CPE 102
FALL 2016
Project 2-Due 10/20

You will be provided with our entire suite of tests for this project. Your code must pass all of the tests to get any credit. Your grade will be based on the date which your code passes all the tests. To get 100% credit for this project, you must submit in PolyLearn by midnight on Thursday, 10/20/6. Code that does not pass all of our tests will not get any credit.

Thursday (10/20) - 100%
Friday (10/21) - 90%
Saturday (10/22) - 80%

Objectives

1. To develop and demonstrate basic object-oriented development skills. Much of the structure of the solution is given - use good judgment to "filling in the blanks"

2. To become familiar with and/or have more practice with:
1. Java ***interfaces***
2. Use of the ***instanceof*** operator
3. Writing ***javadoc***-style comments
4. Using the ***ArrayList*** collection class
5. The concept of ***polymorphism***

Resources

1. How to calculate the area of a general triangle
2. How to calculate the area of a convex polygon
3. Java Standard Library
4. How to write Javadoc comments in your code
5. How to use the javadoc utility to generate the html files
6. P2TestDriver.java (to be published Friday morning at 12:01am, 10/21). Thus, you must pass the test driver without seeing it to get 100%. Thoroughly test your own code!

Ground Rules

Many of the classes and interfaces you will be writing in this project 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. 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.ArrayList classes 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 ArrayList) 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!).

Existing Java Classes

Your classes and interfaces will use many classes that already exist in the Java Standard Library. To use the Color and Point classes, you will need to import java.awt.Color and java.awt.Point. Alternativey you may import java.awt.*. To use the ArrayList class, you will need to import java.util.ArrayList or java.util.*.

Orientation

You will be creating 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 an ***ArrayList*** 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.

Specification

1. You ***must*** implement the classes and methods ***exactly*** as described.

2. Be sure ***all*** instance variables maintain encapsulation (private!).

3. The specific instance variables are not explicitly specified. You are required to make well-reasoned choices as to the data types and number of instance variables used. 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 and/or allow for a significant 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***.

4. ***Include*** the required comment block at the beginning of ***every*** source file (see Resources #4 for an example).

5. Document ***all*** methods in the ***Shape*** interface using the javadoc-style of documentation (see the ***Resources*** section above for help writing javadoc comments). You do not need to javadoc the methods in any of your other files.

6. Write a Java interface called ***Shape*** with the following interface:
    1. double getArea() - Calculates and returns the area of the object.
    2. Color getColor() - Returns the Color of the object.
    3. void setColor(Color color) - Sets the Color of the object.
    4. boolean getFilled() - Returns true if the object is filled with color, otherwise false.
    5. void setFilled(boolean filled) - Sets the filled state of the object.
    6. void move(Point point) - Moves the shape by the x and y amounts specified in the Point.

7. Write a class called ***Circle*** that implements the Shape interface. In addition to implementing the methods specified by the Shape interface you must also implement the following ***public*** methods:
    1. Circle(double radius, Point position, Color color, boolean filled) - Constructor. This should give you a large clue as to what the instance variables of the class should be! The Point specifies the location of the center of the circle. The boolean indicates if the shape is filled (with color) or wire-frame.
    2. double getRadius() - Returns the radius of the Circle object.
    3. void setRadius(double radius) - Sets the radius of the Circle object.
    4. Point getPosition() - Returns the position of the Circle object.

5. Override the equals-method (from Object) so that it returns true for two Circle objects that are logically equivalent based on the state of all of their instance variables.
6. Note: Be sure to use the constant Math.PI (a constant defined in the Math class found in the Java Standard Library) when performing any calculations involving PI.

8. Write a class called ***Rectangle*** that implements the Shape interface. In addition to implementing the methods specified by the Shape interface you must also implement the following ***public*** methods:
    1. Rectangle(double width, double height, Point position, Color color, boolean filled) - Constructor. This should give you a large clue as to what the instance variables of the class should be! The Point specifies the location of the lower-left corner of the rectangle. The boolean indicates if the shape is filled (with color) or wire-frame.
    2. double getWidth() - Returns the width of the Rectangle object.
    3. void setWidth(double width) - Sets the width of the Rectangle object.
    4. double getHeight() - Returns the height of the Rectangle object.
    5. void setHeight(double height) - Sets the height of the Rectangle object.
    6. Point getPosition() - Returns the position of the Rectangle object.
    7. Override the equals-method (from Object) so that it returns true for two Rectangle objects that are logically equivalent based on the state of all of their instance variables.

9. Write a class called ***Triangle*** that implements the Shape interface. In addition to writing the methods specified by the Shape interface you must implement the following ***public*** methods:
    1. Triangle(Point a, Point b, Point c, Color color, boolean filled) - Constructor. This should give you a large clue as to what the instance variables of the class should be! The Point objects represent the three vertices of the triangle in the specified order, a, b, c. The boolean indicates if the shape is filled (with color) or wire-frame.
    2. Point getVertexA() - Returns a specific vertex of the triangle.
    3. void setVertexA(Point point) - Sets a specific vertex of the triangle.
    4. Point getVertexB() - Returns a specific vertex of the triangle.
    5. void setVertexB(Point point) - Sets a specific vertex of the triangle.
    6. Point getVertexC() - Returns a specific vertex of the triangle.
    7. void setVertexC(Point point) - Sets a specific vertex of the triangle.
    8. Override the equals-method (from Object) so that it returns true for two Triangle objects that are logically equivalent based on the state of all of their instance variables.

10. Write a class called ***ConvexPolygon*** that implements the Shape interface. In addition to writing the methods specified by the Shape interface you must implement the following ***public*** methods:
    1. ConvexPolygon(Point[] vertices, Color color, boolean filled) - Constructor. This should give you a large clue as to what the instance variables of the class should be! The array of Point objects represents each vertex of the polygon. Note that the vertices must be specified in ***counterclockwise order*** and that the first/last point (same point in a closed polygon) is not specified twice in the array. You may assume that the array passed to the constructor is in counterclockwise order. The boolean indicates if the shape is filled (with color) or wire-frame.
    2. Point getVertex(int index) - Returns the specified vertex. Note that index must be between zero, inclusive, and the number of vertices, exclusive.
    3. void setVertex(int index, Point point) - Sets the specified vertex of the polygon.
    4. Override the equals-method (from Object) so that it returns true for two ConvexPolygon objects that are logically equivalent based on the state of all of their instance variables. To simplify this implementation, consider two polygons equal only when the have the same vertices in the same order.

11. Write a class called ***WorkSpace***. This class should have ***one private instance variable*** of type ArrayList to hold Shape objects. In addition, the class should have the following ***public*** methods:

1. WorkSpace() - Default Constructor.
2. void add(Shape shape) - Adds objects which implement the Shape interface to the end of the WorkSpace's ArrayList instance variable.
3. Shape remove(int index) - Removes the Shape at the specified index and returns a reference to it or null if the index is out-of-bounds.
4. Shape get(int index) - Return the ith Shape object from WorkSpace.
5. int size() - Returns the number of Shapes contained by the WorkSpace.
6. ArrayList<Circle> getCircles() - Returns an ArrayList of all of the Circle objects contained in the WorkSpace.
7. ArrayList<Rectangle> getRectangles() - Returns an ArrayList of all of the Rectangle objects contained in the WorkSpace.
8. ArrayList<Triangle> getTriangles() - Returns an ArrayList of all of the Triangle objects contained in the WorkSpace.
9. ArrayList<ConvexPolygon> getConvexPolygons() - Returns an ArrayList of all of the ConvexPolygon objects contained in the WorkSpace.
10. ArrayList<Shape> getShapesByColor(Color color) - Returns an ArrayList of all Shape objects in the WorkSpace that match the specified Color.
11. double getAreaOfAllShapes() - Returns the sum of the area of all Shape objects in the WorkSpace.

Suggestions

1. Remember and use - where necessary - the ***instanceof*** operator in the appropriate places of your solution (i.e. when getting all the instances of a certain shape from your workspace). Use getClass() in your equals() methods, though! We'll talk more about the subtle differences between instanceof and getClass() when we learn about inheritence.
2. Implement the Shape interface and document each of the method declarations. Write the comments in a generic fashion that reads well for any shape you might implement. The javadoc utility will use the javadoc comments from the interface to generate the html documentation for any class that implements the interface. This ***saves you the trouble*** of documenting the same methods multiple times in Circle, Rectangle, Triangle, ConvexPolygon, or any other class that my implement the Shape interface in the future.
3. Implement ***one*** shape at a time. Test and debug the shape until you are satisfied with its design, implementation, and functionality ***then*** implement the remaining shapes.
4. Implement the WorkSpace class ***incrementally***. Notice that many of the methods are ***very similar*** and differ only by the Shape type they work with. Implement the ones that relate to a single Shape class, test this code until you are satisfied it is working correctly - you'll then be able to cut and paste it for the other related methods and should only need to make minor modifications.

Handin

Submit in the PolyLearn: Shape.java, Circle.java, Rectangle.java, Triangle.java, ConvexPolygon.java, WorkSpace.java

Note: After the 100% due date has passed, another drop box will be available for next file.