CS601-01/CS601-02

Lab2: Shapes Hierarchy

Due Date: Sep 22, 11:59pm (25 pts)

For this lab, you will design and implement a hierarchy of classes representing several 2D and 3D shapes, as well as a `ShaperSorter` class that can store and manipulate lists of shapes. You will write several JUnit tests to test methods in these classes. You will practice:

- Using abstract classes, inheritance and polymorphism

- Implementing Comparator interface

- File input/output

- Writing JUnit tests

You are required to use the provided **starter code** for this lab**; do not modify signatures of any methods. Do not modify the provided test file.** You may not change the directory structure of the project. You are not allowed to use any libraries for this lab except **JUnit.**

To create a private github repository for lab2 with the starter code, click on the following link: https://classroom.github.com/a/CJxXq2iK and follow instructions. Clone this repository to create a local copy on your computer. Your solution should be submitted to your lab2 private repository on github. Please note that solutions submitted via email, or canvas, on in a different repo, will **not** get any credit.

Before starting on the lab, read the documentation in the **doc** folder (open index.html in your browser).

Shape Classes

All the shape classes, as well as class `ShaperSorter`, should be in the *shapes* package of your project (it has already been created and contains class `Shape` and the starter code for class `ShapeSorter`; you need to add other shape classes there: `Shape2D`, `Shape3D`, `Circle`, `Sphere`, `PlatonicSolid`, `ConvexRegularPolygon`, and fill in code in class `ShapeSorter`). We describe each of the shape classes below.

At the top of the shapes hierarchy is class `Shape` - it's **abstract** and has two methods: `area()` and `toString()`. The `area()` method is abstract and will be implemented in all the concrete subclasses of Shape. For three-dimensional shapes, `area()` should return a surface area of the shape. `toString()` method is non-abstract, and returns the string representation of a shape.

`Shape2D` and `Shape3D` should be **abstract** subclasses of `Shape`. `Shape2D` should have an abstract method `perimeter()` and `Shape3D` - an abstract method `volume()`. These classes need to override the `toString()` method of class `Shape` (refer to the comments above the method).

You need to add the following **concrete** (non-abstract) shape classes:

- `Circle` and `ConvexRegularPolygon` should be subclasses of `Shape2D` and provide implementation of the `area()` and `perimeter()` methods.
- `Sphere` and `PlatonicSolid` should be classes of `Shape3D` and provide implementation of the `area()` and `volume()` methods.

Class `Circle` should have a single instance variable, a radius of the circle.

Class `ConvexRegularPolygon` represents a convex regular polygon. It should store the number of edges and the length of each edge. All edges in a regular polygon will be of the same length. The formulas for the area and the perimeter of a convex regular polygon can be found at `https://en.wikipedia.org/wiki/Regular_polygon`

Class `Sphere` should have a single instance variable, a radius of the sphere.

Class `PlatonicSolid` represents convex regular **polyhedrons**. You should represent each platonic solid by three variables: the number of edges for each face, the number of faces meeting at each vertex, and the length of the edge.
You can read about platonic solids and find the formulas for the surface area and the volume of a platonic solid at:
`https://en.wikipedia.org/wiki/Platonic_solid`
`http://math.wikia.com/wiki/Platonic_solid`
`http://hotmath.com/hotmath_help/topics/platonic-solids.html`

ShapeSorter Class

The `ShapeSorter` class should store three lists: a list of all shapes, a list of 2d shapes, and a list of 3d shapes. It should also have the following methods (see the starter code):

- `loadShapes` reads information about several shapes from a file and adds it to the lists. **shapesFile.txt** in the **input** folder contains some shape info. Each line describes one shape and starts with the name of the class representing the shape, followed by a semicolon, followed by the arguments (the number and type of arguments varies from shape to shape and is consistent with the instance variables for each shape class). You need to fill in code for this method.
- `printToFile` prints shape info to a given file. The format is described in the java doc comment above the method. You need to fill in code for this method.
- `sortShapes` sorts the shapes using a given Comparator. Shapes are sorted in ascending order (of whatever criteria is defined by the Comparator). This method has been provided to you.

Comparator Classes

All your Comparator classes should be in the *comparators* package.

We want to be able to compare shapes based on the name or the area. We also want to be able to compare 2D shapes by perimeter, and 3D shapes by volume. `NameComparator` compares `Shape` objects based on the name of the shape, alphabetically. This class has been provided to you in the starter code. You are required to implement three other classes that implement a `Comparator` interface:

- `AreaComparator` that compares `Shape` objects based on the area. Two areas that are within EPS = 0.001 of each other should be considered "equal" for the purpose of this lab.
- `PerimeterComparator` that compares shapes based on the perimeter. Two perimeter values that are within EPS = 0.001 of each other should be considered "equal" for the purpose of this lab. This class should implement *Comparator<Shape>,* but in the compare method, **you should downcast Shape references to Shape2D references**, before calling the *perimeter()* method on them.
- `VolumeComparator` that compares shapes based on the volume. Two volume values that are within EPS = 0.001 of each other should be considered "equal" for the purpose of this lab. This class should implement *Comparator<Shape>,* but in the compare method, **you should downcast Shape references to Shape3D references**, before calling the *volume()* method on them

JUnit Tests

The instructor provided **ShapeSorterTest** file for this project. Note that the test file will not compile until you add all the classes described above.

In addition to passing the tests from **ShapeSorterTest**, you need to write **your own JUnit tests** for the following classes:

- Each concrete shape class (Circle, Sphere, PlatonicSolid, RegularConvexPolygon) should have a corresponding test class, where you need to test methods area() and either perimeter() or volume() (depending on whether it's a 2D shape or a 3D shape). Example: in a test, you can create a Circle object with a given radius, call area() method on it, and compare the return value with the expected value.
- Each Comparator class needs to have test methods as well. For instance, for AreaComparator, we could write a test that creates three different shapes, puts them in a list, calls Collections.sort, passing AreaComparator as a parameter, and compares the list with the expected list.

You may write additional tests for extra credit (discuss with the instructor prior to implementing the tests).

Javadoc Comments

For this lab (any all the programming assignments for this class from now on), you are required to add comments to your code (in Javadoc style).

Submission: Your lab2 should be submitted to your private lab2 repo on github. Please keep pushing your code to github as you work on the lab. **You are required to have at least 5 *meaningful* commits before the deadline.**
If your repo shows no history for the lab, and has only the final version of the lab, you will not get any credit for it. If you have less than 5 commits, we will take off 10% off your grade for the lab and you need to come in for a code review (see below).

Code reviews:

Any student might be asked to come see the professor for a code review of the lab. Not being able to explain your code during a code review will result in a 0 for the assignment.

Cheating

You are **not** allowed to use any code from the web or collaborate on the project with anybody. You may **not** discuss low-level implementation details with anybody except for the instructor, the TAs and CS tutors. Please note that if as a result of the discussion with another student, your code ends up looking very similar, it is considered cheating.