Lab 5: Classes, Methods, Loops and Satisfying Unit Tests

Let's get started!

Today we'd like you to:

1. **Open Eclipse**. Remember to keep all your projects in the same workspace.
2. **Create a new Java project**. Call it Comp1510Lab05*LastNameFirstInitial*
3. **Complete the following tasks**. Remember you can right-click the src file in your lab 5 project to quickly create a new Java class.
4. When you have completed the exercises, **show them to your lab instructor**
5. **Modular vs non-modular projects**. The Comp 1510 baseline assumption is you will be using non-modular projects. For non modular, do *not* have a module-info.java file and add any required libraries under the Classpath section (for modular projects, do have a module-info.java file and add any required libraries to the Modulepath).

What will you DO in this lab?

In this lab, you will:

1. Design and implement some more classes and their methods
2. Encapsulate your objects by using visibility modifiers and getters/setters
3. Write some more methods that accept parameters and return values
4. Implement constructors that initialize objects in logical and organized ways
5. Evaluate Boolean expressions
6. Make choices using the if and if-else statements
7. Write code that loops using the while statement.

Table of Contents

1. Sphere

Geometry and math in general, is fun (for many software developers). Software developers don't need to be geometers (that's what we call mathematicians who specialize in geometry) but we should be familiar with the basics. Here is the first of some geometry concepts for you to implement:

1. Create a record called **Sphere** inside package ca.bcit.comp1510.lab05:
    a. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
        i. The name of the class and a (very) short description
        ii. An @author tag followed by your name
        iii. An @version tag followed by the version number.
        iv. @param tags for each of the record parameters.
    b. Do not include a **main method** inside the class definition. Remember the main method is what gets executed first when we start our program. In this case, the Sphere class is not a complete program. It will simply be used to represent a Sphere concept. We will create our main method in a separate Driver class.
2. A Sphere is defined mathematically as *the set of points in 3D space that are all at the same distance r (radius) from a given point*. This suggests that a Sphere should have instance variables that represent the following:
    a. X-coordinate
    b. Y-coordinate
    c. Z-coordinate [1]
    d. Radius.
3. Create a method that returns the **surface area** of the Sphere. The formula for the surface area A of a sphere of radius r is $A = 4\pi r^2$.
4. Create a method that returns the **volume** of the Sphere. The formula for the volume, V, of a sphere of radius r is $V = \frac{4}{3}\pi r^3$.
5. Create a **toString()** method which returns a String composed of the concatenation of the information in the Sphere. Customarily the toString() is the last method in the class.

2. Cube

For a cube centered at the origin, with edges parallel to the axes and with an edge length of 1, the Cartesian coordinates of the vertices are (±0.5, ±0.5, ±0.5), and the interior consists of all points (x, y, z) in the range [-0.5, 0.5]. Can you draw this on a piece of paper?

1. Create a new record called **Cube** inside package ca.bcit.comp1510.lab05:

[1] Visit https://en.wikipedia.org/wiki/Cartesian_coordinate_system#Three_dimensions to learn about Cartesian (3D) coordinates, if you do not recall this.

   a. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
      i. The name of the class and a (very) short description
      ii. An @author tag followed by your name
      iii. An @version tag followed by the version number.
      iv. @param tags for each of the record parameters.
   b. Do not include a **main method** inside the class definition. We will create our main method in a separate Driver class.
2. A Cube should have instance variables that represent the following:
   a. X-coordinate
   b. Y-coordinate
   c. Z-coordinate
   d. Edge length.
3. Create a **toString()** method which returns a String composed of the concatenation of the information in the Cube.
4. Create a method that returns the **surface area** of the Cube. The formula for the surface area A of a cube of edge length a $= 6a^2$.
5. Create a method that returns the **volume** of the Cube. The formula for the volume V of a Cube of edge length a $= a^3$.
6. Create a method that returns the **face diagonal** of the Cube. The formula for the face diagonal F of a Cube of edge length a $= \sqrt{2}a$.
7. Create a method that returns the **space diagonal** of the Cube. The formula for the space diagonal S of a Cube of edge length a $= \sqrt{3}a$.

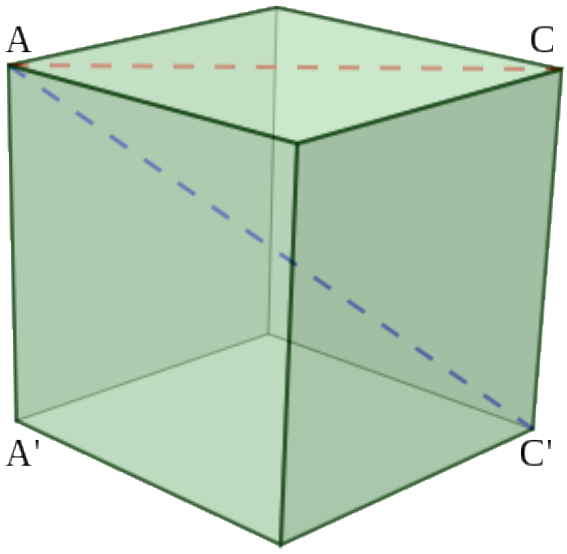

Figure 1 Cube Diagonals. AC' (shown in blue) is a space diagonal while AC (shown in red) is a face diagonal. [2]

[2] By R. A. Nonenmacher [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 4.0-3.0-2.5-2.0-1.0 (https://creativecommons.org/licenses/by-sa/4.0-3.0-2.5-2.0-1.0)], via Wikimedia Commons

3. Cone

Cones come in a few varieties, and we will consider the right circular cone. A cone with a circular base is a circular cone. A circular cone whose axis is perpendicular to the base is a right circular cone.

1. Create a new record called **Cone** inside package ca.bcit.comp1510.lab05:
    a. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
        i. The name of the class and a (very) short description
        ii. An @author tag followed by your name
        iii. An @version tag followed by the version number.
        iv. @param tags for each of the record parameters.
2. Do not include a **main method** inside the class definition. As before, the class is not a complete program.
3. Let's not worry about a Cone's location in 3D space (yet!). If we ignore the right circular cone's location, we can represent it with two variables: radius and height. Define record parameters for its radius and height.
4. Create a method that returns the **volume** of the Cone. The formula for the volume V of a cone of radius r and height h $= \frac{1}{3}\pi r^2 h$.
5. Create a method that returns the **slant height** of the Cone. The formula for the slant height SH of a Cone of radius r and height h $= \sqrt{r^2 + h^2}$.
6. Create a method that returns the **surface area** of the Cone. The formula for the surface area A of a Cone of radius r and height h $= \pi r^2 + \pi r(\sqrt{r^2 + h^2})$
7. Create a **toString**() method which returns a String composed of the concatenation of the information in the Cone.

4. Geometry Driver

We've created three simple shape classes. Now let's practice using the Scanner to interact with the user and create some Shapes:

1. Create a new class called **GeometryDriver** inside package ca.bcit.comp1510.lab05:
    a. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
        i. The name of the class and a (very) short description
        ii. An @author tag followed by your name
        iii. An @version tag followed by the version number.
    b. Include a **main method** inside the class definition. Remember the main method is what gets executed first when we start our program.
2. Inside the main method, **instantiate a Scanner object** and use it to interact with the user. Remember to always label your output.

3. Prompt the user to enter the radius and coordinates for a sphere. Use these values to **create a Sphere** object. Print out its surface area and volume.
4. Prompt the user to enter the edge length and centre coordinates for a cube. Use these values to **create a Cube** object. Print out its surface area, volume, and two diagonals.
5. Finally prompt the user to enter the radius and height for a right circular cone. Use these values to **create a Cone** object. Print out its volume, slant height, and surface area.
6. Modify your GeometryDrive to use a **DecimalFormat** object to format the output. Ensure all values are printed to a maximum of three decimal places.
7. **Challenge**: how can you use a **NumberFormat** object to format your output instead?

5. Enhancing our Name Class

We created a simple Name record in lab 4. It stores three parts, and has accessors and a toString method. Let's enhance it with some more functionality:

1. Let's copy your Name record from lab 4 to lab 5. Find it in your lab 4 project. Right click it in the Package Manager and choose copy. Right click the package ca.bcit.comp1510.lab05 and select paste.
2. Notice that when you copied the class to the new project, the package declaration in the file was automatically updated by Eclipse
3. Change the Name class to a record:
    a. First, middle, last become record parameters.
    b. The class' Javadoc needs @param documentation for each parameter.
    c. Delete your explicit constructor.
    d. Delete your explicit accessors.
    e. You will still have the record's implicit constructor and accessors.
4. Add the following methods to your Name class:
    a. A method that accepts no parameters and returns the length of the name, i.e., the sum of the lengths of the three parts of the name.
    b. A method that accepts no parameters and returns a String consisting of the three initials IN UPPERCASE.
    c. A method that accepts an integer n and returns the nth character in the full three-part name.
    d. A method that accepts no parameters and returns a String consisting of the last name followed by a comma followed by the first name followed by the middle name. Remember to put spaces between the names.
    e. A method that accepts a String and returns true if the String is equal to the first name (use the equals method in the String class!). This is not quite the proper way to define an equals() method, as we will learn later. Your record already has an equals() method so use another name for your method.
    f. Would it be useful to define a method that accepts a Name object and returns true if the three parts of the name object are the same as the three parts of "this" Name object?

5. Create a class called **NameDriver** which contains a main method. Inside the main method, instantiate Name object(s) and test your methods to make sure they work. Does your class work for all names?

6. Satisfying Unit Tests

The Math class is pretty handy. It has a couple of useful constants, like pi and e, and methods that perform many calculations. Let's practice implementing methods with our own version of a Math class with help from some unit tests:

1. Create a new class called **Mathematics** inside package ca.bcit.comp1510.lab05:
    a. Do not include a **main method** inside the class definition. Remember the main method is what gets executed first when we start our program. Our Mathematics class is not a complete program.
    b. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
        i. The name of the class and a (very) short description
        ii. An @author tag followed by your name
        iii. An @version tag followed by the version number.
2. There is a file called **MathematicsTest.java** on D2L. Download it and copy it to the same package in eclipse. Don't edit this file.
3. When you copy the MathematicsTest.java file to your program, it will cause compiler errors. We can tell there are compiler errors because the icon for the file in the Package Explorer contains a little red square with a white X in it
4. As in lab 4, the file we copied is a JUnit Test Case. It uses the JUnit testing framework to test the code you will write in the Mathematics class.
5. Add JUnit 5 to your project by using "Fix Project Setup" or updating the build path.
6. Look at the MathematicsTest.java file. It contains tests. Before we can run the tests, we have to make sure we have created "method stubs" for all the tests in the MathematicsTest class, as in lab 4.
7. There is a compiler error on line 38 of MathematicsTest.java. It is trying to test the value returned by the PI constant in the Mathematics class. The problem is we haven't put anything into the Mathematics class yet! Click on the error.
8. Eclipse will provide some suggestions. In this case, we want to add a constant to our empty Mathematics class. Go ahead and choose that suggestion.
9. Eclipse just created a constant called PI in your Mathematics class. Don't worry about assigning the right value now.
10. On a further line of our test class, there is another compiler error. It looks like the test class is trying to test a method called getCircleArea. Click the error icon in the margin. Eclipse will suggest adding a method to the Mathematics class and click OK.
11. Eclipse created an empty method in the Mathematics class. Right now, it just returns zero. You will implement it later. Save the change.
12. Keep doing this until you have created a method stubs in the Mathematics class and eliminated all the compiler errors.

13. After all the compiler errors have been resolved, run the test file. Right click on it and choose **Run As > JUnit Test**.
14. The JUnit window automatically opened and revealed the results of the JUnit tests. Each test method is run independently. Each method contains a simple test that compares the output or return value from a method with an expected value. If a method returns a wrong value, there is a X beside the test. If any tests fail, the coloured strip is red.
15. Your task this lab is to implement the methods in the Mathematics class so the JUnit tests in the MathematicsTest class pass. You can click on any of the failed tests, or you can view the code in the MathematicsTest file to see how the methods are being tested.
16. When your tests all pass, the coloured strip will be green.
17. Here are some helpful Javadoc comments for you to use for each element you must complete. There are hints in the comments about how your methods should be implemented:

```
/**
 * Returns the area of the circle with the specified radius.
 *
 * @param radius
 *        of the circle.
 * @return area as a double
 */
public double getCircleArea(double radius) {

/**
 * Returns the sum of the positive integers between 0 and the specified
 * number inclusive. If the specified number is negative, returns zero.
 *
 * @param number
 *        upper bound
 * @return sum as an integer
 */
public int sumOfInts(int number) {

/**
 * Returns true if the specified value is positive, else false.
 *
 * @param number
 *        to test
 * @return true if number is positive, else false.
 */
public boolean isPositive(int number) {

/**
 * Returns true if the specified value is even, else false.
 *
 * @param number
 *        to test
 * @return true if number is even, else false.
 */
public boolean isEven(int number) {
```

```
/**
 * Returns sum of the even numbers between 0 and the specified value,
 * inclusive.  The value can be positive, negative, or zero.
 *
 * @param number
 *         upper bound
 * @return sum of the even numbers between 0 and number
 */
public int sumOfEvens(int number) {
```

7. You're done! Give your lab instructor your work.

If your instructor wants you to submit your work in the learning hub, export it into a Zip file in the following manner:

1. Right click the project in the Package Explorer window and select export…
2. In the export window that opens, under the General Folder, select Archive File and click Next
3. In the next window, your project should be selected. If not click it.
4. Click *Browse* after the "to archive file" box, enter in the name of a zip file (the same as your project name above with a zip extension, such as Comp1510Lab05BloggsF.zip if your name is Fred Bloggs) and select a folder to save it. Save should take you back to the archive file wizard with the full path to the save file filled in. Then click Finish to actually save it.
5. Submit the resulting export file as the instructor tells you.