Lab Exercise 10
Java Graphics
CS 2334

October 26, 2017

Introduction

This lab will give you experience with creating graphics in Java. A knowledge of using graphics will allow you to customize your GUI with shapes and colors. Combined with what you have already learned about graphical components (**JButton**, **JLabel**, **JTextField**, etc.), there are infinite possibilities!

Your specific task for this lab is to put together a set of shapes that will create the image of Batman.

Learning Objectives

By the end of this laboratory exercise, you should be able to demonstrate a knowledge of graphics by:

1. Creating a window
2. Adding various graphical components to the window
3. Customizing the size, location, and color of those components to form an organized image

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Import the existing lab10 implementation into your eclipse workspace.

    (a) Download the lab10 implementation:
    `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab10/lab10.zip`

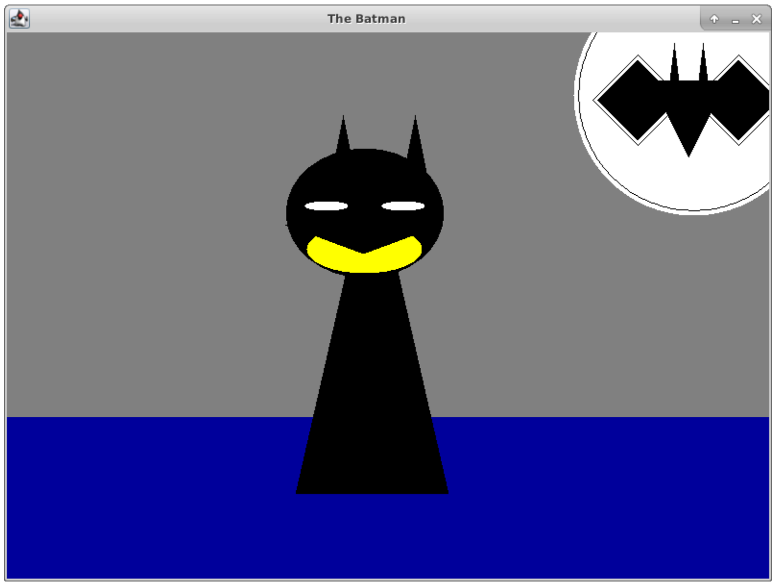
    (b) In Eclipse, select *File/Import*

    (c) Select *General/Existing projects into workspace.* Click *Next*

    (d) Select *Select archive file.* Browse to the lab10.zip file. Click *Finish*

Image

Below is the illustration that you will be mimicking.



This is only an image and does not provide any way for a user to interact with your program (other than closing the window).

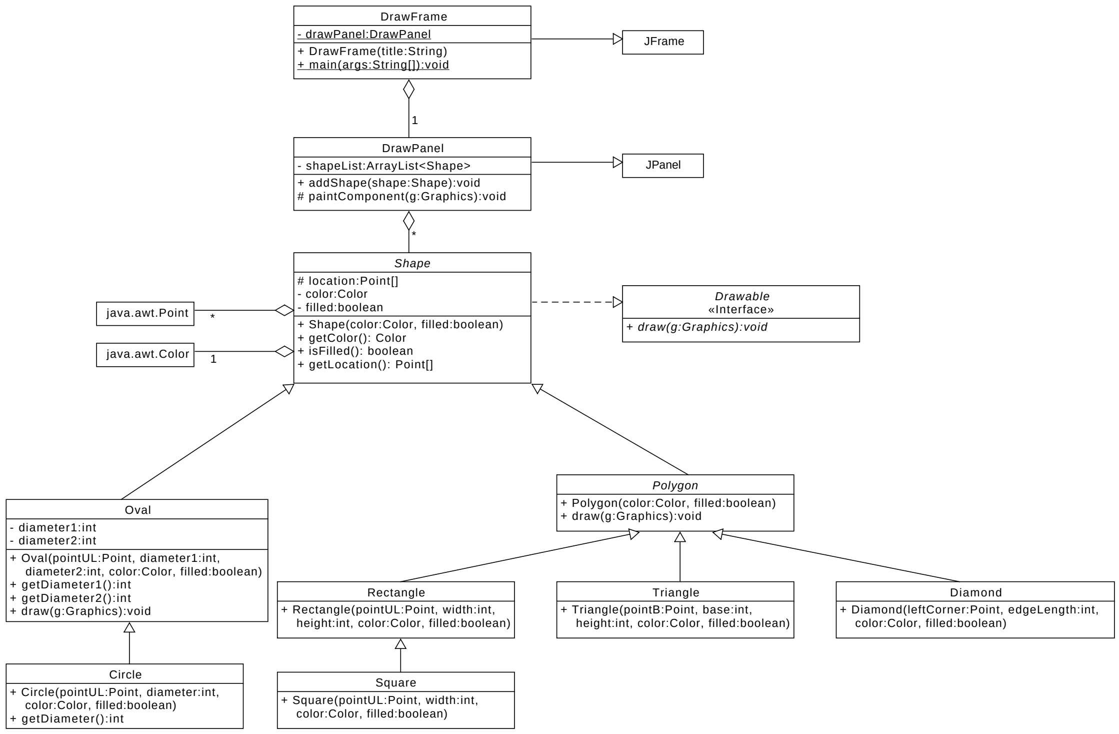
Specifically, the picture shown is made up of the following components:

- 4 instances of **Circle** or **Oval** for the head, face, and eyes
- 2 instances of **Circle** for the light of the bat-signal (upper right of the picture)
- 3 instances of **Triangle** for the cowl (the ears and the nose)
- 3 instances of **Triangle** for the bat-signal
- 1 instance of **Triangle** for the body
- 4 instances of **Diamond** for the bat-signal wings
- 2 instances of **Rectangle** for the sky and ground

You do not need to worry about matching the exact locations shown in the example picture – you are allowed to use a small amount of creativity in your choice of sizes and colors. However, each of the shapes *must* be represented, your final product *must* closely resemble ours, and your tests must cover your code. Be aware that some of these shapes do overlap, so the order in which you add them to the panel is important (e.g., the mouth is constructed by a black triangle sitting over a yellow ellipse). You will not receive credit for a shape that is technically in the panel, but is not visible.

The image also contains a mixture of filled and unfilled objects. This is necessary in order to properly cover all of your code.

UML


DrawFrame
- drawPanel:DrawPanel
+ DrawFrame(title:String)
+ main(args:String[]):void
JFrame
1
DrawPanel
- shapeList:ArrayList<Shape>
+ addShape(shape:Shape):void
# paintComponent(g:Graphics):void
JPanel
*
Shape
# location:Point[]
- color:Color
- filled:boolean
+ Shape(color:Color, filled:boolean)
+ getColor(): Color
+ isFilled(): boolean
+ getLocation(): Point[]
java.awt.Point
*
java.awt.Color
1
Drawable
«Interface»
+ draw(g:Graphics):void
Oval
- diameter1:int
- diameter2:int
+ Oval(pointUL:Point, diameter1:int, diameter2:int, color:Color, filled:boolean)
+ getDiameter1():int
+ getDiameter2():int
+ draw(g:Graphics):void
Polygon
+ Polygon(color:Color, filled:boolean)
+ draw(g:Graphics):void
Rectangle
+ Rectangle(pointUL:Point, width:int, height:int, color:Color, filled:boolean)
Triangle
+ Triangle(pointB:Point, base:int, height:int, color:Color, filled:boolean)
Diamond
+ Diamond(leftCorner:Point, edgeLength:int, color:Color, filled:boolean)
Circle
+ Circle(pointUL:Point, diameter:int, color:Color, filled:boolean)
+ getDiameter():int
Square
+ Square(pointUL:Point, width:int, color:Color, filled:boolean)

Lab 10: Specific Instructions

Most of the classes shown in the UML are provided in lab10.zip. Implementations are provided in full for the **Circle, Diamond, Polygon** and **ShapeUtils**. Create and/or complete the other classes and interfaces. Write JUnit tests for all the shape classes. Partial test implementations have been provided for some shapes; complete those tests classes and/or create the rest.

1. Make sure that in your final implementation, all variables and methods shown in the UML are included and implemented in these classes
   - Be sure that the class name and access keywords are the same as shown in the UML diagram.
   - You must use the default package, meaning that the package field must be left blank.
   - Do not change the variable and method names provided.
   - You are responsible for making sure all method documentation is complete.
2. Create the interface Drawable and the abstract class Shape.
3. Modify the code according to the TODO instructions given in the comments and update and/or include Javadoc comments as needed. These classes are: **Drawframe, Oval, Rectangle, Square** and **Triangle**.
4. Implement JUnit tests to check the non-drawing aspects of all the Shape classes. For example, creating a **Circle** instance should result in an object with the correct radii and color. You can use the **ShapeUtils** class to assist with your tests. See the **DiamondTest** class for examples.
5. On Web-Cat, we will not test the main method of your **DrawFrame** class (or its coverage). However, we will test the constructor, which generates your picture. Note that the **DrawFrameTest** class does not need to be altered and will help you ensure that the image you created tests all of the possible shapes.
6. **DO NOT modify the ShapeUtils class**.

DrawFrame

This class extends from JFrame and is the window that holds all of the drawn components. This class is also the main entry point for the program and where you will draw *The Batman*.

- The constructor takes a String for the title of the window. This method is where you will be creating and adding shapes to draw Batman. You will need to add these shapes into the frame's DrawPanel in order for them to be drawn.

DrawPanel

This class extends from JPanel, maintains a list of all the shapes, and then draws each of them.

- *addShape(Shape shape)*: This method adds shapes the list of shapes.
- *paintComponent(Graphics graphics)*: This method takes a **Graphics** object and uses it to draw the **Shapes** into this component. This method uses the **Shapes'** draw() methods to do this appropriately. Note: you will not explicitly call this method, as it is invoked by Java's painting subsystem whenever a component needs to be rendered.

Drawable

This is an interface for drawing objects.

- You will need to create this interface.
- *draw(Graphics graphics)*: This method defines how to draw the object, and takes a Graphics object to use for drawing.

Shape

This class holds the location and color of the shape, as well as whether the shape is filled, and implements Drawable.

- You will need to create this abstract class.
- The constructor takes a color object (as a **java.awt.Color**), and a boolean determining whether the shape should be drawn filled.
- *getColor()*: Retrieve the shape's color
- *isFilled()*: Returns true if the space occupied by the shape is to be filled by the color, false otherwise
- The *location* instance variable is used to store the **Points** that are necessary to represent the **Shape**. The number of **Points** that are represented depends on the type of **Shape**
- *getLocation()*: Returns a Point array that contains all the points used to define the shape

Oval

This class maintains information for drawing an Oval, defines how ovals are drawn, and extends Shape.

- The constructor takes a single **Point** for the upper left corner of the bounding box that the oval resides in, the first diameter (from left to right) and the second diameter (from top to bottom) as an integer, a Color object, and a boolean determining whether the shape should be filled in.
- *getDiameter1()*: Returns the first diameter (from left to right)
- *getDiameter2()*: Returns the second diameter (from top to bottom)
- *draw(Graphics graphics)*: This method takes a Graphics object to perform the drawing, and places the oval at its location and fills it with the color if filled is true.

Circle

This class maintains information for drawing a Circle, and extends Oval.

- The constructor takes a single **Point** for the upper left corner of the bounding box that the circle resides in, the diameter as an integer, a color object, and a boolean determining if the shape should be filled in.
- *getDiameter()*: Returns the diameter of the circle

Polygon

This abstract class maintains information for drawing a Polygon, defines how to draw them, and extends Shape.

- The constructor takes a color object, and a boolean determining whether the shape should be filled in.
- *draw(Graphics graphics)*: This method takes a graphics object to perform the drawing, places a polygon with the set of points defining its location and fills it with the color if *filled* is true.
- The vertices that define a polygon are stored in the *location* instance variables.

Diamond

This class maintains information for drawing a Diamond and extends Polygon.

- The constructor takes a point for the leftmost vertex, an integer for the edge length, a color object, and a boolean determining whether the shape should be filled in.
- The Diamond object explicitly represents all four vertices.

Triangle

This class maintains information for drawing a Triangle and extends Polygon.

- Only isosceles triangles are represented, in which the base of the triangle is horizontal.
- The constructor takes as input a single **Point** that corresponds to the non-right angle vertex, integers for the height and base width, a Color object, and a boolean determining whether the shape should be filled.
- The given point and the base length and height used to determine the other two points of the triangle. Specifically, if the specified width is positive, then the second vertex is placed to the right of the first **Point**. If the width is negative, then the second vertex is placed to the left. Likewise, if the specified height is positive, then the third vertex is placed below the base. If the height is negative, then the third vertex is placed above the base.

Rectangle

This class maintains information for drawing a Rectangle and extends Polygon.

- The constructor takes a single **Point** for the upper left corner, integers for the width and height, a Color object, and a boolean determining whether the shape should be filled in. The point for the upper left corner is used to determine and initialize the other points of the rectangle.
- This class explicitly represents all four vertices.

Square

This class maintains information for drawing a Square, and extends Rectangle.

- The constructor takes a point for the upper left corner, an integer for the edge length, a color object, and a boolean determining whether the shape should be filled in. The point for the upper left corner is used to determine and initialize the other points of the square.

Notes

- Implementation is provided in full for the **Circle, Diamond, Polygon,** and **ShapeUtils** classes. Complete all other classes, using the TODO comments for guidance. (**JPanel** and **JFrame** are classes defined by the Java SWING API; do not create these classes, as your code will not work)
- Write JUnit tests for all the shape classes.
- DrawPanel is the **only** class that provides a *paintComponent()* method. The shapes are drawn because this method calls their *draw()* methods. Note: you will not explicitly call this paintComponent() method as it is invoked by Java's painting subsystem whenever a component needs to be rendered.
- DrawFrame is the main entry point of the program (it provides a main() method).
- For debugging, the call stack trace can be very long. Start from the top and find the lines of your code that led to the error. From there, start inspecting at the first line of your code, to the top of the stack.

Final Steps

1. Generate Javadoc using Eclipse.
    - Select *Project/Generate Javadoc...*
    - Make sure that your project (and all classes within it) is selected
    - Select *Private* visibility
    - Use the default destination folder

  - Click *Finish*.

2. Open the *lab10/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that that all of your classes are listed and that all of your documented methods have the necessary documentation.

3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission Instructions

Before submission, finish testing your program by executing your unit tests. If your program passes all tests and your classes are covered completely by your test classes, then you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due at 7pm on Monday, October 30. **Submission must be done through the Web-Cat server.**

- Use the same submission process as you used in lab 1. You must submit your implementation to the *Lab 10: Java Graphics* area on the Web-Cat server.

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 40 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

Style/Coding: 25 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

If you do not submit compiled Javadoc for your project, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

Bonus: up to 5 points

You will earn one bonus point for every two hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose ten points for every minute that your assignment is submitted late. For a submission to be considered *on time*, it must arrive at the server by the designated minute (and zero seconds). For a deadline of 9:00, a submission that arrives at 9:00:01 is considered late (in this context, it is one minute late).

After 15 submissions to Web-Cat, you will be penalized one point for every additional submission.

For labs, the server will continue to accept submissions for three days after the deadline. In these cases, you will still have the benefit of the automatic feedback. However, beyond ten minutes late, you will receive a score of zero.

The grader will make their best effort to select the submission that yields the highest score.