Lab 1: Introduction to JUnit

In this lab, you will write JUnit tests on a code base.

Objectives:

- To write unit tests using JUnit

Part 1: Introduction to JUnit

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

If your computer has the standard lab setup, these directories will be:

```
Lab Installation directory:      c:\Users\username\javaclass-p2
```

_2. Start IntelliJ running by selecting from the Windows Start menu.

_3. Your first job is to create a new *project* for your lab. Follow these steps:

a. In the "Welcome to IntelliJ" window, click the *Create New Project* entry.

b. On the first wizard page, note that the default for the *Project SDK* references your Java installation, then press Next.

c. On the next page, press Next.

d. On the next page, for the *Project name*, enter **Lab01**.

For the *Project location*, click the button with the "...", then browse to **{Lab Installation Directory}/workspace**. If your computer has the standard lab setup, the directory should be:

```
c:\Users\username\javaclass-p2\workspace
```

Substitute your actual user name.

Then add **Lab01** to the end of the *Project location* path.

Then press Finish. Note that IntelliJ warns you that the directory doesn't exist and will be created.

On the left side of the window, you should see the Lab01 project - click the triangle to expand it. You should see *.idea* and *src* folders as well as a couple of other entries.

_4. Your first job is to create the code base for which you will write unit tests.

a. Right-click on the *src* folder and create a new package named **formulas**.

b. Right-click on the new package and create a new Java class named **QuadraticFormula**.

c. In the class, write two methods that solve the quadratic formula:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
public double solveQuadraticPlus (double a, double b, double c)
{
    double inner = b*b - 4*a*c;
    double radical = Math.sqrt(inner);
    double upper = -b + radical;
    double result = upper/(2*a);
    return result;
}

public double solveQuadraticMinus (double a, double b, double c)
{
    double inner = b*b - 4*a*c;
    double radical = Math.sqrt(inner);
    double upper = -b - radical;
    double result = upper/(2*a);
    return result;
}
```

_5. Next, create your test-case class:

a. In the editor for QuadraticFormula.java, click on the *QuadraticFormula* text in the class definition so that the text-entry cursor is within the class name, then press Alt+Enter and choose *Create Test* to start the wizard.

When IntelliJ asks to create test in the same Source Root, press OK.

b. On the wizard, at the top, for the Testing Library, click the *JUnit4* selection, then press the Fix button so that IntelliJ adds the JUnit4 JAR to the project's CLASSPATH.

For the *Class name*, accept the default of **QuadraticFormulaTest**.

For the *Generate test methods for* list, put a checkmark next to both of the methods, then press OK. IntelliJ creates the test-case class and opens it into the editor.

_6. Examine the generated test-case class:

- It imports the necessary types for JUnit test-cases.
- It defines a POJO (plain old Java object) class for the test-case class.
- It defines testXXXX() methods for each of the methods you selected in the wizard. Note that each of these test-case methods has the *@Test* annotation and that the wizard generated a call to *fail()* in each test-case method.

If the test-case class has errors at this point, you will need to add JUnit as a *module dependency*. Follow these instructions:

a. From the Intellij menu, choose *File - Project Structure*.

b. In the left-hand pane, click *Libraries*, then click the + icon to add a library. Choose *From Maven*, then enter **junit:junit:4.12** into the search bar and press OK.

In the Choose Modules dialog, select the Lab01 project, then press OK twice to close all dialogs. There should now be no errors in the test-case class.

_7. Let's set up both tests to fail so that we can run an initial test:

a. At the top, write an import:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

b. Add the following to each test-case method:

```
fail("Not yet implemented");
```

_8. Now let's try running the test-cases:

a. In the project list, right-click on QuadraticFormulaTests and choose *Run QuadraticFormulaTest*.

IntelliJ runs the tests. Note that the results show two failures out of two tests run. That should make sense, because each of the test-case methods has a fail() call in it.

_9. Next, let's start writing an actual test case:

a. In the *testSolveQuadraticPlus* method, delete the *fail()* call.

b. Still in testSolveQuadraticPlus, test this method's basic function:

```
QuadraticFormula qf = new QuadraticFormula();
double root = qf.solveQuadraticPlus(1, 3, -4);

assertEquals("'Plus' root", 1.0, root, 0.0);
```

Note: The final parameter is the *delta* - the maximum delta between expected and actual for which both numbers are still considered equal to account for floating-point rounding errors. In this case, we specify zero since we expect an exact result.

c. Save the file -- you should have no errors.

d. Run the JUnit test again. You should now have only one failure.

_10. Repeat the above step to complete the testQuadraticMinus method. With the same input values, the "minus" root should be -4.0.

_11. Run the test again: you should have a green bar!

_12. Next, let's see what happens if we modify the code under test and introduce a bug:

a. Edit QuadraticFormula.java.

b. In the solveQuadraticPlus method, "accidentally" change the line that calculates the *inner* variable so it multiplies by five instead of four.

c. Save QuadraticFormula.java, then re-run the JUnit test.

Do you see how having unit tests in place helps identify regression errors?

Part 2: More JUnit

Steps:

_1. Your first job is to import the existing code for which you will write unit tests. Follow these steps:

a. Open Windows Explorer or My Computer and browse to **{Lab Installation Directory}**/starters/lab01.

If your computer has the standard lab setup on Windows Vista or later, the directory is:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
c:\Users\username\javaclass-p2\starters\lab01
```

b. Right-click on the *exam* folder and choose *Copy* to copy the folder to the Windows clipboard.

c. Back in IntelliJ, in the project pane, right-click on the *Lab01/src* folder and choose *Paste*.

_2. In the *src* folder, expand the *exam* package and note the following provided classes:

- **Exam.java** represents a multiple-choice examination or quiz. Note that an Exam consists of *Question*s and a list of *ExamResult*s. Note also that an Exam knows how to calculate the average score for the examination.
- **Question.java** represents a multiple-choice question on the exam. A Question consists of the question text and a list of choices. Questions also keep track of which is the correct answer by storing the correct answer's zero-based index into the list of choices.
- **ExamResult.java** represents a given student's results of taking the exam. It maintains a list of the indexes of the student's choices as well as a reference to the Question list. Note that ExamResult also has a method that has the logic to score the student's result.
- **Student.java** is a simple class that represents a Student at a school. At this point, it just maintains the student's name and ID.

_3. Let's start by creating a separate *source* folder in which you will write your test cases. This technique lets you have multiple packages with the same name in the separate source folder. That's helpful, since it lets your test cases be in the same package as the code under test, but yet be separate on the file system so it's easy to deploy the code under test and exclude the test cases.

In the IntelliJ Project pane, right-click on the Lab01 project and choose *New - Directory* and name the source folder **test-cases**.

Then right-click on the new *test-cases* directory and choose *Mark Directory as - Test Sources Root*.

_4. Next, create your test-case class:

a. In the editor for Exam.java, click on the *Exam* text in the class definition so that the text-entry cursor is within the class name, then press Alt+Enter and choose *Create Test* to start the wizard.

b. On the wizard, at the top, for the Testing Library, click the *JUnit4* selection.

For the *Class name*, accept the default of **ExamTest**.

For the *Generate test methods for* list, put a checkmark next to all of the methods, then press OK. IntelliJ creates the test-case class in the test-cases folder and opens it into the editor.

_5. Examine the generated test-case class:

- It imports the necessary types for JUnit 4 test-cases.
- It defines a POJO (plain old Java object) class for the test-case class.
- It defines testXXXX() methods for each of the three methods you selected in the wizard. Note that each of these test-case methods has the *@Test* annotation.

_6. As you did in the last part of lab, edit each of the test-case methods to include a call to *fail()*.

_7. As you did in the last part of lab, run all of the tests and verify that they all fail.

_8. Next, let's start writing a test case:

a. In the *testAddQuestion* method, delete the *fail()* call.

b. Still in testAddQuestion, test this method's basic function:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Exam e = new Exam();
Question q = new Question();
q.setQuestionText(
  "Which of the following isn't a Java primitive?");
q.addChoice("char");
q.addChoice("bool");
q.addChoice("float");
q.addChoice("double");
q.setCorrectAnswer(1);
e.addQuestion(q);

assertEquals("Question count should be 1", 1,
   e.getQuestionList().size());
```

c. Run the JUnit test again. You should now have only two failures.

_9. Now let's test the addExamResult method:

a. In *testAddExamResult()*, delete the call to *fail()*.

b. Add the following to testAddExamResult. Note that you can copy and paste some of this from *testAddQuestion* that you did in the last step.

```
Exam e = new Exam();
Question q = new Question();
q.setQuestionText(
  "Which of the following isn't a Java primitive?");
q.addChoice("char");
q.addChoice("bool");
q.addChoice("float");
q.addChoice("double");
q.setCorrectAnswer(1);
e.addQuestion(q);

Student s1 = new Student();
s1.setName("Sue Smith");
s1.setStudentID(12);
ExamResult er1 = new ExamResult(s1, e.getQuestionList());
er1.addAnswer(1);
e.addExamResult(er1);

assertEquals("Exam score should be 100", 100.0,
   er1.getScore(), 0.0);
```

c. Run the JUnit test again. You should now have only one failure.

_10. Note that there's a fair amount of redundancy in the two test-case methods. We will address that in a moment, but first, let's test the fact that we should get an exception if we try to get the average score without adding any questions or results to the exam.

Note: Look at the getAverageScore method in the Exam class and note that it will throw an ArithmeticException if the size of the results list is zero.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Follow these steps to test the exception:

a. In *TestExam.java*, write a new method:

```
@Test(expected=ArithmeticException.class)
public void testEmptyResults()
{
    Exam exam = new Exam();
    exam.getAverageScore();
}
```

b. Run the JUnit test again -- you should still have only one failure, but now Eclipse should report that four tests were run.

_11. Next, let's clean up the test-case class to reduce redundancy. To do this, you will write a "set up" method, and in it, create a *test fixture* that has an exam, questions and results. Follow these steps:

a. In TestExam.java, write a field that holds a reference to an Exam:

```
private Exam e;
```

b. Write a new method in TestExam.java:

```
@Before
public void setUp()
{
}
```

After saving, import *org.junit.Before*.

c. Add code to the setUp() method to build the test fixture, copying and pasting from other methods to save typing:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
e = new Exam();
Question q = new Question();
q.setQuestionText(
  "Which of the following isn't a Java primitive?");
q.addChoice("char");
q.addChoice("bool");
q.addChoice("float");
q.addChoice("double");
q.setCorrectAnswer(1);
e.addQuestion(q);

q = new Question();
q.setQuestionText(
 "What is the difference between 'while' and 'do-while' loops?");
q.addChoice(
  "The do-while loop executes the loop at least once");
q.addChoice("The while loop never terminates");
q.addChoice("The do-while loop is deprecated in Java 6");
q.addChoice(
  "The while loop executes an even number of iterations");
q.setCorrectAnswer(0);
e.addQuestion(q);

Student s1 = new Student();
s1.setName("Sue Smith");
s1.setStudentID(12);

Student s2 = new Student();
s2.setName("George Jones");
s2.setStudentID(44);

ExamResult er1 = new ExamResult(s1, e.getQuestionList());
er1.addAnswer(2);
er1.addAnswer(1);
e.addExamResult(er1);

ExamResult er2 = new ExamResult(s2, e.getQuestionList());
er2.addAnswer(1);
er2.addAnswer(0);
e.addExamResult(er2);
```

d. In the testAddQuestion method, delete all of the lines EXCEPT the assertEquals. Edit assertEquals so that the expected value is two instead of one (the setUp method adds two questions).

e. In the testAddExamResult method, delete all of the lines EXCEPT the assertEquals.

   Immediately before the assertEquals, add this line to retrieve the first ExamResult from the test fixture:

```
ExamResult er1 = e.getExamResultList().get(1);
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Then modify the assertEquals so that the expected result is 50.0. Look in the setUp() method to verify that this is the expected result for the second ExamResult added to the Exam.

f. Run the JUnit tests as before. You should have one failure out of the four runs.

_12. Using the techniques you learned in the chapter and in this lab, complete the testGetAverageScore method to test the average exam score that you'd expect given the data in the test fixture.

After completing this part, you should have a green bar indicating that all of the test cases passed.

Optional Part 3: Testing a Policy and Claim Application

In this part, you will practice the techniques you learned in the last part.

Steps:

_1. Using the technique you learned in the last part, import the existing *insurance* package into your project's *src* folder.

Examine the provided code and ensure that you understand it.

_2. Create a test-case class for the Policy class named **TestPolicy** in the *insurance* package in *test-cases* source folder.

In the wizard, select the addClaim, getClaimCount, getTotalClaims and getAverageClaim methods to test, then update those methods in the test-case class to fail.

_3. Write a "set up" method and in it, create a test fixture with sample Policies, Claims and Customers. Define the Policy reference as a field in the test-case class. Be sure to add the Claims to the Policy.

_4. Implement the testGetClaimCount method to verify that the number of Claims in the test fixture match the expected value.

Run the JUnit test and verify correctness so far.

_5. Implement the testAverageClaim method to verify that the average of the claim amounts set up in the test fixture match the expected value.

Note that you will need to write a try-catch block in the test-case method.

Run the JUnit test and verify correctness so far.

_6. As you did in the last part, write an additional test-case method to verify that the testAverageClaim method throws the expected exception if there are no claims. Note that this test-case method should NOT use the text fixture.

Note: This test-case method should throw java.lang.Exception in its definition to avoid compile errors about unhandled exceptions:

```
@Test ( .......... )
public void testEmptyClaims() throws Exception
{
   . . . . .
}
```

Run the JUnit test and verify correctness so far.

_7. Implement the testAddClaim method to verify that one of the claims you added in the test fixture matches a local claim that you create in this method. Your code might look something like (adjust the values to match your test fixture):

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Claim c = new Claim();
c.setAmount(1500);
c.setClaimID(45);
c.setDate(new Date());
assertEquals("Claims should be equal", c, policy.getClaim(45));
```

Run the JUnit test and verify correctness so far. You should have a green bar at this point.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.