CPSC 2151

Lab 8

Due: Friday October 26th at 11:59 pm

In this lab you will be creating a directional speedster class, providing some functionality for it, and creating JUnit test cases for that class.

Instructions

Start off by creating a new project with a package called cpsc2150.Speedster. Then add the provided IDirectionalSpeedster interface (provided for you) to your cpsc2150.Speedster Package. This Interface. will help us track the time, distance and speed of our travel. However, unlike the example given in class, we must now consider the direction of travel. We can do that by imagining an x,y coordinate plane. If we start at the origin (0,0) we can define movement in any direction by looking at the change in x and the change in y. Those changes can be positive or negative, to allow us to move in any direction. If we have moved from point (x1, y1) to point (x2, y2) then our distance = sqrt((x2 – x1)^2 + (y2 – y1)^2).

Our IDirectionalSpeedster interface needs to keep track of the total distance travelled, the total time of travel, the net distance travelled, the net velocity travelled, and the average velocity travelled. What's the difference between total distance and net distance? Imagine that we start at (0,0) and move 4 places to the right, putting us in position (4, 0). Then we move 2 spaces to the left to position (2, 0). Our total distance travelled was 6, but our net distance travelled was 2, since at the end of our travel we were only 2 spaces away from where we started. The difference is the same when we compare average velocity to net velocity. When we calculate average velocity, we look at the total distance travelled divided by the total time, and for net velocity we look at the net distance travelled divided by the total time.

Next you should provide a class called DirectionalSpeedster that implements the provided interface. There are many ways that this class could be represented in our private data. Feel free to use whichever makes the most sense to you. The contracts in the interface give you this flexibility by enforcing rules in the constraints and not just in the post conditions. If you can implement a method in the interface as a default method, feel free to do so. Otherwise no other changes should be made to the interface. Make sure to include comments, correspondences and invariants in your class.

Remember, you will need to provide a constructor for your class as well.

Constructor: DirectionalSpeedster()

This constructor takes in no arguments and sets all values to 0, *including time*.

DirectionalSpeedsterTest class

Use the instructions from last week's lab to set up your test folder in IntelliJ and create a class called DirectionalSpeedserTest.java. You will create your JUnit test cases in this class. Remember to set up your build configuration to run JUnit test cases (instructions in Lab 7).

We will make several test cases for this class. Due to our class having private data, and limited public functionality, we may need to have more than one function call in each test case. That's OK, and is often necessary, but we still want to limit each function in our JUnit test class to be one test case. For instance, in order to see if addTravel was successful, we need to check to see if the current X position and Current Y position have changed, so we will need to call getCurXPos and getCurYPos. We'll just have

to remember that if the test case fails, there could be an issue with any of those functions and we would need to check them all for potential faults. It may be helpful to separate our test case into two functions that both use the same input, but one checks if the total distance is correct, and one checks if the total time is correct. This would limit the places in code that our fault could be if only one of the test cases fails. You don't have to do that for this assignment, but it may be helpful if you are struggling to find the cause of the failure.

For each test case, we should still only call the function we are testing once, even if we will need to call other functions to see if the test case was successful. The only exception to this is the addTravel method. If we have a test case that involves travel having occurred in the past, we will need to call addTravel more than once to set up our test case. But we will only call our assert statements after the last addTravel call.

Even though we will discuss Path testing in class before most of you have this lab, you do not *need* to use that methods for identifying test cases for this assignment.

Make sure to follow our best practices for JUnit such as naming our functions appropriately. Remember that best practices for JUnit and best practices for code are different. You are allowed to use magic numbers in JUnit to hard code our inputs and expected outputs.

Testing the Constructor

You will need to test the constructor of the DirectionalSpeedster class. The constructor doesn't take in any arguments, so we really only have one test case. However, we do need to check to make sure that the distance, time and current position on the grid are correct. You could separate these into different test cases with the same input if you want.

Testing addTravel

Create 10 unique and interesting test cases for the addTravel function. Make sure your test cases are not really testing the same concept, just with slightly different numbers. Consider different challenging situations for the programmer. For each test case include a comment explaining why you chose that test case. At this point I would only use the getCurXPos and getCurYPos functions to verify that this function is working correctly. Remember, even though we may call addTravel more than once to set up our test case, we should only call our assert statements after the last addTravel call.

Testing getTotalDistance()

Create five unique and interesting test cases for the getTotalDistance function. Make sure your test cases are not really testing the same concept, just with slightly different numbers. Consider different challenging situations for the programmer. For each test case include a comment explaining why you chose that test case. You will need to call addTravel multiple times to have an interesting test case, so if your test case fails make sure that addTravel is working as intended as well. Remember, since we are testing getTotalDistance, we should only call getTotalDistance one time.

Testing getNetDistance()

Create five unique and interesting test cases for the getNetDistance function. Make sure your test cases are not really testing the same concept, just with slightly different numbers. Consider different challenging situations for the programmer. For each test case include a comment explaining why you chose that test case. You will need to call addTravel multiple times to have an interesting test

case, so if your test case fails make sure that addTravel is working as intended as well. Remember, since we are testing getNetDistance, we should only call getNetDistance one time.

Testing getAverageSpeed()

Create five unique and interesting test cases for the getAverageVelocity function. Make sure your test cases are not really testing the same concept, just with slightly different numbers. Consider different challenging situations for the programmer. For each test case include a comment explaining why you chose that test case. You will need to call addTravel multiple times to have an interesting test case, so if your test case fails make sure that addTravel is working as intended as well. Remember, since we are testing getAverageVelocity, we should only call getAverageVelocity one time.

Testing getNetSpeed()

Create five unique and interesting test cases for the getNetVelocity function. Make sure your test cases are not really testing the same concept, just with slightly different numbers. Consider different challenging situations for the programmer. For each test case include a comment explaining why you chose that test case. You will need to call addTravel multiple times to have an interesting test case, so if your test case fails make sure that addTravel is working as intended as well. Remember, since we are testing getNetVelocity, we should only call getNetVelocity one time.

Running your code on Unix

How to run JUnit code on unix was covered in the slides. Make sure your code will run in Unix. You must create a makefile that will compile you code with the "make" command and run the JUnit test cases with the "make test" command.

TIPS and additional Requirements

- In JUnit 4 assertEquals(double actual, double expected) has been replaced with assertEquals(double actual, double expected, double epsilon). This handles the fact that we lose precision with double values at very small decimal places. We could expect a function to return 2.0 and it actually would return 2.00000000000001, which would be marked as a failed test case. We can use epsilon to define an acceptable error, so assertEquals will return true as long as abs(actual – expected) < epsilon. You can define a constant value to use as epsilon
- It will be difficult to test our function calls individually, since we have private data. It is ok to have more than one function call to properly test your code. Just be aware of that when you see a failed test case. However, a test case should still only have one input set and one final expected output, and should only call the tested function once (except for addTravel).
- Inputs that are interesting and unique for one function may be interesting and unique for another function. It is ok to reuse inputs to test different functions as long as they are good test cases for both functions.
- Don't get caught up on the contracts we wrote for the Speedster class during lecture. Adding in the directional functionality actually has a big impact on what is and is not allowed in the contracts.
- Note that Speed, time, and velocity are all allowed to be zero with this interface. Someone can call getAverageSpeed before adding travel, and should get a zero in return.

- Remember, our best practices are different for JUnit and for normal code. Make sure to follow the appropriate best practices.

Groups

You may, but are not required to, work with a partner on this lab. Your partner must be in the same lab section as you, not just the same lecture section. If you work with a partner, only one person should submit the assignment. You should put the names of both partners in a comment at the top of the file in order for both partners to get credit. This assignment may take more than just the lab time. Make sure you are able to meet outside of class to work on the assignment before you decide to work with someone else. Remember to actively collaborate and communicate with your partner. Trying to just divide up the work evenly will be problematic.

Before Submitting

You need to make sure your code will run on Unix with the makefile you have provided. Make sure your package directory is set up correctly. The TA should be able to unzip your code and enter the commands “make” and “make test” to run your code.

Submitting your file

You will submit your files using handin in the lab section you are enrolled in. If you are unfamiliar with handin, more information is available at https://handin.cs.clemson.edu/help/students/