Lab Exercise 10
Java Graphics
CS 2334

April 11, 2019

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, colors, and behavior. 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 an image of any animal, character, or scenery of your choosing (of course I drew Batman :]). You should not have the same image as any other student. Be creative and have fun!

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
4. Writing images to files

Proper Academic Conduct

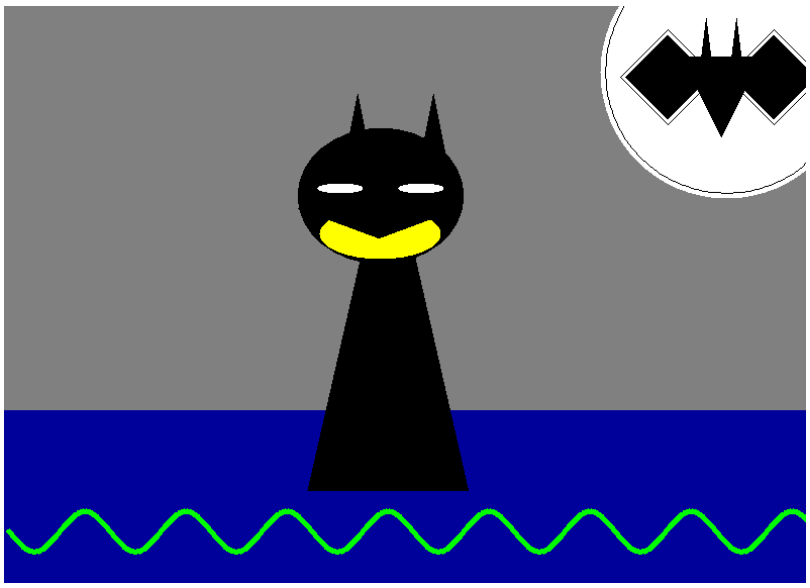
This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Import the existing lab10 implementation from canvas.
   (a) In Eclipse, select *File/Import*
   (b) Select *General/Existing projects into workspace*. Click *Next*
   (c) Select *Select archive file*. Browse to the lab10.zip file. Click *Finish*

Image

Below is the illustration along what you should do.



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

If you want, for extra credit you could animate the image.

Specifically, your picture should contain at least 15 shapes. The image above has:

- 4 instances of **Circle** or **Oval** for the head, face, and eyes
- 2 instances of **Circle** for the light of the bat-signal
- 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
- 1 instance of **Sinusoid** for the grass?

You do not need to worry about matching the example picture above exactly. You *must* use all of the Shapes listed in the UML at least once, and then any additional Shapes you deem necessary to create your master piece.

You should first write unit tests (one method at a time) to cover all your code, and then one method at a time, one class at a time, implement the Shapes to pass all your tests and update your tests as needed.

Be wary while constructing your Sistine Chapel, as some shapes will overlap, others either partially or entirely, so the order in which you add them to the panel is important. You will not receive credit for a Shape that is technically in the panel, but is not visible.

You will embed your image into the README.md file of your git repo. See the GitHub discussion on Canvas for how to do this in markdown.

Lab 10: Specific Instructions

Only a subset of the classes shown in the UML are provided in lab10.zip. Implementations are provided in full for: **Circle, Diamond, DiamondTest, OvalTest, PanelToIMage, PolyLineTest, ShapeUtils, SinusoidTest, and Triangle**. Implementation for **Sinusoid** is incomplete. Create the other classes and interfaces. Write JUnit tests for all the shape classes. Coverage is not necessary for the draw methods, all other methods should show 100% coverage.

1. Implement JUnit tests to check the non-drawing aspects of the 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.

2. Create the missing classes and interface.

3. Create the code according to the documentation, fix any bugs, and update and/or include javadoc comments as needed.

4. 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.

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 *your masterpiece*.

- The constructor takes a string for the title of the window. This method is where you will be creating and adding shapes to draw your image. 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 the draws each of them.

- *addShape(Shape shape)*: This method adds shapes to the this component's 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.

See the javadocs for details on the other classes

Notes

- Write JUnit tests for all the shape classes.
- Implementation is provided in full for the **Circle, Diamond, DiamondTest, OvalTest, PanelToIMage, PolyLineTest, ShapeUtils, SinusoidTest, and Triangle**. Implementation for **Sinusoid** is incomplete. Complete all other classes; see the documentation for guidance. (**JPanel** and **JFrame** are

classes defined by the Java SWING API, do not create these classes, as your code will not work)

- 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 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, and where the lab will be executed from.
- If our code has a bug, fix it. Fill in comments as needed.
- For debugging, the call stack trace will be very long. Start from the top, and go through the stack and look for the lines of your code that led to the error, and start inspecting at the first line of your code, from the top of the stack.
- You can still use the Debugger with graphics, just be aware that there are a lot more variables being created to render the visuals.

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

Make sure you are committing and pushing working code frequently to git. 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 submit. Print out the report of your unit tests and push that to git as well. Create a README.md file and embed the image of your masterpiece into the document. See the GitHub discussion on Canvas for more information.

- All required components (source code and compiled documentation) are due at 11:59p on Tuesday, April 16.

Rubric

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

Correctness/Testing

Test and verify your code.

Style/Coding

Use good programming practices, such as meaningful variable names, and a single line of code should *not* be more than 100 to 120 characters long. If it's too long you should probably break it into multiple lines of instructions.

Design/Readability

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)

Push your compiled Javadocs to your git repo.

Bonus

You will earn a bonus if you animate your drawing and save it into a .gif file. Embed the .gif instead of the static image to the README.md file on git.

Penalties: up to 100 points

You will lose ten points for every minute that your assignment is submitted late.