Lab 09 C212 FA22

Last modified 2022-10-26 11:59 PM

Summary

The goals of this lab are as follows:

- Part A - Introduction to the JUnit Unit Testing Framework
    - Adding JUnit to a project
    - Creating test folders and files
    - Writing JUnit unit tests
- Part B - Inheritance
    - From a Circle to more tasty types of circles

Part A - Introduction to the JUnit Unit Testing Framework

If you look at the rubric of Assignment 6, you'll notice testing makes up equally as many points as the code itself. Testing is important! But, if you've been writing the tests for A6 (hopefully you have, it's due pretty soon), you'll quickly notice how tedious it can get. We said testing was important, but we didn't say it was fun. However, with how widespread testing is in the wide world of software development, it's no surprise that tools have been designed to make testing better. You didn't think that comparing outputs in the console would be the ultimate form of testing, did you?

Unit testing is the process where individual pieces of a program – the units – are tested individually. It's the lowest-level of the different stages of test suites, but perhaps the most important: if your program is going to fail on unit testing, it'll probably fail on later levels of testing, too, and by catching it in unit testing, it's much easier to pinpoint and fix.

JUnit is a popular Java unit testing framework, and will be the format of our testing for the remainder of the semester. It's not officially part of Java, but rather a third-party library, or collection of classes made to be used by other developers (that's you!). It's integrated into IntelliJ, which makes it pretty easy to set-up and use.

Start. For this week's lab, start with a new project and add the starter code file Circle.java to the src folder. You'l notice this is similar to what we did for Lab08.

Create Test Folder. Next, we're going to create a new folder in our project, separate from src, which will hold all of our files for testing. It's not strictly necessary to separate source code from test files, but it's good to keep things organized in this way, and it's not difficult to do anyway. Right-click (or Control-click on Mac) on the Lab09 folder from the left menu in IntelliJ and select New > Directory. Name it test. Then, right-click on that folder, and select Mark Directory As > Test Sources Root. It'll now be green, in contrast to src's usual blue color.

Create Test File. Next, we'll have IntelliJ create a test file for us. We do have a sample testing file created for you, CircleTest.java, but don't use it yet. Each time you go to create your own tests, you'll have to follow this process: With Circle.java open in IntelliJ, right-click anywhere inside the class and click *Generate…* Then, select *Test…* A menu will appear, with the first dropdown being the testing library. *JUnit 5* should be selected by default, but below it there will be a warning saying the library isn't found in the module. Hit *Fix*, which will open yet another menu. There should be something typed into the box for you, along the lines of **org.junit.jupiter:junit-jupiter:5.8.1**. Hit OK, and it'll add JUnit to your project. Then, back in the Create Test menu, we have some more options to choose from. You can leave its name as CircleTest (remember we'll overwrite this file with the starter code eventually), but near the bottom there should be some check boxes labeled with the methods in Circle.java. Select at least one of these and hit OK.

Run Test File. Now, a file should've been created for you and added to the test folder. Inside, there'll be one or many void methods prefixed with @Test. @Test is what's called an Annotation in Java. They don't do anything in the code, but are treated as metadata about the file. You'll also notice some import statements at the top of the file, importing the necessary JUnit classes. Assuming there are no errors at this point, click the green play button to the left of the class header to run all tests, or the one to the left of any method to run that specific method.

If you're having any trouble with IntelliJ adding JUnit to the project, here are the steps to do it manually: Select File > Project Structure > Libraries > New Project Library *(the '+' in the second column of the menu)* > From Maven > *Type* org.junit.jupiter:junit-jupiter:5.8.1 *into the search bar, and hit OK. You can also try version 5.6.0 if that fails.*

What this did was create a new test file with one or many test methods, depending on how many of Circle's methods you checked in the Create Test menu. Test methods are how your tests are grouped. There should be one test file for each source file you want

to test (like how we're making CircleTest to test Circle), but the number of test methods doesn't need to match the number of methods in the program you're testing. Doing this can be a way of making sure you test every method from the class, but it's usually just kind of excessive to test only one method per tester method, and you're better off writing your own test methods which you use to group tests in a more logical way.

Something to watch out for if you ever do use these pre-generated tester methods is that their name will actually match exactly that of the method it's made to test. This means, if you're wanting to test the hypothetical getPi() static method from Circle inside of the tester method also called getPi(), the compiler will think you meant the tester method of the same name and end up in an infinite recursion. To fix this, either rename the tester method from its default or specify Circle.getPi().

Moving on, replace whatever file you just made with CircleTest from the starter code on Canvas. This has all the testing needed for Circle. Can you run this and see all the tests pass? If not, get help from your TA before moving on. If so, congratulations, you now know how to set up JUnit for any project.

All we have left is to understand how to write tests in JUnit. JUnit testing is based on **assertions**. Like the name implies, these are things we declare will be true, because if they aren't, the test will fail. Look at the provided CircleTest class and notice its use of JUnit methods like assertEquals, assertTrue, and assertThrows. These are just some of the assertion types, but these three will get you pretty far. You can use the code-completion after typing assert to see more. assertEquals takes two arguments, an expected and an actual, which you can use to compare some predefined values with the results of your code. It works well for primitive types and Strings. assertTrue just take one expression and check its result. assertThrows is weirder to use. First, it takes the kind of Exception, but it has to be an expression, so you'd use, in the case of IllegalArgumentException, `IllegalArgumentException.class`. Next, it takes an executable, or some block of code that can be run. This is different from a method! We can get in the habit of giving it a lambda expression, which is what the `() -> {...}` is doing. Just throw in the code you want to test into the braces.

You now know the basics of unit testing with JUnit. The main benefits are in how it speeds up our testing – we don't need to manually check the output of print statements anymore – and with the kinds of things it lets us test – assertThrows is great for testing Exceptions.

Last but not least, please keep some things in mind when you write your own tests. First, you shouldn't write tests that are supposed to fail. JUnit gives you a range of test

methods, each with their own opposites, like assertTrue and assertFalse, and assertThrows and assertDoesNotThrow. Use the appropriate JUnit method for what you want to test. Second, when working on any assignment or lab, do NOT (do NOT!) fake a passing test case. For example, if I knew my getTwoPlusTwo() method was returning 5 instead of 4 and my tests were failing, I should NOT change the expected result in the test case to 5 just so that the test appears to pass. This is arguably academic dishonesty (but we're not going to hunt you down for this, plus you might've just genuinely expected the value to be something else). Instead, you'd just lose points twice for the method being wrong and for the test being wrong, instead of not lying and only losing points for the method being wrong.

Part B - Inheritance

If you subscribe to the "four pillars of OOP" as a way of describing object oriented programming, then **inheritance** is right up there with abstraction, encapsulation, and polymorphism as some of the defining features of the paradigm. Inheritance is a way of setting up "is-a-type-of" relationships between classes. The classic example of this is an Employee and Person, where Employee is a type of Person, but not the opposite. If we give Person some instance variables like first and last name and date of birth and maybe a method getFullName(), inheritance means that Employee will also have those same members that work the same way automatically. This is because the **subclass** Employee inherits it from the **superclass** Person. These are also known as child and parent classes.

Where things get interesting is when a subclass needs to override a superclass in some way. Though you didn't fully understand it yet, we've already been doing this when we override toString on a class we create, because toString is actually inherited from the Object class in Java, which is the ultimate parent class of all classes.

Task 1. Circle.java is good, now we're going to create a subclass of Circle called Pizza. To subclass another class in Java, write extends NameOfSuperclass in the method header, like this:

```
public class Pizza extends Circle
```

We want our Pizza to have a radius and a topping. Again, due to inheritance, Pizza will automatically have the instance variable radius and the getters and setters for radius. We need to give it a constructor, which takes in a radius and topping String. When we write constructors for subclasses, we first need to include in the constructor's body a call to the superclass's constructor. Remember, Circle needs a radius in its constructor, and we give that to Circle from Pizza's constructor with `super(radius);`. Just as we can refer to the current class with `this`, we can refer to the parent class with `super`, and it

just so happens that since constructors in Java have the same name as the class itself, `super()` refers to the constructor method of the parent class.

For this task, add getters and setters for topping to Pizza. In Circle.java's comments, there's a really handy tip for auto-adding getters and setters in IntelliJ. Also, add a method getPrice() to Pizza which returns the price of the pizza, which is $0.15 times the area of the pizza (remember, we refer to pizzas in common conversation by their diameter, not radius). Do not re-calculate the area here – use this class's inherited getArea() method!

Task 2. Write a test class for Pizza called PizzaTest using JUnit. Within, test everything from Circle AND Pizza, to ensure that Pizza inherits everything from Circle as expected. Your tests here should include calls to assertEquals, assertTrue, and assertThrows at least once each. Use assertEquals to test getRadius(), assertTrue to test getTopping(), and assertThrows to test creating a pizza of negative size.

Task 3. Create a subclass of Pizza called DeepDishPizza. It is distinct from its superclass in only one way: it should override the getPrice() method to charge $0.25 per square unit of pizza instead of $0.15. Write a tester class for this class using JUnit to ensure the price is calculated correctly.

Task 4. Think about these two things and answer the questions asked in comments in DeepDishPizza.java:

- If we wanted to create a square pizza, we could theoretically subclass Pizza to make a class SquarePizza, and override its getArea method to instead return radius * radius, or we could add a new instance variable sideLength and return the square of that for getArea. Nothing seems immediately wrong with this – I mean, a SquarePizza is a type of Pizza, right? – and we could do it in Java. However, this violates the philosophy of object-oriented programming. The "is-a-type-of" relationship should work through all levels of the hierarchy. This requirement for OOP is known as the Liskov substitution principle, and is the "L" in the five-letter acronym of another competing definition of OOP besides the "four pillars". What is this five-letter acronym?
- If I wrote a method that takes in an argument of type Circle, could I call that method and pass in an instance of Pizza or DeepDishPizza? You might not know the answer yet – we'll learn this soon – but do you think that would work? Why or why not? (Hint: what is the overlap between what a DeepDishPizza has and what a Circle has?)