Instructions for Lab Exercise 2

Debugging with Eclipse

CS 2334, Spring 2016

Due by: Friday, January 29, 2016, 4:00 pm CST

This lab is a group exercise. Students must complete this assignment with at least one partner.

Learning Objectives (Milestones):

1. Create and use JUnit tests to debug a sample Java program.

Inspiration:

Have you ever made a million dollar mistake? How about a three hundred million dollar mistake? A NASA contractor has. Because the contractor used the wrong units of measurement, the Mars Climate Orbiter was lost at a total mission cost of over $325,000,000. We should try to avoid making such costly mistakes ourselves.

One way to reduce the likelihood of such mistakes is to represent measurements as objects, rather than just numbers. Think of it this way, what if I asked you, how far is 12 or how fast is 20? These are nonsensical questions because 12 is not a distance and 20 is not a speed. To have a distance, we need to know the distance units we are using. 12 centimeters is a very different distance than 12 inches, 12 feet, 12 meters, 12 yards, 12 miles, 12 light years, or 12 parsecs. Similarly, 20 miles per hour is a much different speed than 20 meters per second. You wouldn't want someone to assume you meant inches when you really meant centimeters or to assume you meant miles per hour when you really meant meters per second any more than you would want someone to think that a parsec is a unit of time. So, rather than representing a distance measurement as an `int` or a `double`, we could create a **Distance** object that keeps track of the numerical value (such as 12) along with the unit (such as inches). Today's lab does just that.

Unfortunately, dealing with lots of different measurement types (distances, speeds, times, accelerations) and many different unit possibilities for each type, results in a lot of code. Fortunately, using object oriented approaches can simplify the design and implementation of that code. Unfortunately, it is nonetheless very possible to wind up with errors in all that code. Fortunately, there is a method that can help with finding and fixing such errors – unit testing.

When we talk of "unit testing," we are using the word "unit" in a different sense than we did with units of measurement. With unit testing, we are referring to small parts of our code as "units." We test each of those small parts independently, to try to make sure that it works correctly on its own, rather than trying to debug a large program all at once. In object-oriented unit testing, we test each Class separately and within each Class, we test each method of any complexity. For unit testing in Java, JUnit is very popular and important unit testing framework. It is also the framework we will use for this lab and in your projects.

In this lab, we are going to crudely simulate a very simple spaceship that moves in a straight line.

Instructions:

This lab exercise *requires your group to have a laptop with an Internet connection*. Your group will work collaboriately to complete the exercises described in these instructions. Some exercises will require you to fill in the associated answer sheet with answers while others will require you to work on a Java project in Eclipse. You will submit a PDF copy of the completed answer sheet as well as an Eclipse archive of the completed Java project.

1. Download and save the answer sheet (`Lab2-CS2334-answerSheet.odt`) and fill in the names of all group members where indicated at the top of that document.

2. Download and save the sample `Lab2-eclipse.zip` source archive from the class website. Import the archive into Eclipse using the instructions given in the "Basic Eclipse Tutorial" slides.

3. Using Eclipse's Package Explorer, take a look at all of the Classes in the Lab 2 project. You should see 12 of them there. In Java, each Class is stored in its own file. List the 12 Classes in the space indicated on the answer sheet.

4. Open the **SpaceshipDriver** Class and look at its source code. All Java programs must have a `main` method within them so that the JVM knows where to start running the code. (Even though Java is object oriented, at the hardware level the computer is procedural, so the computer will still execute the code step by step by step.) As you can see, the **SpaceshipDriver** Class contains the `main` method for this program. Using "Driver" at the end of the name of the Class that contains the `main` method is a common coding convention.

5. Within the main method, **SpaceshipDriver** creates a new spaceship model of Class **SpaceshipModel** and initializes its speed to zero meters per second. (Of course, the code for the Class **SpaceshipModel** is in the file `SpaceshipModel.java`.) On the answer sheet, list the name of the variable that refers to the new spaceship model created here.

6. Next, **SpaceshipDriver** gives a sequence of speed commands to the ship and, finally, it calculates some values and prints them out. Run **SpaceshipDriver** and list its output on your answer sheet.

7. Obviously, there are errors in this project, which is why you are using it to practice debugging. But what kinds of errors? Are there compilation errors? Runtime errors? Logic errors? How can you tell? Fill in your answers to the question "How can you tell?" on your answer sheet.

8. One of the errors that should be apparent is that the minimum acceleration reported is "null." That can't be right. Looking at SpaceshipDriver, you can see that line reporting the minimum acceleration is this one:

```
System.out.println("Minimum acceleration encountered will be " +
ship.calculateMinAcceleration());
```

Highlight `calculateMinAcceleration` then right (or control) click on it. On the menu that appears, select "Open Declaration." In the Search window below (on a tab next to Console), you'll see that `calculateMinAcceleration` was declared in **SpaceshipModel**. Double click on `calculateMinAcceleration` in the Search window and Eclipse will jump you to the declaration of `calculateMinAcceleration` in your code window. Answer the following question on your answer sheet: Why is the minimum acceleration being reported as "null"?

9. Before you write code to correctly calculate the minimum acceleration, note that the maximum acceleration reported can't be right either. Answer the following questions on your answer sheet: What is the reported value and why must it be wrong?

10. Now, you could try to fix the code so that it correctly reports the maximum acceleration just by hunting around the code until you find problems. But where? Is the error in `calculateMaxAcceleration` or is it in one of the methods called by `calculateMaxAcceleration`, such as the static helper method used to calculate each acceleration based on speed and time values `calcAccelerationFromSpeedsAndTimes`? It isn't clear, so we'll try to be systematic by using JUnit.

First, we can't expect `calculateMaxAcceleration` to get the right answer if its helper methods don't give it good values, so we'll start with the helper method `calcAccelerationFromSpeedsAndTimes`, which is a static method

from the **PhysicsCalculator** class.

So, open up **PhysicsCalculator** and look inside it. There you will see methods for calculating distance, speed, and acceleration based on other physical quantities. Rather than looking at `calcAccelerationFromSpeedsAndTimes` to see if it looks right, we're going to test it. Highlight **PhysicsCalculator** in the Package Explorer and right (or control) click on it. On the menu that appears, select New → JUnit Test Case and create a JUnit 4 test case for the `calcAccelerationFromSpeedsAndTimes` method. Answer the following question on your answer sheet: What did Eclipse create when you clicked "Finish"?

11. Replace the placeholder line (marked with the comment TODO) from the test with a line that would help you to see if `calcAccelerationFromSpeedsAndTimes` is working correctly. For example:

```
Speed speed1 = new Speed(0);
Speed speed2 = new Speed(10);
Time time1 = new Time(0);
Time time2 = new Time(1);
Acceleration calculatedAcceleration =
  PhysicsCalculator.calcAccelerationFromSpeedsAndTimes (speed1, speed2, time1, time2);
assertEquals(10.0, calculatedAcceleration.getValue(), 0.0);
```

This code creates two speeds and two times. Given a speed of 0 m/s at time 1 and a speed of 10 m/s at time 2, and given that these times are one second apart, the acceleration should be 10 m/s^2. So, we assert that the value calculated by the **PhysicsCalculator** method `calcAccelerationFromSpeedsAndTimes` will be 10 (with a margin of error of 0.0).

Save the file you are editing, then select it in the Package Explorer and right (control) click it and, from the menu that appears, select RunsAs → JUnit Test. Answer the following questions on your answer sheet: What was the result and what does that tell you about the method `calcAccelerationFromSpeedsAndTimes`?

12. Fix the problem with `calcAccelerationFromSpeedsAndTimes`. Explain on your answer sheet what you did to fix it.

13. BONUS POINTS. There are several other errors in the file. Use JUnit to find at least one of them and explain on your answer sheet how you used JUnit to find that out.

14. Export your completed answer sheet to PDF and submit it to the dropbox for Lab 2.

15. Export your Eclipse project file to an Eclipse archive (like the one you imported for this lab) and submit it to the dropbox for Lab 2.