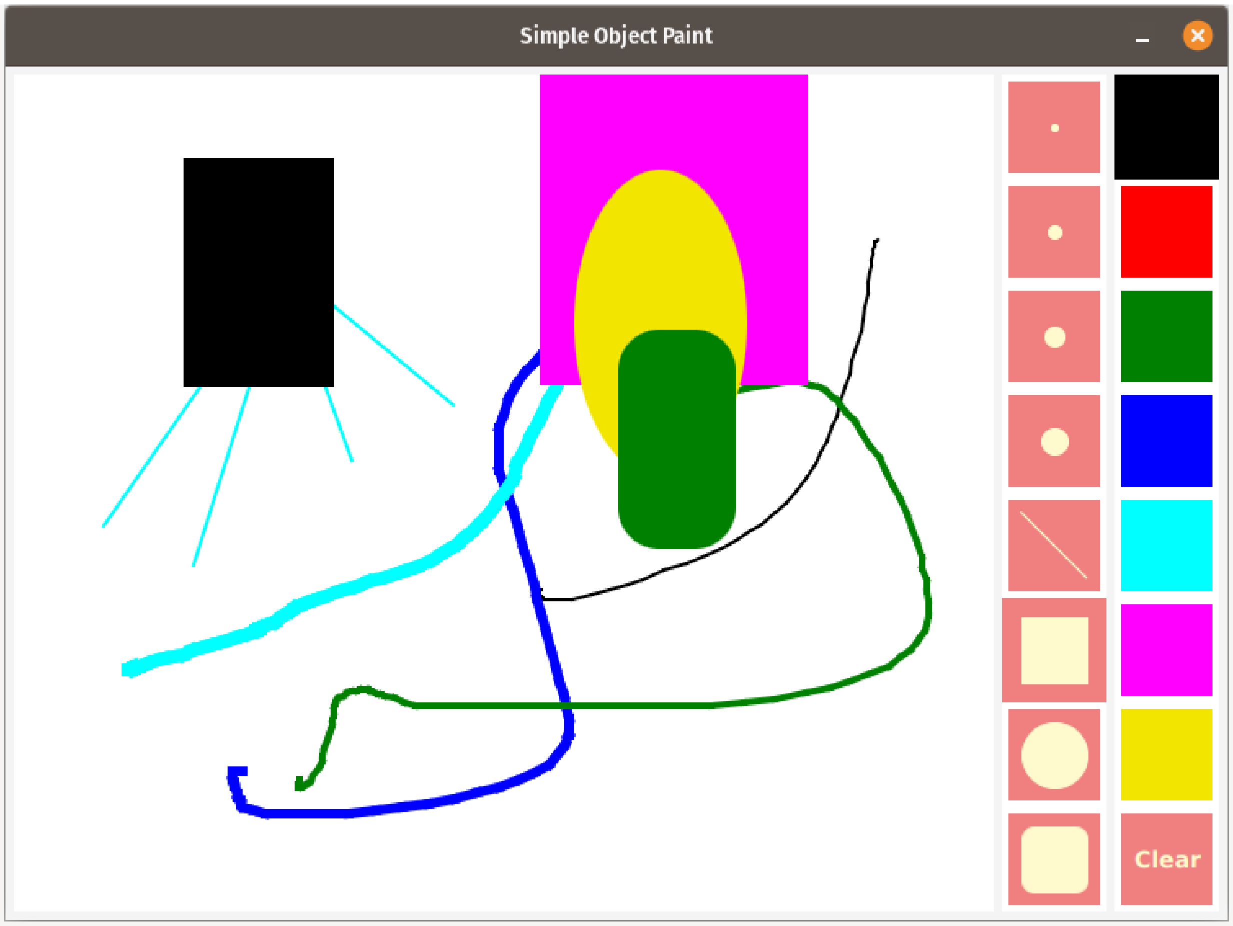
1 Introduction

V.001

The first homework assignment introduced you to the basics of graphics and the canvas object in JavaFX. The simple paint application that you created allowed you to draw shapes on the canvas in real time, however, all changes were committed immediately to the canvas as you drew.

In this second homework assignment you will only commit the shape objects when the mouse is released. However, while the mouse is pressed, you will still be able to see the object form in real time. This will allow you to both grow and shrink each shape object until you are satisfied with the object's size.

The goal of this second assignment is to give you some experience working with applications that have a more complex class structure while you learn more about JavaFX. For this assignment the class structure is specified precisely for you and you must follow the given structure to the letter. A key part of the *puzzle* for this assignment then is to determine how to effectively use the parts of the structure that is given to you. The example that you see below implements this structure exactly. In other words, the UML in the class diagram below was taken from the solution that generated the following image.

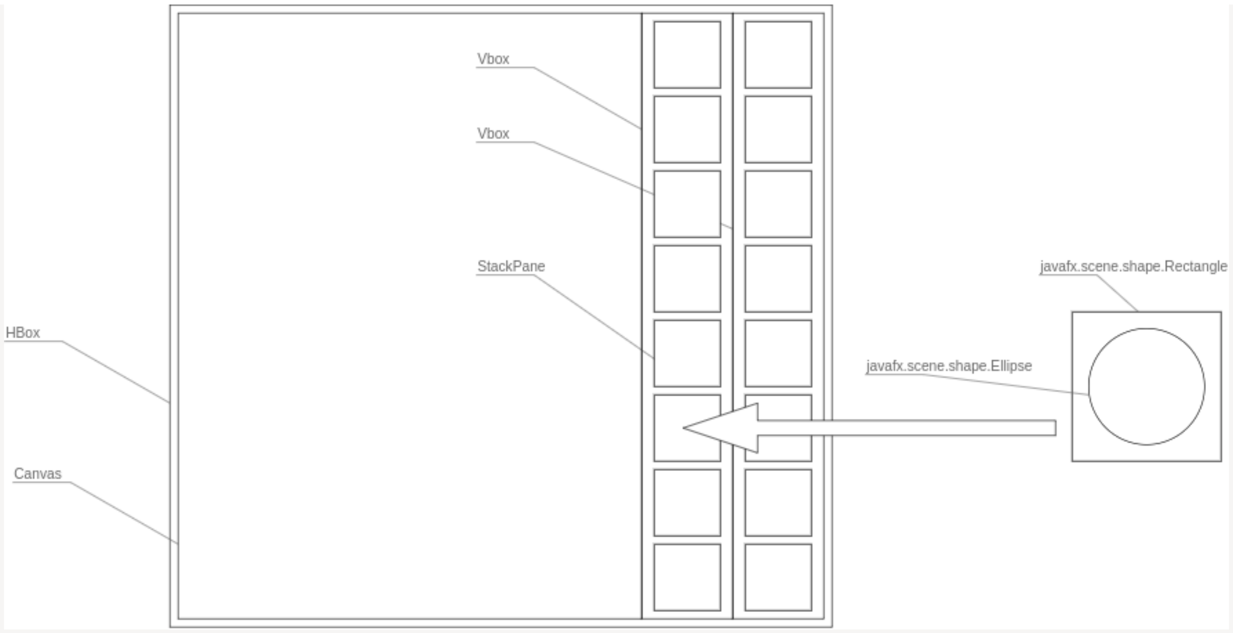


2 Application Structure

We need to discuss how this application will be organized. Before we get into the details of the graphics objects as displayed on the canvas, let's first discuss how we will implement the user interface. In the prior assignment the tools were drawn on the canvas and you used the mouse location to determine which tool the user was interacting with. In this assignment we will use a *scene graph* and build up the top layer of the application using JavaFX

2.1 Layout Panes and the Scene Graph

You will use the concept of *layout panes* to layout the application. The *root* node of the application should be an *Hbox*. The *HBox* layout pane places elements side by side in order of insertion. So first insert the *canvas* object, then two *VBox* panes to hold each of the columns of tools and colors. The *VBox* pane orders elements vertically in the order that they are placed within the pane.



Finally, to create each tool you will start with a *StackPane* that stacks objects on top of each other in the order that they are placed in the pane. Hence, the first object that you place in the pane is a *javafx.scene.shape.Rectangle* object that represents the background square of the tool. You will set the color and shape of this object to reflect whether or not the tool is either a shape or color tool. Finally, for the tool objects, you will place another *javafx.scene.shape* object over the top of the rectangle, e.g., for the Ellipse/or Oval tool, you will use a *javafx.scene.shape.Ellipse* object.

2.2 Graphics Objects and the Canvas

The next thing that we need to determine is the structure of the graphic shapes and how we will draw, store, and manage these shapes. In particular, we need to think about how we can constantly redraw the shape that we are currently exploring without committing it to the canvas, something that we could not do in the prior version of this application.

2.2.1 Layers

One way to accomplish this would be to overlay a transparent canvas on top of the canvas where the drawing is located. Shapes could then be drawn and erased on the transparent *overlay* without disturbing the underlying canvas that contains the drawing. Once a shape is committed, the drawn object could be drawn on the lower canvas committing it to the painting and the top canvas could then be erased in preparation for the next shape. This is a simplified *layers* concept that is prevalent in many painting and drawing programs.

2.2.2 Object Lists

Another way to accomplish this would be to constantly redraw all objects currently on the canvas every time there is some mouse movement. Then only when the mouse pointer is released is the current object committed to the list of objects. This latter approach requires us to maintain a list of objects that are currently drawn on the screen. While this may not seem as efficient, in practice it works fine and has the advantage that we can work with the list of drawn objects. Since it supports learning goals for this course, we will choose this approach.

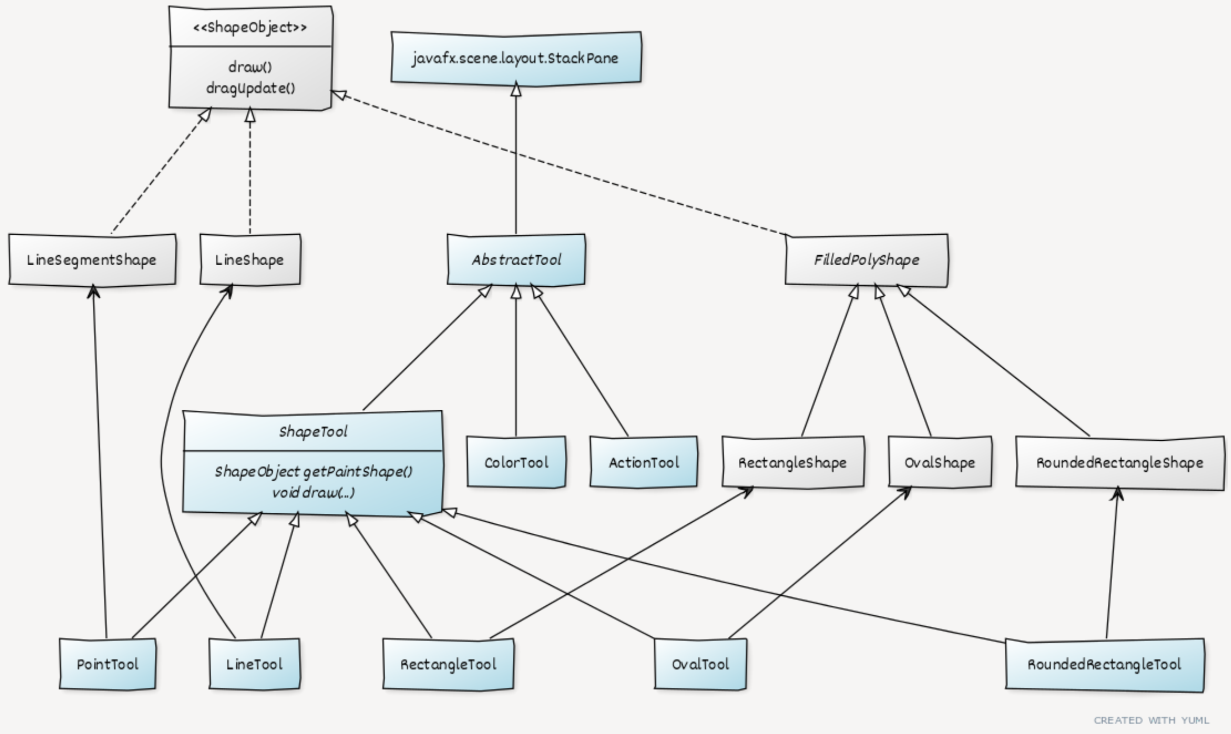
2.3 Shapes (in general)

We know that we need to maintain a list of objects, but what kinds of objects? I will warn you in advance, there are multiple objects in this assignment that appear to represent the same kind of graphical shape, however, each *Class* and its associated hierarchy serves a particular purpose.

2.3.1 Tools

The tool classes extend *javafx.scene.layout.StackPane* and as they are *nodes* in your *scene graph*, they can be directly inserted into the *VBox* pane. Each tool class must build up the structure of the tool that you see on the tool pane. It has other responsibilities that I discuss below. First let's take a look at the basic *class diagram* for the structure of the tools and shapes within your applicaiton.

2.3.1.1 Basic Class Diagram



This diagram is not complete in the sense that there are other methods, fields. and relationships within the project. However, it is complete enough to give you a basic structure. Further, it describes all necessary classes except for the main class, described below. For this assignment you must implement this class diagram. Consider this aspect of the assignment similar to what you might experience in the work environment when you have to work with a structure designed by someone else.

The tools class hierarchy described above is shown in the diagram in blue. Note that it interacts with the *shape* hierarchy which we have not discussed yet.

2.3.1.2 *AbstractTool*

The class *AbstractTool* is, as you should expect, an abstract class, i.e., a class which cannot be instantiated, that is the base class for other tool classes. You will want to instantiate a *javafx.scene.shape.Rectangle* object to represent the background of the tool and you will want the ability to set this Rectangle's color. Since all tools can be both activated and deactivated, you will want to implement methods here to activate and deactivate the tool. Note that this only involves changing the properties of the base Rectangle.

2.3.1.3 ActionTool

For this assignment there is only one *ActionTool*, namely, the clear button. An *ActionTool* object performs an action when depressed. Since this action may be a method outside of the Tool class hierarchy, you will need to pass this action in as an object of type *Runnable*. This is much easier to do than it sounds. Simply pass a method reference, e.g., *this::myClearAction* as the parameter to the constructor. In your main class you will implement a method *myClearAction()* that implements the clear function. We will discuss how this works in more detail later in the course. In order to execute the action on a mouse press, you will need to call the *run()* method on the *Runnable* object that you have stored within your *ActionTool*.

2.3.1.4 ColorTool

A *ColorTool* sets the color within the application. When a color activates, it should deactivate any other *ColorTool* objects. There are a number of ways to do this. One reasonably clean way is to store the active *ColorTool* object in your main class. Thus, you only need to deactivate the current tool, set the current tool to the new *ColorTool* object, then activate the new tool. Note it is not necessary to also save the color that the tool represents, rather, implement a getter for the color to be called, as necessary, on the current color tool.

2.3.1.5 ShapeTool

A *ShapeTool* represents one of the shapes to be drawn. As with the *ColorTool* you will want to actively track the current shape tool in your main class. The base *ShapeTool* is not complex, it simply sets up the tool rectangle with the appropriate color and provides whatever minor services derived classes may need. However, it provides one mandatory abstract method, namely, *draw()*. The *draw()* method must be implemented in each of the derived shape tool classes. It is this method that draws the shape on the canvas. Note, however, that, within the tool hierarchy, this method delegates the actual drawing to the *ShapeObject* hierarchy described below. Note also that the *draw()* method is not parameter-less. At minimum, the graphics context must be passed in but it will also need to know the start and end points of the shape as well as the color of the shape.

2.3.1.6 PointTool, LineTool, RectangleTool, OvalTool, and RoundedRectangleTool

Each of these tool classes implements the tool for their respective shape. The *PointTool* class implements the top four tools that draw points, really line segments, continuously as you drag the mouse. Its constructor should take a size parameter. There is one key difference between *PointTool* and the other tools. The *PointTool* object must save every drawn object to the list of objects to be drawn in your application. In other words, the shapes that it draws are committed to the list of drawn objects on every call. The other shapes do not share this property. They are only committed to the drawn object list when the mouse is released. I discuss how this works below in the discussion of the *ShapeObject* classes.

However, what this highlights is that the tools themselves must be able to maintain a reference to, and draw, the shape that they represent. The actual drawing is delegated to the referenced shape object, however, the reference to the current shape object should be stored in the tool class. While this is not the only way to implement this behavior, for now, it's a good choice. We'll discuss this idea further in class. For now, know that each tool class should hold a reference to a shape object that represents the current object being drawn. Further, every time the tool is asked to draw that object, the object is reinitialized with a new object based on the passed parameters.

Each tool must provide a method to get the current paint shape from the tool. As discussed below, all paint shapes share a common interface. This is important for storing the shapes in a list. Because we won't know the actual type of the shape when referencing the current tool within our main class, we must provide a common *service* for obtaining the shape from the tool. This is provided by the *ShapeObject getPaintShape()* method. A partial implementation of these ideas for the *LineTool* is shown below. You may use this exactly if you wish.

```
@Override
public void draw(GraphicsContext g, Color color, Point2D start,
      Point2D end) {
    currentLineShape = new LineShape(start, end, color);
    currentLineShape.draw(g);
}

@Override
public ShapeObject getPaintShape() {
    return currentLineShape;
}
```

Notice that every call to the tool's *draw()* method creates a new *LineShape* object and then delegates to the shape object to draw that line. Thus, whenever the mouse is moved in the main application, a new *LineShape* object is created and then drawn. When the mouse is released, the main application asks the current tool for its *ShapeObject* so that it can be stored in the list of currently active objects.

The other *ShapeTool* objects are very similar.

2.3.2 ShapeObjects

The JavaFX framework defines objects for use in the scene graph. You will use such objects to create the tool icons for this application. However, the canvas is not a scene graph and so we are not going to use these objects to represent the shapes that we draw on the canvas. JavaFX also provides methods to draw shapes on the canvas, however, these methods belong to the *GraphicsContext* and that won't help us create a list of shape objects to draw on demand. Hence, we must create our own hierarchy of shapes that we will store in a list and draw on demand.

As we discussed above, we cannot easily erase and restore what is on the canvas between the drawing of each shape. Well, in fact we can, but that's a story for another day. For now, we want to simply keep a list of objects that represent the image on the canvas so that we can redraw them on demand. Every time a mouse event fires to indicate that the mouse is being pressed down we want to respond by redrawing the canvas. To do this we first clear the canvas, then iterate over our list of objects drawing each object in turn, then finally we want to invoke the draw method of the current tool to draw the currently selected shape. When the mouse is released we ask the currently selected tool to give is the last shape that was drawn and we add that to our list of drawn shapes. For the *PointTool*, however, it must constantly add the last drawn object to the list of objects and not wait for mouse release. Since the tools have no knowledge of your list of drawn objects, they cannot execute this behavior themselves. This is the purpose of the required *dragUpdate()* method which returns *true* for the *PointTool* and false for all other shape tools.

2.3.2.1 *ShapeObject*

The *ShapeObject* interface defines the methods that all *ShapeObjects* must implement. This structure is outlined in gray in the class diagram shown above. Each shape object must know how to draw itself when provided with a *GraphicsContext* object. Hence the method to draw each object is, precisely, *void draw(GraphicsContext g)*. This means that each shape must know its origin, dimension, and color. The interface also defines the *boolean dragUpdate()* method that returns true if the object should update the objects list while the mouse is dragging.

2.3.2.2 LineSegmentShape

The *PointTool* draws *LineSegmentShapes* which are nothing more than lines extending from wherever the mouse was when the last mouse event fired to wherever the mouse is when the current mouse event fired. I concede that the class names could use some improvement here, but such is the nature of software development. Sometimes we have to freeze a decision and deal with the *technical* *debt* at a later time. *LineSegmentShapes* have a width and color as well as a start and end location.

2.3.2.3 LineShape

The *LineShape* class isn't all that different from the *LineSegmentShape* class except that its width is fixed and it is not updated on drag. This is created by the *LineTool* object.

2.3.2.4 RectangeShape, OvalShape, RoundedRectangleShape

These classes create the three filled polygon shapes. The only significant difference between them is their type and the slightly different calls to the *GraphicsContext* method to draw the shape. Hence, they are all derived from a base class *FilledPolyShape*.

2.3.2.5 *FilledPolyShape*

This class represents the detail of the filled polygon shapes. Make sure that you think carefully about how to correct the size of the polygon based on the mouse coordinates. This correction is identical for all filled polygon shapes, hence the code to do this should be in this base class.

2.4 Overall Structure

I have described the main class hierarchies for this application. The remainder of the application is contained in the main class which should be called *SimplePaintOjbects*. Your job here is to create and show the scene graph, setup handlers for mouse events, and maintain and draw the list of *ShapeObject* graphics objects.

There are some hard requirements with respect to programming for this assignment.

2.4.1 Constants

For this assignment you must use static defined constants in lieu of magic numbers for virtually all constants in your code. I've given some samples below and for these samples, the names, as given, are required. While I want you to have some freedom with respect to introducing your own style to the projects, for this assignment, please use the colors that I've given below to ensure that my grader can quickly check other aspects of your code. This is not a full list of constants, you will want to add other constants as needed. You must define your canvas height as an expression related to other constants. The idea here is that if you changed the cell size *(CELL_W)* that your canvas height would adapt accordingly. Note that *CELL_W* could have been as easily called *CELL_H* as the cells are squares.

```
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// constants
//
static final Color TOOL_RECT_FG = Color.LIGHTCORAL;
static final Color TOOL_RECT_BG = Color.WHITE;
static final Color TOOL_FG = Color.LEMONCHIFFON;
static final int CELL_W = 60;
static final int PADDING = 5;
```

2.4.2 Private, Final, Getters, and Setters

As discussed in class you must always use private member variables. In addition you must not implement any pure setters in this application, nor must you implement any unnecessary getters. Obviously, you will need to use many setters and getters from the JavaFX framework, e.g., *g.setFill(someColor)*. This is not a problem, but it is not necessary to implement any setters in your own classes. In addition, we will be practicing using as much immutability as possible, hence, you should use final as much as possible.

My advice to get started is to begin with non-final, non-private and non-public variables. That is, just use the standard package-protected access and get your code working. When it is working, start making your variables private and fix what breaks. Once this is working start making your variables final and fix what breaks.

2.4.3 Comments and Methods

I expect you to lift blocks of code to methods as much as possible. If you are identifying a block of code with a comment, then create a method that makes use of the comment in its name and move that code into the method. I want to see a greater number of small methods as opposed to many large methods. You will find, over time, that this approach is far easier to debug. Sometimes you create a long method while you're initially getting something working, that's ok. Once it works, break it up.

2.4.4 Classes and Files

As with the first assignment, you must put all of your classes in a single file. For this assignment your file name will be *SimplePaintObjects.java*. Note: you cannot use the IntelliJ *create new class* action to create new classes in this way as it will create a new file. Just type the name of the class, without the public modifier, directly in the editor.

3 How to Approach this Assignment

I highly recommend that you break this assignment up into a number of phases.

3.1 Stage 1 :: The U/I

First get the U/I working. This means implementing the tool hierarchy, and getting the mouse interaction with the U/I working. The tools don't have to do anything at this point, but the will look correct and will be enabled/disabled appropriately. You do not need to implement the shape hierarchy to make this work.

3.2 Stage 2 :: Basic Painting

The next phase would be to get the *PointTool* working correctly. This does not require that you do anything special with the *mouseUp* events and will give you some practice with the integrating the two class hierarchies as you work out some details.

3.3 Stage 3 :: Bring in the LineTool detail

Now get the *LineTool* working. This will required that you add the collection of *ShapeObjects* to the main class and properly implement the response to *mouseUp* events. Do not proceed further until this works as expected.

3.4 Stage 4 :: Adding the detail for the FilledPolyShape Tools

Now get the other tools working, do them one at at time. If you don't know what to put in the *FilledPolyShape* at first then leave it blank except for the required methods. After you implement two of the three derived classes you should be able to see the common elements that should be lifted to the abstract parent class *FilledPolyShape*.

3.5 Stage 5 :: Refactor

Finally, you should refactor, rethink, and clean. The code isn't finished when it works. You should always leave time to refactor. This is when you should be lifting blocks of code to methods, making member variables private and/or final, etc.

4 Submission and Requirements

4.1 Requirements

- All member variables must be private
- No pure setters are allowed
- Use only necessary getters
- All member variables that can be made final should be made final
- All numeric constants must be defined as static final member variables in the main class.
- All static final constants must be all uppercase with underscore separators
- You must limit your code to 80 characters (width) so that it is readable in Speed Grader. We will talk more about this as we go forward and it is good practice to improve the readability of your code. I will discuss this briefly in class.
- Your code must be neat and properly aligned. Sloppy code will keep you out of the top spot.
- For this assignment, you should minimize comments for most methods and use method names themselves to document your code. That said, you may use larger block methods at strategic locations in your code to describe major structure.

- You must include your entire program in a single Java file. All required classes should be included as non-public classes in the same java file. There is no need for any additional classes beyond what is described in this document. If you choose to use them, however, they must also be non-public and included in your single Java file
- Your java file submission must be entitled SimplePaintObjects.java

4.2 Submission

You will submit three files:

- Your SimplePaintObjects.java

- A file called SimplePaintObjectsWriteup.pdf. This file will contain a brief writeup for this assignment. Your writeup itself should fit on one page. Do not overthink this. In your writeup, give a brief summary of your solution and then answer the following questions. For each question you must include the question in your writeup and you must format the question so that it is different from your answer, I suggest that you simply italicize the question.

  1) How did the additional code requirements hinder your solution to this problem?

  2) What are some advantages to these code requirements going forward?

  3) Where do you think the required class hierarchy gets in the way of code reuse?
  (To think about this problem, where did you find the heirarchy tedious in terms of typing code that is almost the same from one class to the next, e.g., the classes derived from FilledPolyShape*)*

  Finally, in an appendix, labeled as such, include your code. Yes, you will be including your code in your submission twice. The java file is so that we can run your code, the pdf is so that we can see your code while we grade the answers to your questions. Make sure that you code is properly formatted and super neat inside of the pdf. A little extra effort here goes a long way in terms of not eliciting grimaces from the grading team while grading your work.

- A file called SimplePaintObjectsDemo.mp4 that is a video of you briefly demonstrating you compiling, running, and using your paint program. Demo as much as you can in a program that is strictly less than two minutes in length. You must narrate your video and tell us what you're doing.

4.3 Extra Credit

There are up to 10 (ten) extra credit points available for this assignment. To obtain these points you must submit a working version of your java file and a sub 1 minute video by 5pm on the first Monday following assignment release. You do not need to submit a pdf for the extra credit. This will give you about a week to get your code ready. The points are granted as follows:

- 4 Points : You have achieved stage 1 :: Basic U/I
- 6 Points : You have achieved stage 2 :: Basic Painting
- 8 Points : You have achieved stage 3 :: Bring in the LineTool detail
- 10 Points : You have achieved stage 4 :: Adding the detail for the FilledPolyShape Tools

The idea here is to encourage you to start early. The sooner that you realize that refactoring is the secret to good programming the better. The more time that you have for refactoring and, for this course, engaging with the reading material and assignment questions, the better.

Note: this is Release V001. There will be at least one more release. I really encourage you not to wait to get started. The thinking that you will somehow avoid mistakes by waiting for the final version is broken. You will learn more through the process of adapting any changes. I cannot emphasize this enough. Broken thought processes regarding what programming is really about is right up there with procrastination in terms of the pattern of failure in this course.

5 References

The UML diagram was created with **yUML.me**. I highly recommend this as a quick tool to generate UML diagrams to help you think about code structure.