The University of Hong Kong

Department of Computer Science

COMP2396 Object-oriented Programming and Java

Assignment 2

Deadline: 11:55pm, 2nd Oct, 2022.

Overview

This assignment tests your understanding of encapsulation and constructors, and their implementations in Java. You are required to implement 2 public classes, namely Shape and RegularPolygon. The RegularPolygon class is used to model regular *n*-sided polygons, and it is a subclass of the Shape class which provides an abstraction of general shapes. For each of the 2 classes mentioned above, you are required to design and implement a tester class to test the correctness of your implementation. You are also required to write Javadoc for all public classes and their public class members.

Specifications

The Shape class

The Shape class is used to model general shapes. It has private instance variables for storing color, fill-type, orientation, canvas coordinates of the center, and the local coordinates of the vertices of a shape. It has public getters and setters for accessing its private instance variables. It also has public methods for translating and rotating a shape, and for getting the canvas coordinates of the vertices of a shape. Below is a detailed description for the Shape class.

Specification of the Shape class:

private instance variables

`Color color` – a Color object specifying the color of the shape.

`boolean filled` – a boolean value specifying whether the shape is filled or not filled.

`double theta` – a double value specifying the orientation (in radians) of the shape in the canvas coordinate system.

`double xc` – a double value specifying the *x*-coordinate of the center of the shape in the canvas coordinate system.

`double yc` – a double value specifying the *y*-coordinate of the center of the shape in the canvas coordinate system.

`double[] xLocal` – an array of double values specifying the *x*-coordinates of the vertices (in counter-clockwise order) of the shape in its local coordinate system.

`double[] yLocal` – an array of double values specifying the *y*-coordinates of the vertices (in counter-clockwise order) of the shape in its local coordinate system.

public methods:

`Color getColor()` – a method for retrieving the color of the shape.

`boolean getFilled()` – a method for retrieving the fill-type of the shape.

`double getTheta()` – a method for retrieving the orientation (in radians) of the shape in the canvas coordinate system.

`double getXc()` – a method for retrieving the *x*-coordinate of the center of the shape in the canvas coordinate system.

`double getYc()` – a method for retrieving the *y*-coordinate of the center of the shape in the canvas coordinate system.

`double[] getXLocal()` – a method for retrieving the *x*-coordinates of the vertices (in counter-clockwise order) of the shape in its local coordinate system.

`double[] getYLocal()` – a method for retrieving the *y*-coordinates of the vertices (in counter-clockwise order) of the shape in its local coordinate system.

`void setColor(Color color)` – a method for setting the color of the shape.

`void setFilled(boolean filled)` – a method for setting the fill-type of the shape.

`void setTheta(double theta)` – a method for setting the orientation of the shape.

`void setXc(double xc)` – a method for setting the *x*-coordinate of the center of the shape in the canvas coordinate system.

`void setYc(double yc)` – a method for setting the *y*-coordinate of the center of the shape in the canvas coordinate system.

`void setXLocal(double[] xLocal)` – a method for setting the *x*-coordinates of the vertices (in counter-clockwise order) of the shape in its local coordinate system.

`void setYLocal(double[] yLocal)` – a method for setting the *y*-coordinates of the vertices (in counter-clockwise order) of the shape in its local coordinate system.

`void translate(double dx, double dy)` – a method for translating the center of the shape by *dx* and *dy*, respectively, along the *x* and *y* directions of the canvas coordinate system. (i.e., `dx` and `dy` should be added to `xc` and `yc` respectively.)

`void rotate(double dt)` – a method for rotating the shape about its center by an angle of *dt* (in radians). (i.e., `dt` should be added to `theta`.)

`int[] getX()` – a method for retrieving the *x*-coordinates of the vertices (in counter-clockwise order) of the shape in the canvas coordinate system*.

`int[] getY()` – a method for retrieving the *y*-coordinates of the vertices (in counter-clockwise order) of the shape in the canvas coordinate system*.

** Please refer to the formula given in assignment 1 on how to compute the canvas coordinates of the vertices of a shape from its local coordinates based on its center and orientation.*

The RegularPolygon class

The RegularPolygon class is a subclass of the Shape class and is used to model regular *n*-sided polygons. Besides the properties it inherited from the Shape class, the RegularPolygon class also declares a number of private instance variables for storing the number of sides and the radius of a polygon. It has public getters and setters for accessing its private instance variables. It also has public methods for setting the local coordinates of the vertices of a polygon and for checking if a point (in the canvas coordinate system) is contained by a polygon. Below is a detailed description for the RegularPolygon class.

Specification of the RegularPolygon class:

public constructors:

`RegularPolygon(int n, double r)` – a constructor for building a regular `n`-sided polygon with a radius of `r`. (Note that if the argument `n` is less than 3, the number of sides will be set to 3; if the argument `r` is less than 0, the radius will be set to 0).

`RegularPolygon(int n)` – a constructor for building a regular `n`-sided polygon with a radius of 1.0. (Note that if the argument `n` is less than 3, the number of sides will be set to 3).

`RegularPolygon()` – a constructor for building a regular 3-sided polygon with a radius of 1.0.

private instance variables:

`int numOfSides` – an integer value specifying the number of sides of the regular *n*-sided polygon.

`double radius` – a double value specifying the radius of the regular *n*-sided polygon.

public/private methods:

`int getNumOfSides()` – a method for retrieving the number of sides of the regular polygon.

`double getRadius()` – a method for retrieving the radius of the regular polygon.

`void setNumOfSides(int n)` – a method for setting the number of sides of the regular `n`-sided polygon. This method should also reset the local coordinates of the vertices of the regular `n`-sided polygon. (Note that if the argument `n` is less than 3, the number of sides will be set to 3).

`void setRadius(double r)` – a method for setting the radius of the regular *n*-sided polygon. This method should also reset the local coordinates of the vertices of the regular *n*-sided polygon. (Note that if the argument `r` is less than 0, the radius will be set to 0).

`void setVertices()` – a method for setting the local coordinates of the vertices of the regular *n*-sided polygon based on its number of sides and radius (see appendix). If the number of sides is an odd number, the first vertex should lie on the positive *x*-axis and its distance from the origin is given by the radius of the regular *n*-sided polygon. The rest of the vertices can be obtained by rotating this vertex about the origin by a multiple of $2\pi/n$, where *n* is the number of sides, in a counter-clockwise manner (see figure 1).

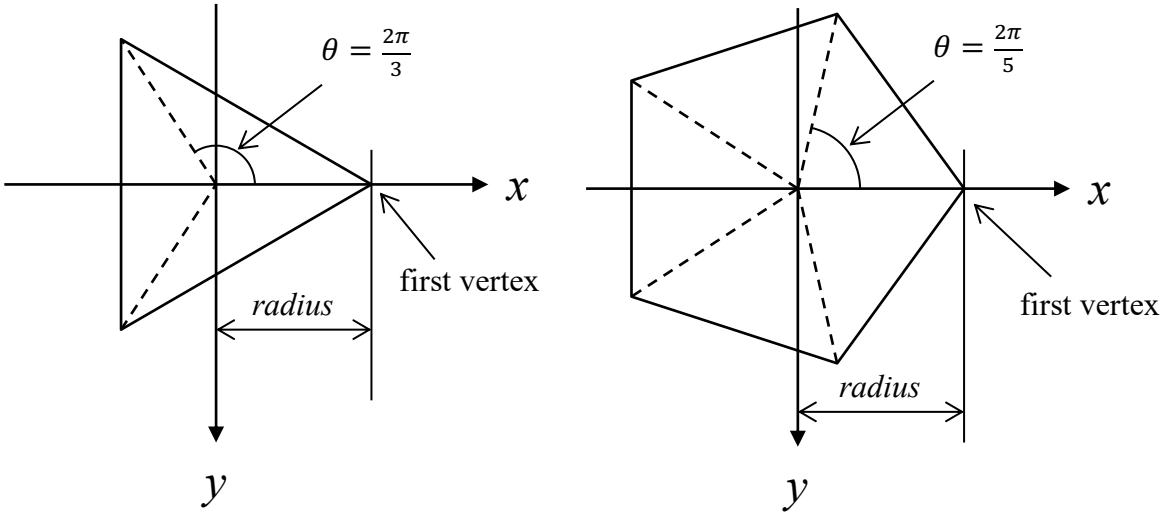

Figure 1. Examples of regular *n*-sided polygons with an odd number of sides.

If the number of sides is an even number, the first vertex should lie in the first quadrant (i.e., both its *x* and *y* coordinates being positive) and make an angle of π/*n*, where *n* is the number of sides, with the positive *x*-axis. Its distance from the origin is again given by the radius of the regular *n*-sided polygon. Similarly, the rest of the vertices can be obtained by rotating this vertex about the origin by a multiple of 2π/*n*, where *n* is the number of sides, in a counter-clockwise manner (see figure 2).

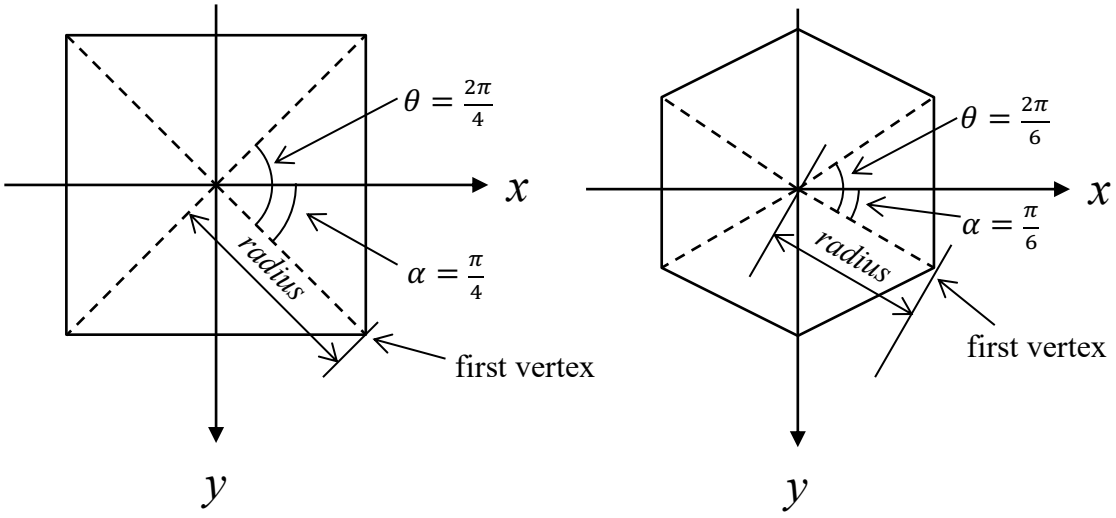

Figure 2. Examples of regular *n*-sided polygons with an even number of sides.

`boolean contains(double x, double y)` – a method for determining if a point (*x*, *y*) in the canvas coordinate system is contained by the regular *n*-sided polygon. A point is considered to be contained by a polygon if it lies either completely inside the polygon, or on any of the sides or vertices of the polygon (see appendix).

** You should decide whether you declare the above methods of the RegularPolygon class as public or private depending on whether they are supposed to be called from outside the class.*

*** Remember to re-initialize the local coordinates of the vertices of the regular n-sided polygon after you change its number of sides and/or radius.*

The tester classes

The tester classes are used to verify the correctness of your implementation of the above 2 classes. You should design your own tester classes. Generally, your tester class should create an object of a class and access all its public instance variables and methods using the dot operator, and print out debugging messages to the console. (You do not need to draw the shapes in your tester classes!)

A simple GUI for visualization

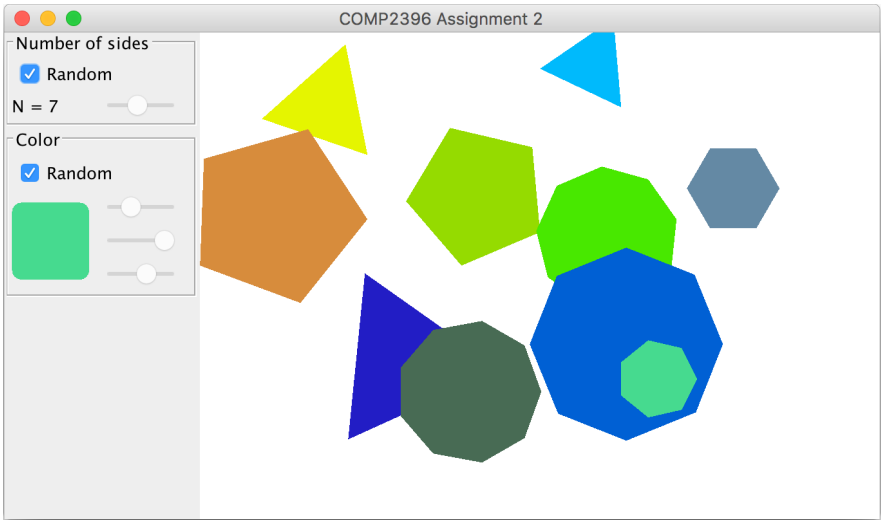

Figure 3. `Assign2_GUI.java` implements a simple GUI for visualizing the shape classes.

`Assign2_GUI.java` implements a simple GUI for you to test and visualize your shape classes. You can select the number of sides and color on the tools panel on the left, and draw a regular *n*-sided polygon by clicking and dragging with your left mouse button in the drawing canvas on the right. You can also select and move an existing polygon by clicking and dragging with your right mouse button. Finally, you can select and remove an existing polygon by clicking with your right mouse button while holding the crtl-key down.

Marking Scheme

Marks are distributed as follows:

- Implementation of the Shape class and its tester class (20%)
- Implementation of the RegularPolygon class and its tester class (40%)
- Proper design and use of encapsulation (20%)
- Javadoc and comments (20%)

Submission

Please pack the source code (`*.java`) of your shape classes and tester classes into a single zip file, and submit it to the course Moodle page.

A few points to note:

- Always remember to write Javadoc for all public classes and their public class members.
- Always remember to submit the source code files (`*.java`) but **NOT** the bytecode files (`*.class`).
- Always double check after your submission to ensure that you have submitted the most up-to-date source code files.
- Your assignment will not be marked if you have only submitted the bytecode files (`*.class`). You will get zero mark for the assignment.
- Please submit your assignment on time. Late submission will not be accepted.

~ End ~

Appendix

Local coordinates of the vertices of a regular *n*-sided polygon

For a regular *n*-sided polygon with a radius of *r*, the local coordinates of its *i*-th vertex can be computed using the following formula:

$$x_i = r\cos(\alpha - i\theta)$$
$$y_i = r\sin(\alpha - i\theta)$$

where $i \in \{0,1,..,n-1\}$, $\theta = 2\pi/n$, $\alpha = 0$ if *n* is odd (see figure 1) or π/n if *n* is even (see figure 2).

Transforming a point from canvas coordinates into local coordinates

Given a point defined in the canvas coordinate system, it can be transformed into a point defined in the local coordinate system of the polygon using the following formula:

$$x = (x' - xc)\cos(-\theta) - (y' - yc)\sin(-\theta)$$
$$y = (x' - xc)\sin(-\theta) + (y' - yc)\cos(-\theta)$$

where (*x*, *y*) and (*x'*, *y'*) denote the local and canvas coordinates of the point, respectively, θ the orientation (in radians) of the shape, and (*xc*, *yc*) the canvas coordinates of the center of the shape.

Checking if a point is contained by a regular *n*-sided polygon

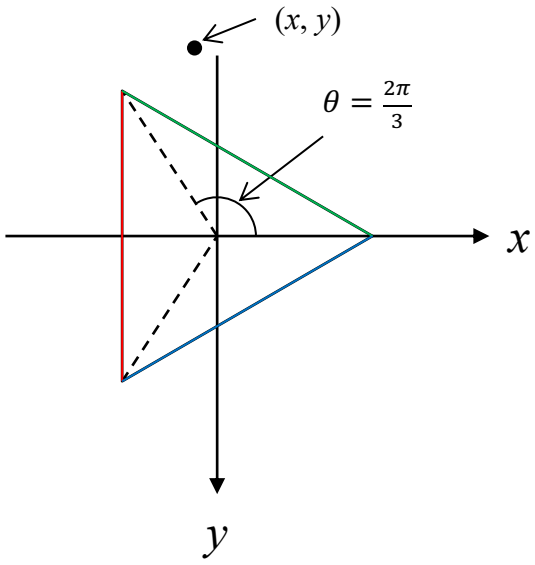

Figure 4. Checking if a point is contained by a triangle.

Consider a regular 3-sided polygon (i.e., a triangle) and a point (*x*, *y*) defined in the local coordinate system of the triangle. Referring to figure 4, obviously the point must lie outside the triangle if it lies to the left of the vertical (red) side of the triangle. This condition can be checked easily by simply comparing the *x*-coordinate of the point with that of the vertices forming the vertical (red) side. In figure 4, since the point lies to the right of the vertical (red) side, further checking against the other 2 sides is needed.

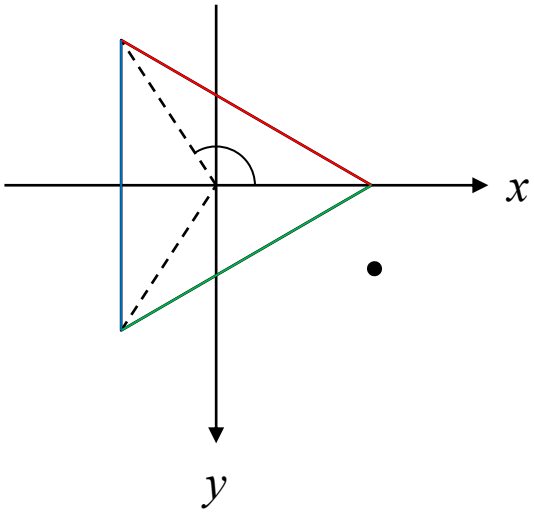

Figure 5. Checking if a point is contained by a triangle.

To simplify the checking, both the triangle and the point are rotated together about the origin by an angle of $2\pi/3$, giving raise to the configuration in figure 5. The point is now checked against the vertical (blue) side. Since the point lies to the right of the vertical (blue) side, further checking against the remaining side is needed.

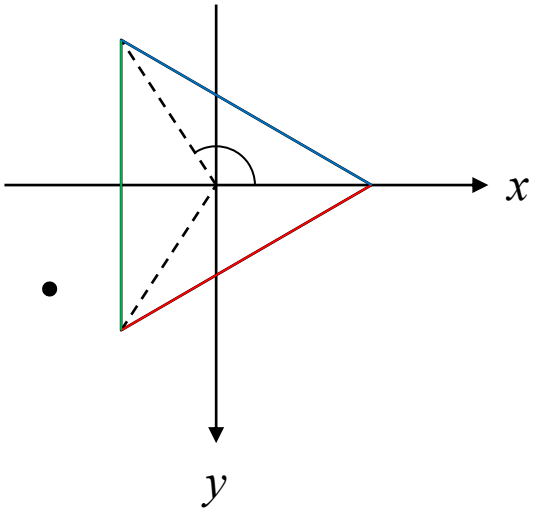

Figure 6. Checking if a point is contained by a triangle.

Same as above, both the triangle and the point are rotated together about the origin again by an angle of $2\pi/3$, giving raise to the configuration in figure 6. This time the point lies to the left of the vertical (green) side, and therefore it can be concluded that the point is not contained by the triangle.

Note that in the above process, the triangle stays unchanged after each rotation. Hence, it is sufficient to only rotate the point when carrying out the checking. If the point does not lie to the left of the vertical side after each rotation, it can then be concluded that the point is contained by the triangle, otherwise the point is not contained by the triangle. This method can be generalized and applied to check if a point is contained by a regular n-sided polygon. In this case, the point needs to be rotated $(n - 1)$ times about the origin by an angle of $2\pi/n$.