# CS 371 Laboratory 8: JUnit

This lab is designed to give you experience with unit testing procedures and tools with emphasis on JUnit in Android Studio.

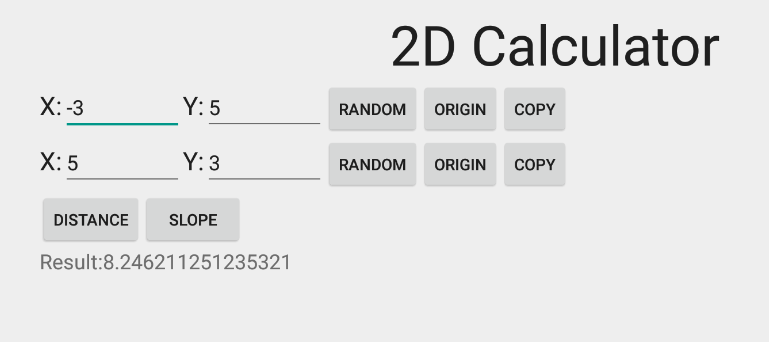
This lab is modified Andrew Nuxoll and Kyle Kearney from a tutorial written by Lauren Darcey and Shane Conder.

## Preliminaries

For this lab, you should work with a partner who is part of your team for the game project. If your team has three members in CS371, work as a team of three.

### Checkout the Starter Code

Clone the Github repository https://github.com/cs301up/CS301JUnitStarter by downloading the zip, extracting it, and then loading the project into Android Studio. You may get asked to update Gradle to work with this project. Go ahead and do so.



### Run DimCalc

Verify that the DimCalc app runs correctly in Android Studio and that you understand its use.

The app lets you enter two 2D Cartesian coordinates. You can type these in directly, generate them randomly, set them to origin or copy the values from one coordinate to the other.

Once both coordinates are entered, you can calculate the distance between the two points or calculate the slope of a line that passes through the two points.

This application has bugs. Perhaps you can discover some of them already? If so, **don't fix them just yet**.

### Examine the Java Source Code

The Java source code for DimCalc consists of two files:

* TwoPoints.java is a class that is designed to perform operations on two points.
* DimCalc.java contains an Android app that interacts with the user. It uses an instance of the TwoPoints class to store the users input and calculate the values that the user requests.

If you are familiar with the model-view-controller design pattern, then you should recognize that TwoPoints.java contains the model and DimCalc.java is both the view and controller.

## Laboratory

### Part 1A: Create a Unit Test Class for TwoPoints.java

TwoPoints.java is easier to unit test because it doesn't use a GUI. You can simply create an instance of the class and see if it behaves as you expect. There is no need to even run such a test on an Android device. To setup a unit test for TwoPoints.java follow these steps:

1. Add JUnit4 support to your build.gradle file
   1. In the project window (left-hand pane), set the dropdown at the top of the pane to show “Project” view. Under the app directory, open build.gradle. **Important**: do *not* edit the other build.gradle file in the project.
   2. In the dependencies section, add this line:

testImplementation 'junit:junit:4.+'

* 1. A prompt will appear near the top of your Android Studio window asking you to sync gradle with your project. Select the Sync Now link.

1. Setup a directory structure for your unit tests
   1. Open a Windows explorer file browser. Locate your project's app/src folder within the directory where you cloned your project. This folder should already contain folders named androidTest and main. Add a third folder named test.
   2. Create a subfolder of the test folder you just created that is named java.
   3. Switch back to Android Studio. With your dropdown set to “Project Files,” navigate the structure to see if the test and java folders you just created are there. If not, select File→Reload All From Disk.
2. Create the unit test file
   1. Open the TwoPoints.java file so you can see the source code.
   2. Right click on the code editor window (not the project hierarchy pane on the left) and select Go To -> Test. When prompted. Select Create New Test. A Create Test dialog will appear.
   3. Select JUnit4 as the Testing library.
   4. If the dialog states that JUnit4 library is not found, click the Fix button.
   5. The bottom of the dialog asks you which methods you want to write test methods for. Check all the methods that are listed.
   6. Leave all other fields at their default values.
   7. Click OK. If you are asked to select a destination directory, choose “../app/src/test/java/up/edu/dimcalc”.
   8. You may be asked if you want to add the new file to your git repository. You probably want to do this.
   9. The new test file should be loaded for you to edit. It will contain a TwoPointsTest class with several empty methods.
3. Run the unit test file.
   1. Right click on the class declaration, TwoPointsTest, in the source code and select Run ‘TwoPointsTest’. It may take a few moments for the test results to appear.
   2. All tests should pass (since they aren't testing anything yet!)

### Part 1B: Create Successful and Failed Unit Tests

Below is the code for the first two tests (you will also need to import the Point class). Copy and paste this code into your class, replacing the existing, empty methods by the same name. Re-run the tests to verify that these real tests pass.

/\*\* when created, getPoint() should show both points at the origin \*/

@Test

public void getPoint() throws Exception {

TwoPoints testPoints = new TwoPoints();

Point p1 = testPoints.getPoint(0);

Point p2 = testPoints.getPoint(1);

assertEquals(0, p1.x);

assertEquals(0, p1.y);

assertEquals(0, p2.x);

assertEquals(0, p2.y);

}

/\*\* verify that arbitrary values are properly stored via setPoint() \*/

@Test

public void setPoint() throws Exception {

TwoPoints testPoints = new TwoPoints();

testPoints.setPoint(0, 5, -3);

testPoints.setPoint(1, -3, 5);

Point p1 = testPoints.getPoint(0);

Point p2 = testPoints.getPoint(1);

assertEquals(5, p1.x);

assertEquals(-3, p1.y);

assertEquals(-3, p2.x);

assertEquals(5, p2.y);

}

Take a screenshot of the passing tests, including the code you pasted in.

Make a minor edit to one of the tests so that it will fail. Then run the tests again and see how JUnit reports failure.

**checkpoint 1 (20 points): Show your instructor the passing and failed test results. After you’ve been checked off, return the test to normal so that it will pass.**

### Part 2: Create your own Unit Tests

Note, you can run a single test method by right-clicking on the method and selecting Run <name of method>. This may save a bit of time as you’re testing the rest of the methods for this checkpoint.

Implement the remaining unit test methods in TwoPointsTest. Each method should create an instance of the TwoPoints class, call the relevant method at least once and then verify that the class behaves as it should as a result of calling that method.

As you are writing these unit tests, you should discover at least two bugs in the behavior of TwoPoints. Be sure to write your unit tests so that these bugs are detected. **Do not fix the bugs** but examine the source code and determine how you will fix them.

If you are having trouble locating bugs, make sure to carefully read the comments that document the expected behavior of the methods that you are testing.

**checkpoint 2 (30 points): Have your instructor or lab assistant verify that at least two unit tests fail due to bugs in the source code. Explain what the bugs are and how you will fix them.**

### Part 3: Fix the Bugs

Fix the bugs you discovered in the previous checkpoint and verify that all tests pass.

**checkpoint 3 (10 points): Show your instructor or lab assistant that all tests pass.**

### Part 4: Create Unit Tests for your Game State

Close the DimCalc project and load the project you’ve created for your game (if you and your partner are not on the same game project team, pick either game). Write at least three thorough unit tests per person for your game state class.

**checkpoint 4 (40 points): Show your instructor or lab assistant the tests that you’ve created. Explain why they are effective and demonstrate that all tests pass.**

Make sure to push your changes to Github (particularly the tests that you wrote for your game state).