Lab 4: Inheritance and composition

Submit source code and UML diagram in PDF to Mooshak problem E at:

http://deei-mooshak.ualg.pt/~hdaniel

up to Apr 19, 2021

This lab assignment is composed by:

- the **source code**
- the **corresponding UML class diagram** in a *.pdf file.

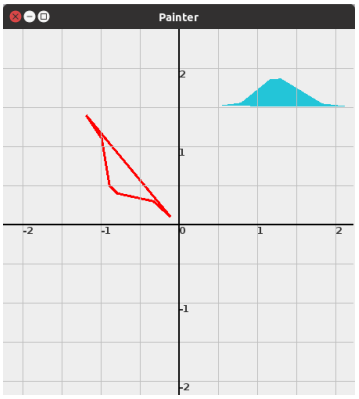
When submitting to Mooshak include inside the folder with the source code a **uml.pdf** file with the UML class diagram.

Make sure that all the **Constraints specified below** and required for validation are met, before discussion.

How to check if the implementation is correct:

1)
Check that the output of the implementation is the expected in the **Examples** and **Sample Cases**, presented in the specifications below.

2)
The output of the implementation can be passed as the input of the **Painter** program distributed with these specifications (painter.zip), to display figures. The first figure, the original is displayed with a red outline and the transformed in blue.



Extract the **painter.zip**. Inside the extracted folder **painter** there is a jar file: **painter.jar** and a folder **src** with the source code of the painter.
Just run the jar from command line (or compile the source code in folder **src**) and pass in 2 lines: the specification of the original figure and the specification of the transformed, ie. the output of problem E implementation:

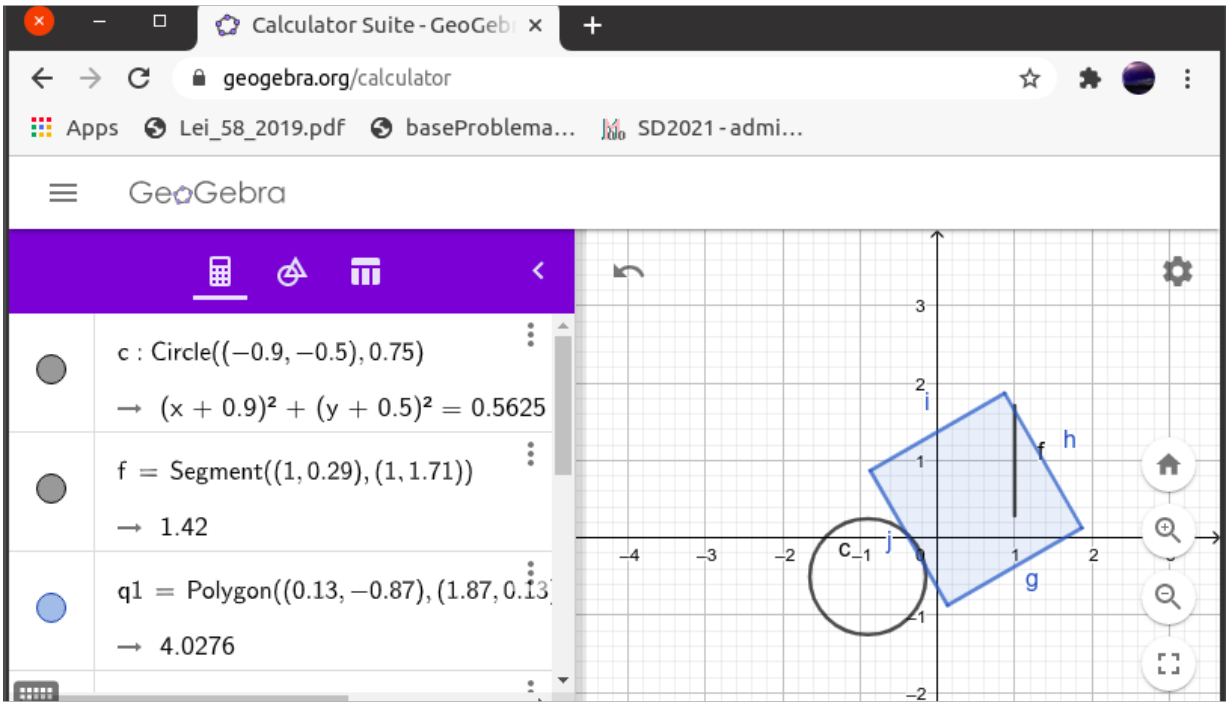
Example:
java -jar painter.jar

Polygon((-1.19,1.40),(-0.99,1.10),(-0.89,0.50),(-0.79,0.40),(-0.33,0.30),(-0.11,0.10))
Polygon((2.18,1.51),(1.83,1.55),(1.30,1.86),(1.16,1.85),(0.79,1.56),(0.49,1.52))

3)
The output of the program can be used also in **GeoGebra** to display figures:

https://www.geogebra.org/calculator

Example:
Circle((-0.90,-0.50),0.75)
Segment((1.00,0.29),0.75)
(...)



Problem E: Moving Objects

Submit source code and UML diagram in PDF format.

To develop a graphics editor it is needed to implement 2 operations on geometric figures:

move
rotate

The supported geometric figures are:

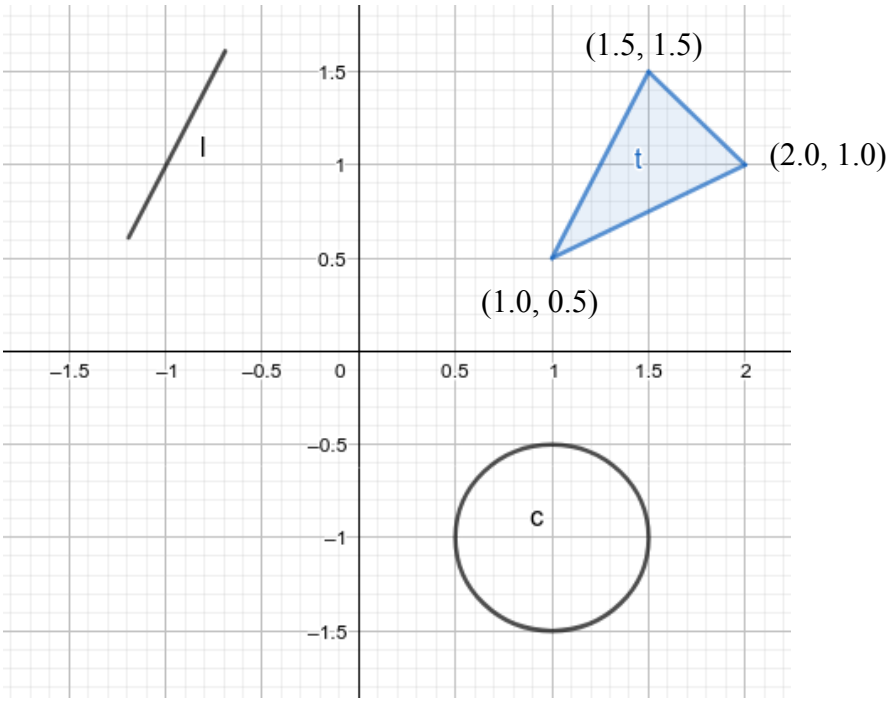
Circles, defined by a centre point and a radius.

Line segments, defined by 2 points, one in each extremity.

Polygons, defined by 3 or more points, **ordered counter clockwise**.
For the triangle in the figure could be:

{ (1.0, 0.5), (2.0, 1.0), (1.5, 1.5) }
or { (2.0, 1.0), (1.5, 1.5), (1.0, 0.5) }
or { (1.5, 1.5), (1.0, 0.5), (2.0, 1.0) }

Points coordinates and radius are real numbers.



The behaviour of **move** and **rotate** operations is always the same for all the figures and operates on all the points in each figure.

To **move** a figure **dx** and **dy** units, respectively across x-axis and y-axis, all the points in that figure must be moved by the same **dx** units and the same **dy**.

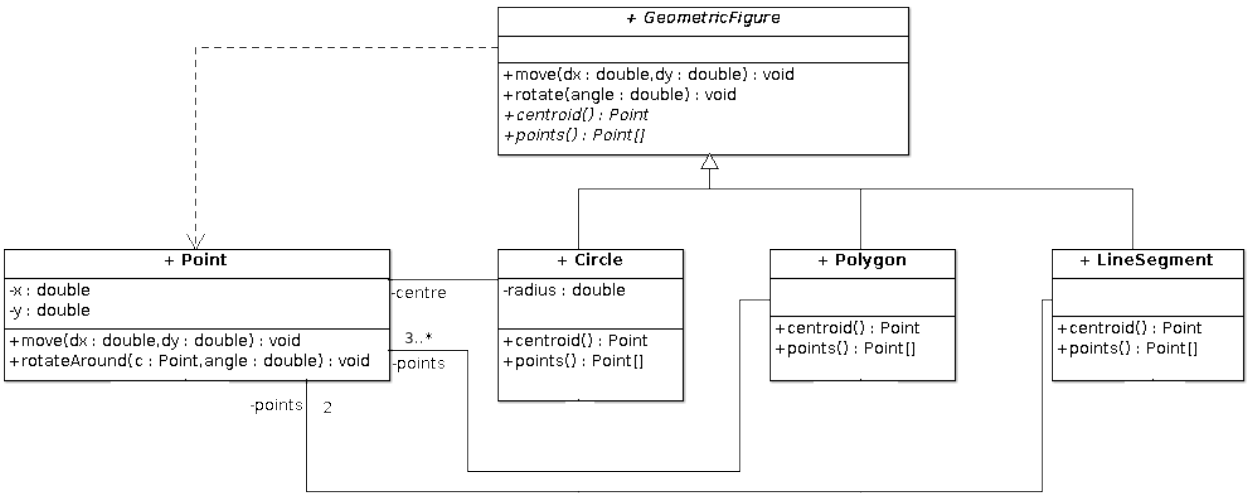
To **rotate** a figure some degrees counter clockwise, it is considered a rotation around its **centroid**. For a **circle** the centroid is the centre point. For a **line segment** the centroid is the middle point, for a **polygon** can be computed as described in the proposed algorithm below.
Again, rotating all the points in the figure, around the same centroid, rotates the whole figure around the centroid.

Since all figures are defined by points, even a circle have a centre point, and **move** and **rotate** operations involves transformations in every point, the **move** and **rotate around a point** operations must be defined in a **Point** class.

Then to move or rotate figures it is just needed to operate on every point of the figure, essentially a loop calling the **move(dx, dy)** or **rotateAround(centre, angle)** methods defined in class **Point**, for every point that defines a figure.
This behaviour can be implemented in **2 template methods** defined in an abstract super class **GeometricFigure**: **move(dx, dy)** and **rotate(angle)** around the centroid. Since these 2 methods need to access the centroid of each figure and all the points as well, to implement in the abstract class **move()** and **rotate()** template methods, it is needed to declare 2 abstract methods in **GeometricFigure**: **centroid()** and **points()**.

Then, sub classes must implement these 2 methods, one to get the centroid point: **Point centroid()**, which is computed differently for every sub class, and another to get all the points that defines the figure in a collection, ordered counter clockwise (in the case of the circle it is just the centre point): **Point[] points()**.

An initial UML class diagram that presents the above relations can be one below:



Move a figure
To move a figure **dx**, **dy** units, respectively across x-axis and y-axis, simple add **dx** and **dy** to the **x** and **y** coordinates of all the points that defines the figure.

A possible algorithm to compute the centroid of a polygon
For n points ordered counter clockwise: (x_0, y_0), (x_1, y_1), … (x_{n-1}, y_{n-1}), the centroid point (x_c, y_c) coordinates x_c and y_c can be obtained with:

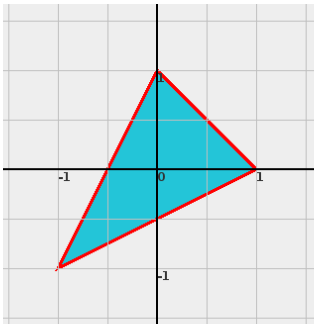
$$x_c = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) + \frac{1}{6A}(x_{n-1} + x_0)(x_{n-1} y_0 - x_0 y_{n-1})$$

$$y_c = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i) + \frac{1}{6A}(y_{n-1} + y_0)(x_{n-1} y_0 - x_0 y_{n-1})$$

Note that after the sum it is added another parcel, with the same operation, but just from the last point to the first point of the polygon, to close it.

The **A** in the above formulas represents the area of the polygon, and can be computed with **shoelace** formula *(again after the sum it is added a parcel to close the polygon)*:

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) + \frac{1}{2}(x_{n-1} y_0 - x_0 y_{n-1})$$



Note that using this formula, the area of a point or a line segment is 0 as expected.

Example
For the triangle defined by points: { (-1.0, -1.0), (1.0, 0.0), (0.0, 1.0) }
The area is 1.50.
The centroid is point: (0.0, 0.0).

A possible algorithm to rotate a point around another point (the centre or centroid point)
To rotate point P (x_p, y_p) around point C (x_c, y_c), counter clockwise, degsº degrees:

#1
move point C, the rotation centre point, to the origin (0.0, 0.0) to simplify the rotation step. This means that point P must be moved the same amount that point C needs to be moved to reach the origin:

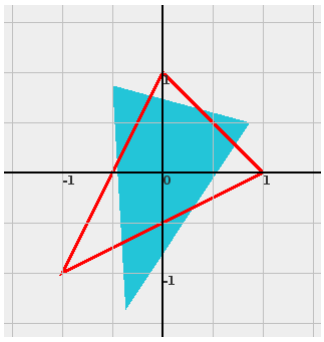
$x_p = x_p - x_c$
$y_p = y_p - y_c$

#2
rotate point P around the origin, in radians:

rads = degs * π / 180;

$x_n = x_p$ * cos(rads) - y_p * sin(rads);
$y_n = x_p$ * sin(rads) + y_p * cos(rads);

#3
move point P back by the same amount it was moved in step #1:



$x_p = x_n + x_c$
$y_p = y_n + y_c$

Example
For the triangle defined by points: { (-1.0, -1.0), (1.0, 0.0), (0.0, 1.0) }
Rotating it 30 degrees, will transforms the 3 defining triangle points to:

{ (-0.37, -1.37), (0.87, 0.50), (-0.50, 0.87) }

In the figure the red outline is the original position of the polygon (triangle).

Note that if the value of degrees to rotate is negative, this algorithm rotates the figure clockwise.

Task
Implement this class hierarchy and a client, **Transformer.java**, that will get the definition of a figure as input, as well as transform instructions, and presents the centroid and area of the original figure and the definition of the rotated and moved figure, as specified below
.
The UML diagram can be tweaked and changed, however the constraints below must be fulfilled to validate the submission.

Constraints
The validation of an accepted submission depends on all of the following requirements:

1) **Point** class must implement **move()** and **rotateAround()** concrete methods, as defined in the UML class diagram above.

2) **GeometricFigure** abstract class must implement **move()** and **rotate()** abstract methods and also declare the abstract methods **centroid()** and **points()**, as defined in the UML class diagram above.

3) Subclasses must implement their own versions of **centroid()** and **points()** concrete methods, as defined in the UML class diagram above.

4) The final UML class diagram, in *.pdf format, must be included in the source code folder

Input
The input has three lines:

The first line is an integer that specifies how many doubles will have the second line.

The second line is a set of doubles separated by spaces.

> If there are 3 doubles, they specify a circle centre point x coordinate, centre point y coordinate and the circle radius, respectively.

If there are 4 doubles, they specify a line segment between to points. Respectively the x and y coordinates of one point, followed by the x and y coordinates of the other point.

If there are more than 4 doubles, the count of doubles will always be even, and they specify the vertices of a polygon, ordered counter clockwise:

x0 y0 x1 y1 x2 y2 ...

The third line is the transformation to operate on the figure, specified by 3 doubles, respectively the movement on the x-axis, movement on y-axis and rotation counter clockwise in degrees, if degrees is positive. If it is negative the rotation should be clockwise.

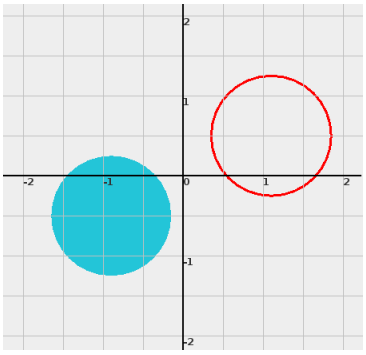
Output
The output has 4 lines.
The first line is the centroid of the original figure before transformation: (centroidX,centroidY)
The second line is the area of the figure.
The third line represents the figure before transformation with the syntax below.
The fourth line represents the figure after transformation with the same syntax:

Circle((centreX,centreY),radius)
Segment((x0,y0),(x1,y1))
Polygon((x0,y0),(x1,y1), … (xn,yn))

all double values must be written with 2 decimal places precision.

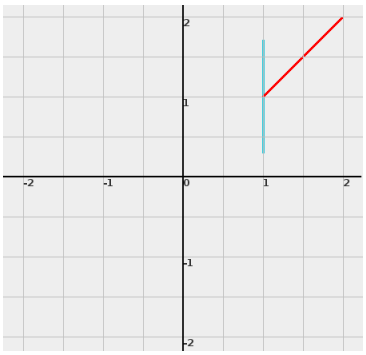
Operation samples
Below are some samples of expected operation. In the pictures, the red outline is the original and the blue the transformed figure.

Sample Input 0
3
1.1 0.50 0.75
-2 -1 45



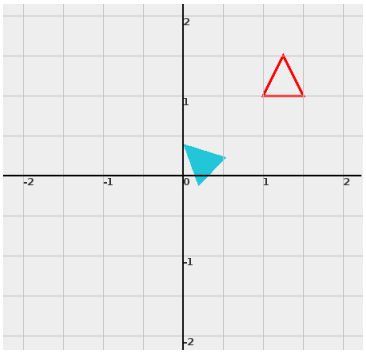
Sample Output 0
(1.10,0.50)
1.77
Circle((1.10,0.50),0.75)
Circle((-0.90,-0.50),0.75)

Sample Input 1
4
1 1 2 2
-0.5 -0.5 45



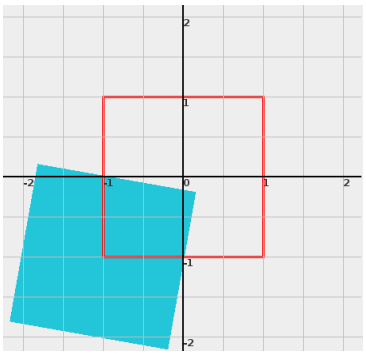
Sample Output 1
(1.50,1.50)
0.00
Segment((1.00,1.00),(2.00,2.00))
Segment((1.00,0.29),(1.00,1.71))

Sample Input 2
6
1 1 1.5 1 1.25 1.5
-1 -1 45



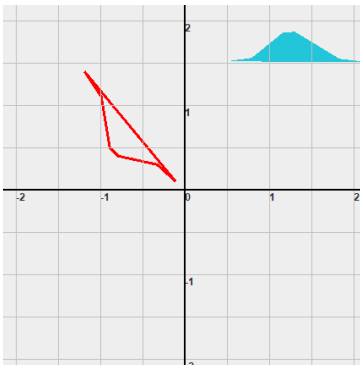
Sample Output 2
(1.25,1.17)
0.13
Polygon((1.00,1.00),(1.50,1.00),(1.25,1.50))
Polygon((0.19,-0.13),(0.54,0.23),(0.01,0.40))

Sample Input 3
8
-1 -1 1 -1 1 1 -1 1
-1 -1 -10



Sample Output 3
(0.00,0.00)
4
Polygon((-1.00,-1.00),(1.00,-1.00),(1.00,1.00),(-1.00,1.00))
Polygon((-2.16,-1.81),(-0.19,-2.16),(0.16,-0.19),(-1.81,0.16))

Sample Input 4
12
-1.19 1.4 -0.99 1.1 -0.89 0.5 -0.79 0.4 -0.33 0.3 -0.11 0.1
2 1 -130



Sample Output 4
(-0.71,0.63)
0.23
Polygon((-1.19,1.40),(-0.99,1.10),(-0.89,0.50),(-0.79,0.40),(-0.33,0.30),(-0.11,0.10))
Polygon((2.18,1.51),(1.83,1.55),(1.30,1.86),(1.16,1.85),(0.79,1.56),(0.49,1.52))