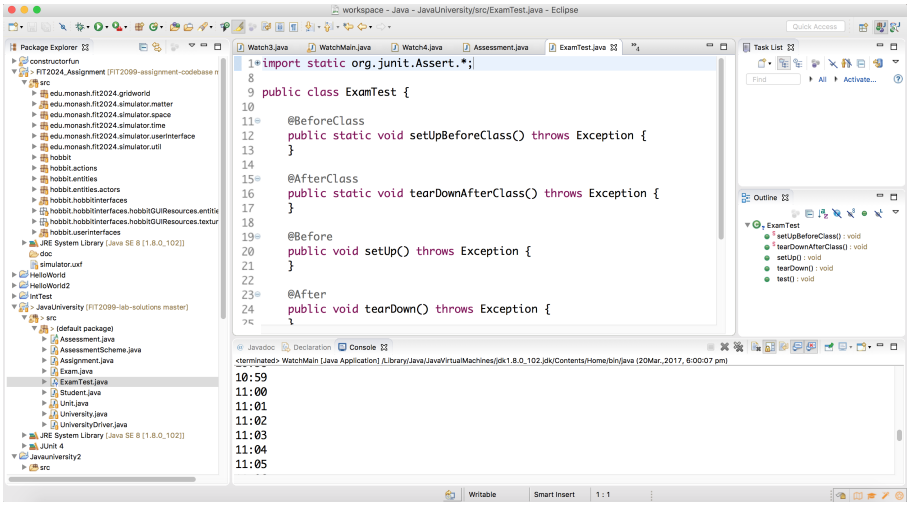
JUnit test cases are themselves written in Java. Typically, the unit test cases for a class will be put in a single unit test class, and by convention the test class for a class named **`Foo`** would be named **`FooTest`**.

Right-click on the Exam class within the Package explorer, and select `New` → `JUnit Test Case`. A dialog box will appear so you can easily configure your test class.



Create a test class called `ExamTest`. Select the four checkboxes to create all four method stubs.

The `ExamTest` class will be created:



[1] `http://junit.org/junit4/`

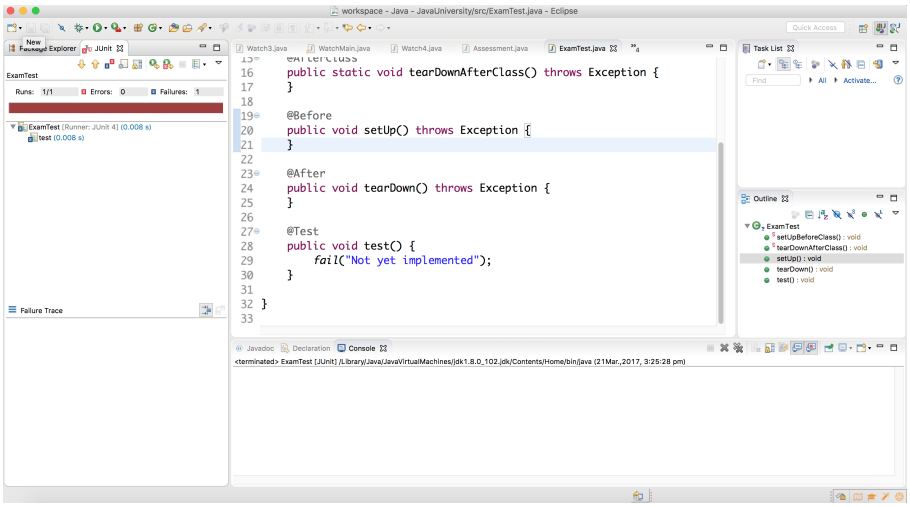
[2] If you don't know what open source means, you really should find out. You can start with `https://opensource.com/resources/what-open-source` for a pragmatic take on what it is and why it's important. An alternative view on the virtues of the same approach to licensing software - free software - is available from the GNU Project `https://www.gnu.org/philosophy/free-sw.html`.

For a full explanation of the methods in the screenshot, see the appendix at the end of the lab sheet.

At the bottom of the new test class, there is a test stub. At the moment, it fails with the error message "not yet implemented".

Let's try to run the tests for your project. You can do this in a number of ways; the easiest is to right-click `JavaUniversity` in the Package Explorer, select `Run As...` → `JUnit Test`.

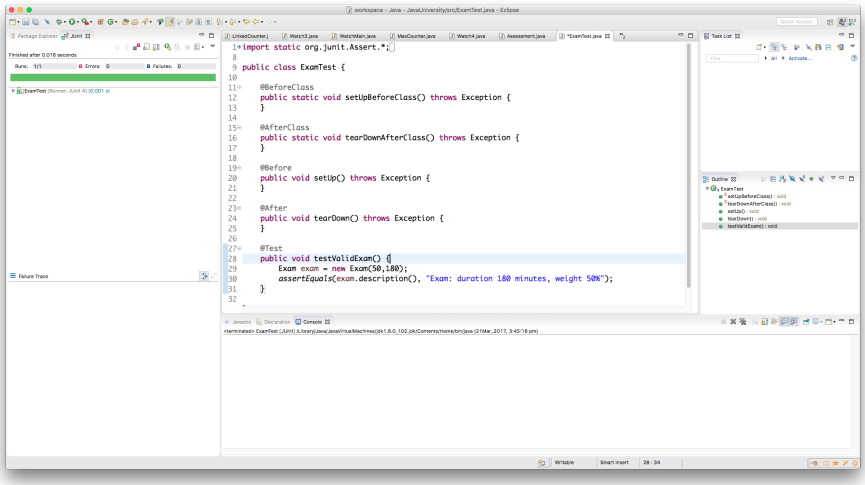
Eclipse runs the test and reports the result. As expected, the test fails:



Let's create a real test. Replace the `test()` method with the following

```
@Test
public void testValidExam() {
    Exam exam = new Exam(50, 180);
    assertEquals(exam.description(), "Exam: duration 180 minutes, weight 50%");
}
```

This test creates an `Exam` object and asserts that the description matches a string provided by the tester. As the assertion is true, the test succeds when you run it:



Task 3. Add input checks to `Exam`, and check with unit tests

.

Now, let's add some proper input validation to `Exams`, and use unit tests to give confidence that we have implemented the validation correctly.

Firstly, according to Java University policy, the minimum and maximum duration of an exam is 30 and 180 minutes, respectively. Add a check to the `Exam` constructor, and throw an exception if the duration is shorter than 30 minutes. The exception message should be "Exam duration too short".

Run your unit test again and check to see that it still passes.

Now, add another test to `ExamTest` to confirm that the correct exception is thrown if the exam is too short. There are a couple of ways to do this [3], but in this case we will use a try-catch block within the test code. Copy this code into `ExamTest`:

```
@Test
public void testTooShort() {
    try {
        Exam exam = new Exam(50, 25);
        fail("exception␣expected␣but␣none␣thrown");
    } catch (Exception e) {
        assertEquals(e.getMessage(), "Exam␣duration␣too␣short");
    }
}
```

Please note: this does not *guarantee* that the check has been properly implemented. It's not even a particularly good choice of test inputs (hint: if you need to test a numeric boundary, test *on* the boundary).

Now that you've seen how unit tests work, have a go at some basic *test-driven development*.

Add another test to check that the code throws an exception if the duration is too long. Check that the test fails, and only *then* implement the code sufficient to make the test pass.

Now add checks for `Assessment` weights. The minimum weight of an assignment is 1%, and the maximum is 100%. Note that you can't create plain `Assessment` objects, so to test your code you'll need to create either an `Exam` or an `Assignment` test.

Make any changes you need to make to other classes to make the rest of your code run. For the moment, you can handle exceptions in the main `JavaUniversity` application by quitting and printing a stack trace.

Task 4. Add validation to `AssessmentScheme` and unit test it

The sum of the weights in an `AssessmentScheme` must be 100. Add code to throw an exception if this condition does not hold.

Add unit tests to check that your validation code works.

Task 5. Document class using Javadoc

As discussed in lectures, code can be worked on by many people and have a very long lifespan. As such, it is vitally important to document your code properly.

[3] See `https://github.com/junit-team/junit4/wiki/exception-testing` for relatively easy to understand documentation

The easiest and best place to document the details of an individual class is in the class itself, and it's essential to maintain or extend the class. However, other programmers using your class should not be required to look at the source code - it's slow and can lead to them relying on implementation details you want to change later. So it's important to be able to produce separate interface documentation.

We can use Javadoc to simplify this process - if you format your comments in the right way, you can generate spiffy-looking API documentation automatically. This is how the Java API documentation is generated.

We have provided you with two guides to writing Javadoc documentation on the subject Moodle page, including one on the syntax of Javadoc, and another on the principles of what to write in code comments. We have also provided you with a documentation standard for code written for FIT2099.

Document the `Assessment` abstract class, and its subclasses, following the documentation standard we provided, and generate HTML documentation using the instructions in the guides. Make sure you comment each method as well as the classes.

Task 6. Polishing Marks

Document your Mark code appropriately using Javadoc.

Design and implement unit tests for the Mark class. Try to include tests for valid and invalid input data where possible.

Getting marked

Once you have completed this lab, you must submit it on Moodle in order to receive marks. This applies even if your demonstrator has given you feedback in class – you may need to be re-marked and this won't be possible unless your work has been uploaded.

Note that to receive full marks, it is not sufficient merely to produce correct output. Your solution must implement the functionality described above.

Marking

This lab is worth 2% of your final mark for FIT2099. Marks will be granted as follows:

- 0 marks if you do not complete all tasks
- 1 mark if all tasks are completed but the implementation is buggy or does not match the spec
- 2 marks if all tasks are complete and the implementation is correct

Appendix: JUnit Test Class Methods

Note that the five methods each have their own *annotation*. You've already seen one of Java's inbuilt annotations, `@Override`. The annotations in this class are unique to JUnit, and tell the JUnit runtime when those methods should be run.

Implementing your own custom annotations is possible but using them to do anything requires a fairly advanced knowledge of the innards of Java. We won't cover doing so in this unit.

In any case, when a JUnit test class is run, the JUnit test runner executes the test code in the following sequence:

1. The method annotated with `@BeforeClass`. This method does any initial setup that only needs to be done once. It will only run once per test invocation.
2. For *each* method `foo` decorated with `@Test`:
   (a) The method annotated with `@Before`. This method does any setup that needs to be done before *each* test. This potentially runs many times!
   (b) `foo`
   (c) The method annotated with `@After`, which cleans up after each test. This method potentially runs many times!
3. The method annotated with `@AfterClass`. This method is used to do any final cleanup.

Note that tests may be executed in any order by the JUnit test runner. If your test code relies on tests executing in a particular order, this is bad practice, and the tests may stop working at any moment.

The principle of Don't Repeat Yourself applies to test code, too. If you find yourself repeating code in many test cases, you should move it into a separate method, or perhaps into one of the setup or teardown methods.