Lab 4, CSC/CPE 203 – Interfaces

Due: 10/19

Orientation

For this lab you will create a virtual drawing workspace on which you can place various geometric shapes. You will not actually be drawing the shapes in this program rather; you will simply be representing geometric shapes as objects and providing the ability to place them in space. The workspace will make use of a `List` from the Java Standard Library to collect shapes and will have methods that allow you to add, remove, and inspect the shapes it contains. All of the shapes share a common set of methods defined by a Java interface. The behavior of these common methods varies, as appropriate, by shape. In addition, each shape has one or more methods unique to it. The following shapes will be supported: circle, rectangle, triangle, and convex polygon.

Objectives

- To develop and demonstrate basic object-oriented development skills. Much of the structure of the solution is given - use good judgment when "filling in the blanks".
- To become familiar with Java interfaces and the concept of polymorphism.
- To practice using the `instanceof` operator.
- More practice using existing classes and interfaces from the Java Standard Library, such as: `List`, `ArrayList`, `LinkedList`, `Point`, and `Color`.

Given File

Retrieve the file provided for this lab from Canvas.

The given file is a *starter* junit test file for your code. Be sure to add to this file to make sure your code is working properly. Your instructors have a much more extensive test file that they reserve the right to test your implementation with!

Ground Rules

Many of the classes and interfaces you will be writing in this lab have counterparts in the Java Standard Library. You ***may not*** use the Java Standard Library versions in your implementation unless explicitly instructed to do so (see below). The Java Standard Library classes and interfaces you may use are identified by a complete package specification. For example, you will be using the `java.awt.Color`, `java.awt.Point`, and `java.util.List` classes/interfaces from the Java Standard Library. You may also use the `java.lang.Math` class from the Java Standard Library (even though it is not explicitly referenced in this document). Any classes or interfaces in the specifications below (other than Point, Color, and List) must be entirely written by you. If you have any questions regarding this restriction please be sure to ask your instructor early in the development process (ideally, before you have written any code!).

As stated above, your classes and interfaces will use many classes that already exist in the Java Standard Library. For example, to use the `Color` and `Point` classes, you will need to import `java.awt.Color` and `java.awt.Point`.

Some of your code from Lab 2 can be reused but will need to be re-written to match this specification.

Task

You must define the `Shape` interface and implement the `Circle`, `Rectangle`, `ConvexPolygon`, `Triangle`, and `Workspace` classes as described below. The specific instance variables are not explicitly specified. Remember, each instance variable makes every object of the class require more memory. All instance variables should be essential to defining the state of an object and/or provide a significant computational efficiency or reduction in code complexity. Other variables should be declared as local variables in the method where they are used. *Be ready to justify your choices!* Be sure all your instance variables maintain encapsulation (i.e. are private).

Also, you may add helper methods other than the methods specified below, but the additional methods must be private.

1. Define a Java interface `Shape` with the following methods. (Note that the input parameters for each method, if any, are intentionally not shown. You must figure out what is needed):
    - `Color getColor()` - Returns the `java.awt.Color` of the `Shape`.
    - `void setColor()` - Sets the `java.awt.Color` of the `Shape`.
    - `double getArea()` - Returns the area of the `Shape`
    - `double getPerimeter()` - Returns the perimeter of the `Shape`
    - `void translate()` - Translates the entire shape by the (x,y) coordinates of a given `java.awt.Point`
2. Define a Java class `Circle` that implements the `Shape` interface. Circles have a radius, a center, and a `Color`. In addition to implementing the required methods specified by the `Shape` interface, you must also implement the following methods. (Again, note that the input parameters for each method, if any, are intentionally not shown):
    - `Circle()` - A constructor with parameters to initialize all its instance variables. Do not implement a default constructor.
    - `double getRadius()` - Returns the radius of the `Circle`.
    - `void setRadius()` - Sets the radius of the `Circle`
    - `Point getCenter()` - Returns the center of the `Circle`

    Note: Be sure to use the constant `Math.PI` when performing any calculations involving PI.

3. Define a Java class `Rectangle` that implements the `Shape` interface. Rectangles have a width, a height, a Point indicating the topLeftCorner, and a `Color`. In addition to implementing the required methods specified by the `Shape` interface, you must also implement the following methods. (Again, note that the input parameters for each method, if any, are intentionally not shown):
    - `Rectangle()` - A constructor with parameters to initialize all its instance variables. Do not implement a default constructor.
    - `double getWidth()` - Returns the width of the `Rectangle`.
    - `void setWidth()` - Sets the width of the `Rectangle`
    - `double getHeight()` - Returns the height of the `Rectangle`.
    - `void setHeight()` - Sets the height of the `Rectangle`

- `Point getTopLeft()` - Returns the `Point` representing the top left corner of the `Rectangle`

4. Define a Java class `Triangle` that implements the `Shape` interface. Triangles have three Points indicating its vertices and a `Color`. To simplify your solution, the vertices will always be given in counter-clockwise order. However, there are calculations for the area of a Triangle that will work with the vertices in any order. In addition to implementing the required methods specified by the `Shape` interface, you must also implement the following methods. (Again, note that the input parameters for each method, if any, are intentionally not shown):
    - `Triangle()` - A constructor with parameters to initialize all its instance variables. Do not implement a default constructor. The first `Point` passed to the constructor will be vertex A, the second `Point` vertex B, and so on.
    - `Point getVertexA()` - Returns the Point representing vertex A of the `Triangle`.
    - `Point getVertexB()` - Returns the Point representing vertex B of the `Triangle`.
    - `Point getVertexC()` - Returns the Point representing vertex C of the `Triangle`.
5. Define a Java class `ConvexPolygon` that implements the `Shape` interface. A ConvexPolygon have a java `Point` array (not `ArrayList`) indicating its vertices and a `Color`. Assume the Points in the `Point` array will always be given in counter-clockwise order. In addition to implementing the required methods specified by the `Shape` interface, you must also implement the following methods. (Again, note that the input parameters for each method, if any, are intentionally not shown):
    - `ConvexPolygon()` - A constructor with parameters to initialize all its instance variables. Do not implement a default constructor.
    - `Point getVertex()` - Takes an index and returns the specified vertex of the `ConvexPolygon`.
6. Define a Java class `WorkSpace` to hold `Shape` objects. The `WorkSpace` class should have *one private instance variable* of type `List<Shape>` to hold `Shape` objects. You may choose which kind of `List` you use. The `WorkSpace` class should have the following public methods. (*Note that this time the parameters **are** specified!*):
    - `WorkSpace()` - A default constructor to initialize its instance variable to an empty List. The constructor should not take any parameters. (If you do nothing in this constructor, you may omit it from your code altogether.)
    - `void add(Shape shape)` - Adds an object which implements the `Shape` interface to the end of the `List` in the `WorkSpace`.
    - `Shape get(int index)` - Returns the specified `Shape` from the `WorkSpace`.
    - `int size()` - Returns the number of `Shapes` contained by the `WorkSpace`.
    - `List<Circle> getCircles()` - Returns a `List` of all the `Circle` objects contained by the `WorkSpace`.
    - `List<Rectangle> getRectangles()` - Returns a `List` of all the `Rectangle` objects contained by the `WorkSpace`.
    - `List<Triangle> getTriangles()` - Returns a `List` of all the `Triangle` objects contained by the `WorkSpace`.
    - `List<ConvexPolygon> getConvexPolygons()` - Returns a `List` of all the `ConvexPolygon` objects contained by the `WorkSpace`.
    - `List<Shape> getShapesByColor(Color color)` - Returns a `List` of all the `Shape` objects contained by the `WorkSpace` that match the specified `Color`.
    - `double getAreaOfAllShapes()` - Returns the sum of the areas of all the `Shapes` contained by the `WorkSpace`.
    - `double getPerimeterOfAllShapes()` - Returns the sum of the perimeters of all the `Shapes` contained by the `WorkSpace`.

Remember to use the `instanceof` operator in the appropriate places of your solution.

Suggestions

Note that these are merely suggestions. You are not required to follow them.

- Write the `Shape` interface first.
- After creating the `Shape` interface, write one class at a time that implements the interface. Test and debug until you are satisfied with its design, implementation, and functionality. Then move on to the next class.
- Implement the `WorkSpace` class last, and develop it incrementally, testing as you go. Notice that many of the methods are similar and differ only by the `Shape` type that they work with.
- The given `TestCases` test the implementation specifics of each of your classes. Comment out the classes that you have not yet implemented so that you can develop and test incrementally.
- Mathwords is a good reference!

Testing

You are given a starting test file to help you write your tests. It includes some tests for Circle, WorkSpace, and your implementation specifics. This file is a good place to find some of the expected method parameters if you are stuck. You are expected to add to this test file to thoroughly test all of your code!

Submission

All your .java files must be submitted as a one ZIP file, to the Canvas before deadline. Demo your work when you finished the Lab.

Rubric

		Points	
1)	Part one Implementation:		
	a. Interface	5	
	b. Circle	5	
	c. Triangle	5	
	d. Poly	5	
	e. Rectangle	5	
	f. Workspace	5	
	g. Test Cases	10	
2)	ExtraTest Cases	30	(Test all your classes and methods)
3)	Submission	30	