The City College of New York

CSC 221 – P || Professor Hesham Auda

Exercise 1: MyShape Class

Student: Binyamin Radensky

Contents

[Introduction 2](#_Toc21880216)

[Class Main 2](#_Toc21880217)

[Class MyShape 5](#_Toc21880218)

[Class MyLine 7](#_Toc21880219)

[Class MyRectangle 8](#_Toc21880220)

[Class MyOval 9](#_Toc21880221)

[Enum MyColor 10](#_Toc21880222)

[Conclusion 12](#_Toc21880223)

[Works Cited 15](#_Toc21880224)

[Code - Main 15](#_Toc21880225)

[Code – MyShape 17](#_Toc21880226)

[Code – MyLine 18](#_Toc21880227)

[Code – MyRectangle 18](#_Toc21880228)

[Code – MyOval 19](#_Toc21880229)

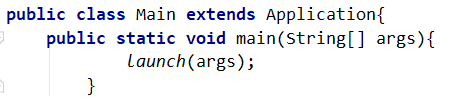
[Code – MyColor 20](#_Toc21880230)

# Introduction

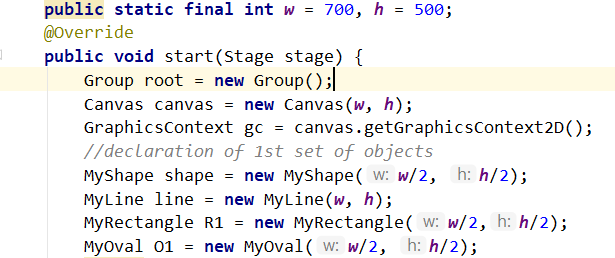
The goal of this project was to familiarize us with the basic workings of Java and JavaFX such as graphics context, canvas, stage, scene, paint, and most importantly, inheritance. We also learned how to use enum types by creating our own MyColor enum. Important aspects such as overriding methods, using static and non-static variables and methods, using wrapper classes, importing methods from other classes and using objects were some of the many tools we learned from this project. The details of what each piece of code does and how the program as whole works and how the code was developed is detailed below. Additionally, the complete code can be found at the end of this report. In preparation of starting this project several pages of the oracle documentation were read as well as our notes from class, and cross referencing our class notes with the oracle documentation. This project gave us a good understanding of Java and after a close reading of the Oracle documentation and our class notes this project was straightforward and actualized what we had studied in class.

# Class Main

In order to create the application window, set the stage, create the group, and create the canvas we need to extend javafx.application.Application. This is because the stage, which is what we call the application window, can only be started using that class. In order to start everything to our specifications we override the start method from that class. We launch the overridden start function using the launch command in the main method of the Main class.



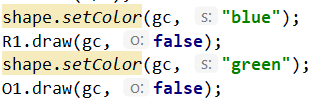
In addition to creating the canvas, the group, and graphics context object, the start method is also where we instantiate, and call the methods of, our custom classes (MyShape, MyLine, MyRectangle, and MyOval). Instantiation calls the class default constructor and creates a virtual object within the start method where it was called. Each instantiation call also passes critical parameters for the creation of the object such as the width and the height of the object on the canvas. We also name the objects in this process which we can then use to call methods in those classes to do operations on our objects. The methods will also allow us to set the colors of the objects, as well as where we place the objects on the canvas.



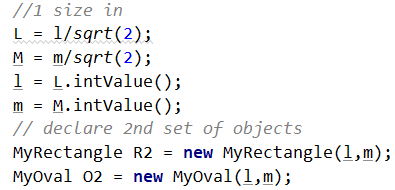
In order to set the value of width and height which will be consistent while we run the programs, we declare width and height as constant static variables. This ensures that width and height aren’t changed by another class or in the start method.

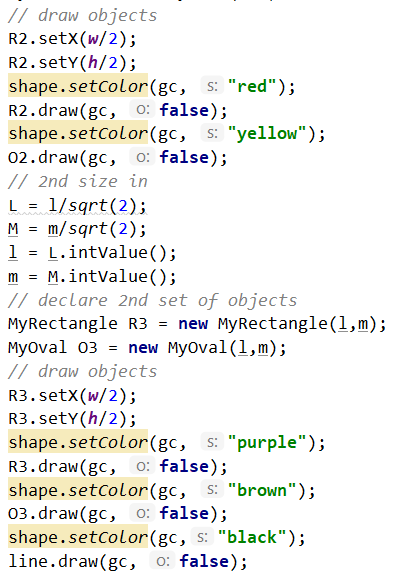


Next, we had to set X and set Y which are the coordinates of the center of our shapes from MyShapes. In order to specify the middle of the canvas we took the middle of the width and length of the canvas. We then passed these values into the setX and setY methods inherited by our shapes subclasses. Importantly, we must call the set methods by the objects of the subclasses since the width and height of the shapes are different than the width and height of the original shape object. Therefore, to calculate where to place the shape on the canvas, it uses the height and width values specific to that shape which are private variables of that shape as we will see later in the sections about those subclasses.

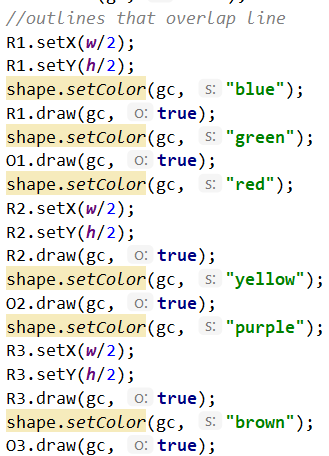
Then we set the color of the object and drew the first rectangle and oval. In order to draw the shapes, the functions need two things. First, the draw() function uses the GraphicsContext (gc) to create the shapes, and it must use the same one that we instantiated in the Main class so that they draw on the same canvas. We also must pass in whether we are drawing the outline of the shape, where we will pass in boolean o = true, or the fill of the shape, where o will be false.

A close up of a map

Description automatically generatedOnce we drew our first two objects, we wanted to draw our next two objects inside of those. In order to do this we found that the width and height for a square to fit into an oval (or ellipse) is Widthnew = Widthoriginal/sqrt(2); height = heightold/sqrt(2); When we instantiate our second set of objects we first used the java.lang.math.sqrt function which returns a double, which we stored in a Double wrapper class so that we could convert it to an integer type easily and pass it into the 2nd set of objects of the shapes that we instantiated.



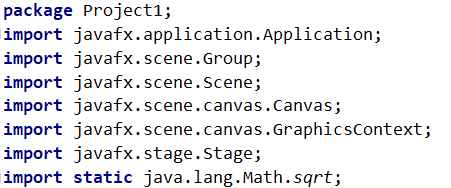
We then repeated the above steps in order to draw all seven, technically 13 when we count the outlines, of our objects. As seen on the right.

We also had to draw the outline of the shapes over the line going across the shapes as can be seen in this closeup of the image from the assignment:

In order to achieve this effect, we redrew the outlines of the shapes, using the same objects and just re setting the color for each one. This time we used the outline part of the draw() function by passing in o (for outline) as true. We can see that doing this creates the same effect below in the image generated by our program.

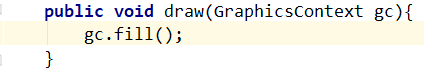
To display what we drew we had to tell Java to first add the canvas to the group we made. Then create and set the scene on the stage using the group we made. Lastly, we show the stage with all our elements on our canvas, in our group, with the scene set to the user.

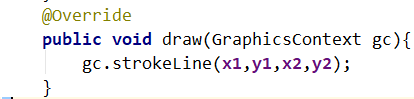
Lastly, to implement all the different methods from other libraries that we used we must import them into our class.

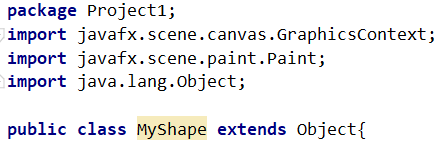


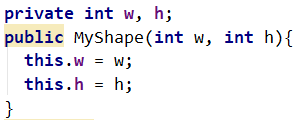
Additionally, we declare that this file Main.java and the classes and methods inside of it are all part of the package Project1. Packages groups together all the classes and methods and ensures that there are no other naming conflicts.

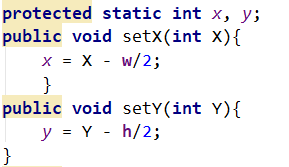
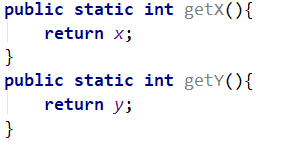
# Class MyShape

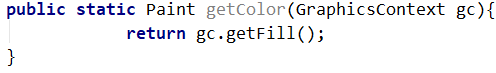
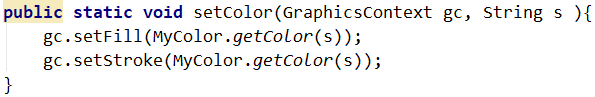
Class MyShape is the superclass, the highest class in the hierarchy, from which all the other custom classes inherit. This means that the MyShape class is the template that all the other classes follow. Any function that you can do in MyShape you can do with one of the subclasses. The difference is that certain methods are overridden, so when you call them in the subclasses, you will get that subclasses’ version of the method instead of the one from MyShape. Any method that isn’t overridden in the subclasses will just call the method from the superclass, in this case MyShape. This can be illustrated using the methods draw() and setColor() as examples.

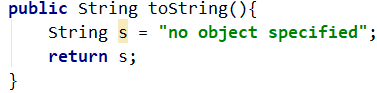
When we call the draw() method from the MyShape class as seen above it will call the method gc.fill() from the GraphicsContext class. However, if we were to call the same method from the MyLine class then gc.strokeLine(x1,y1,x2,y2) will be called instead, as seen to the right.

 Our MyShape class extends the Object class which is part of the java.lang library. This means that MyShape inherits all the methods from the object class and can override any of the methods from that class that it needs to. Any methods that we need that we don’t inherit from the object class we imported from other classes. Additionally, since this is part of our package, we must declare it so.

 The first method of our function is the constructor. We don’t need a main method since we have the main class which is calling all the methods and is the entry point of our application. The constructor initializes the variables and chooses what to do with values passed into the object when it was instantiated. For our constructor we take the width and height values passed in, and we use them as the values of our class variable width and heights. Obviously, as seen above our constructor, we must declare our class variables before we can assign them values.

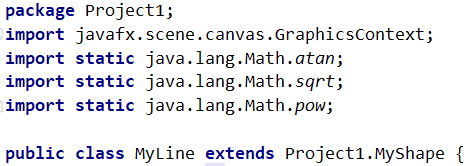
 We have 3 different set\_ functions and 3 different get\_ functions. setX, setY set the coordinates for where the middle of the object should be, and getX and getY return them. setX and setY will change based off the value of the local variables of width and height of the class that the method is being called in. The value passed into them is the coordinate on the canvas where the center of that shape should be. The equation x =X -w/2 basically says that x is equal to the point where the center of the object should be minus the distance from that point to the edge of that object which is based off the specific width and height of that object. We do this since when we call the methods in graphics context to draw these objects, graphics context places these objects based off the coordinate of the top left corner of the object as opposed to the center of the object. Therefore, we calculate the center of the object by width divided by two or height divided by two. We made x and y protected and static so that way its functions and values can be accessed without having to instantiate the MyShape class and so that all the subclasses can access the values of x and y.

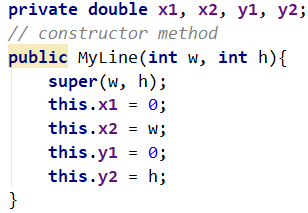
 The setColor and getColor functions can be used to set the fill color and stroke color, and can also return a paint color that can be used to fill an object if need be. All of these colors are managed by graphics context and are retrieved from MyColor enum class as can be seen in the code on the right. In order to return a paint type we had to import the paint class and the graphics context class.

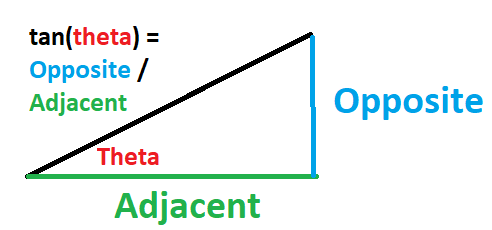
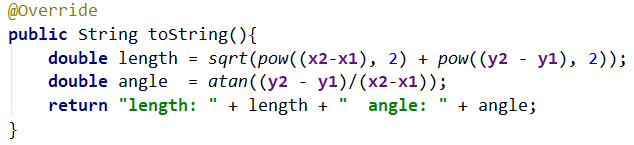
 The toString method in MyShape exists basically as a placeholder. Since my shape does not actually instantiate an shapes itself, there is nothing to describe in a string. Therefore it will return a string saying that no object was specified to be described.

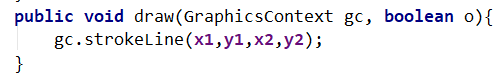
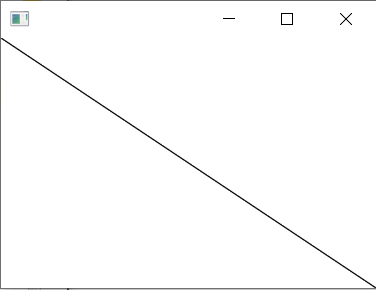
Lastly, we created the draw method that will be overidden in every class. In our MyShape class it just fills the canvas color using the color set by setColor.

# Class MyLine

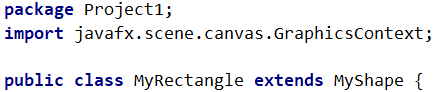
 Class MyLine extends the class MyShape which means that it inherits all the methods of MyShape and includes it functionality. Methods that MyLine needs that it does not have such as the mathematical functions and graphics context for draw(). We also declare that this is part of Project1.

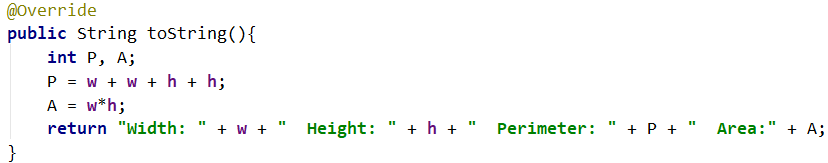
 The constructor initiates all the variables and sets the 4 coordinates of the line. For our implementation we set the values in the constructor itself. We also insert super which takes the width and height that was passed into the constructor of our subclass and uses it on the constructor of the superclass, in this case is MyLine and superclass is MyShape. This is important to that any methods that are inherited have variables that are properly initialized.

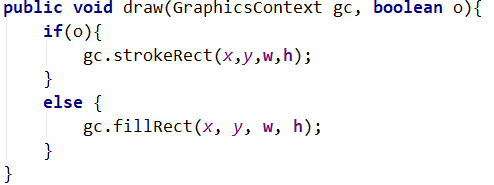
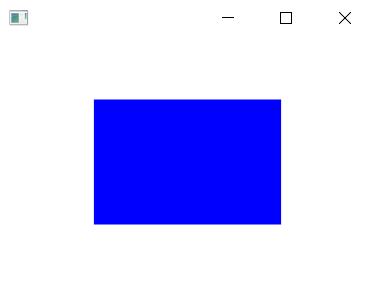
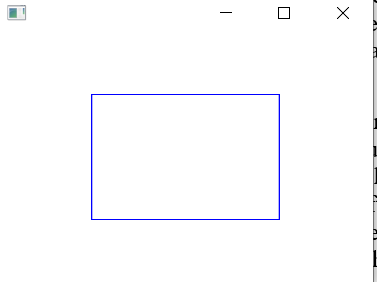
 The toString method overrides MyShape toString and returns a string containing the description of the shape, in our case the length and angle of our line. To find the length we use Pythagorean’s theorem which is A­­­2 + B­2 = C2 where x2 - x1 is the length of the x component and y2 – y1 is the length of the y component and then to find the length of the line which is comprised of both he ex and y component we take the square root of both sides of the equation to get C which we call the length. For the angle we did arc tan of the y component over the x component in order to get the angle. The output of toString when the canvas width is 300 and height is 200 can be seen above on the right.

 Lastly, the overridden draw method which has the parameters of the graphics context and boolean o will draw the stroke line onto the canvas based off the values of the variables set in the constructor. For this class boolean o does not serve a purpose, however since it is overriding the inherited draw method, we put it in. The line made will go from point (x1, y1) to point (x2,y2) on our canvas. As seen below on the right.

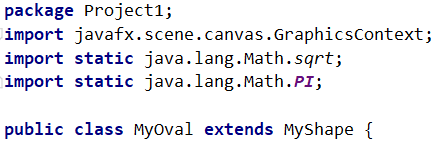
# Class MyRectangle

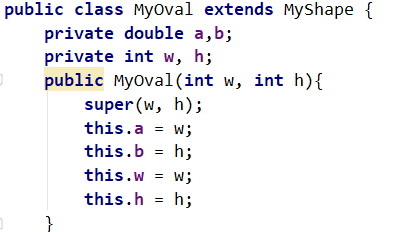
 MyRectangle extends MyShape, is part of the package Project1 and uses graphics context for its draw method as seen on the right.

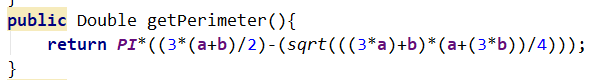
 The overridden method toString returns a string with the width, height, perimeter, and area of the rectangle. We calculated perimeter by adding the length of each side of the rectangle together and the area by multiplying the width by the length of the rectangle. The output of the method can be seen on the right.

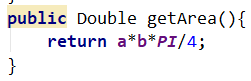
 The overridden method draw has parameters graphics context and boolean o. If the parameter o is true then draw will put the outline of a rectangle of width w and height h with the top left corner of the rectangle situated at the point (x,y) which was created based off the center of the shape that was set and calculated in the methods of the MyShape class. The output when o = true, the width of the rectangle has been set to 150 and the height set to 100. It is centered at the point (150,100) which means the value passed in to the gc.fillRect was 75 and 50. Below on the right is the outline of the rectangle which was created by setting the parameter o to true.

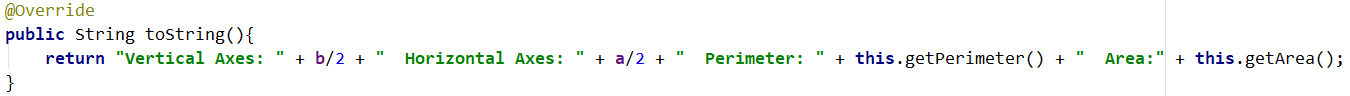
# Class MyOval

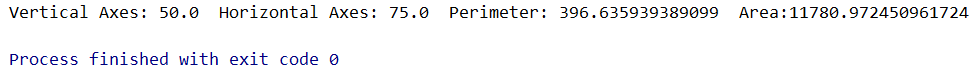
MyOval extends MyShape, is part of the package Project1, uses graphics context for its draw method, sqrt and PI from the lang.Math for its getPerimiter and getArea method as seen on the right.

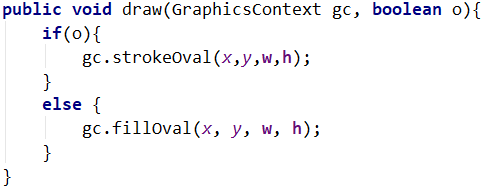
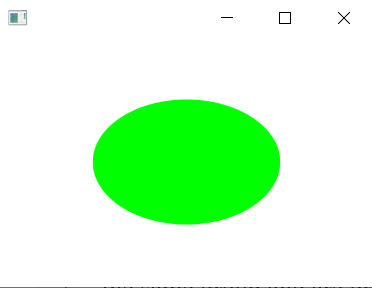
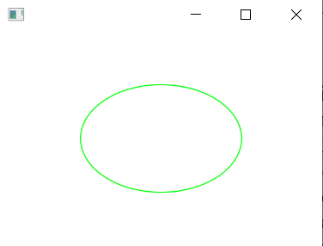
The constructor for MyOval initiates 4 local variables. All of them are initialized to the values w and h which are passed in from the main function. The difference is that a and b are of type double so that calculations made using them are accurate and able to be done with ease. The local variables w and h are integers, they were kept this way for consistency.

getPerimiter is a method that returns the perimiter of the oval as a double type value. We calculate the perimiter of the oval using an approximation made by the mathmatician Ramanujan.[[1]](#footnote-1) This approximation is fairly accurate while only requiring basic algebraic calculations.

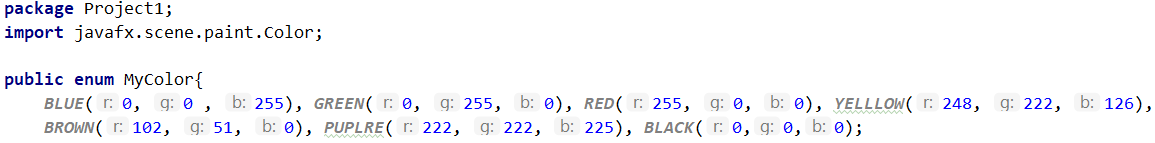
For getArea we used oval Area equation which is the vertical\*horizontal axes \* PI. It is similar to the area equation of a circle expet instead of having one radius we have to different ones that we multiply together. Since both a and b in our method is double the value of the axes they are assosciated with we divided the equation by 4.

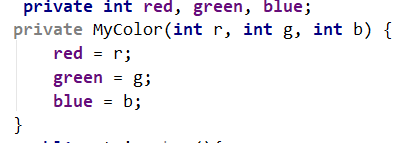
toString is an overriden method that return a string containing the sizes of the axes, the perimiter and the area of the oval. It uses getPerimiter and getArea to get those values and then divides a and b each by two to get the value of the axes. The method can be seen below and the output of the method with w = 150 h = 100 can be seen below it.

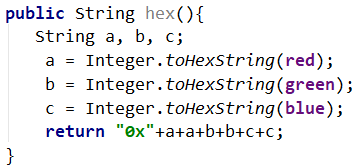


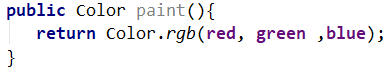
 The draw method is an overridden method that can either draw the outline of the shape onto the canvas or put a solid filled shape onto the canvas. The parameters are graphics context gc and boolean o. Graphics context gc is the instance that was created in main so that they are all part of the same group and are drawn on the same canvas. Boolean o is used to choose between drawing the outline of the oval or a solid oval shape. We use both in creating the final image. To the far right is the solid fill oval when o = false. On the close right is the outline of the oval created by calling draw and passing o as true.

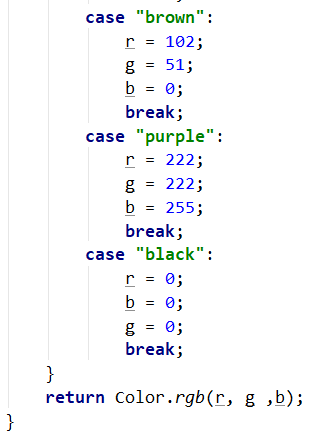
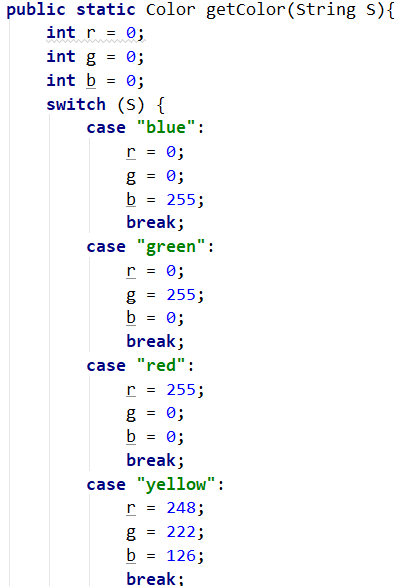
# Enum MyColor

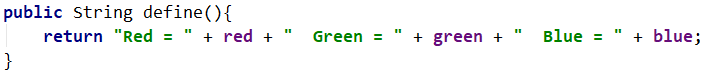
MyColor is an enum type that is used to define, mix, and return colors. It is part of the package Project1 and uses javafx.scene.paint.Color to return usable colors for use with graphics context objects. The header of an enum type has every different option of that type and the values associated with it. That way if we call mycolor.BLUE then the values for blue will be returned. There were several different ways of implementing this. We could have had the associated values for each enum type to be the paint.color value of that color. However, in order to make it easier to return colors with different functions, we made the values integer values of red, green, and blue. 

 The constructor for the enum type can take red, green, and blue integer values and use them as a custom color. We can then use different methods that will make the custom color that was passed in usable for graphics context objects.

 Our first method returns a string with the hex number of the enum color chosen. It is designed to mimic the way that javafx.scene.paint.Color returns chosen colors. The output of brown, red = 102; green = 51; blue = 0, can be seen below. 

 The method paint uses the enum value and will return the JavaFX color type associated with that value. That way we could call mycolor.BLUE.paint() and It would be like calling color.BLUE.

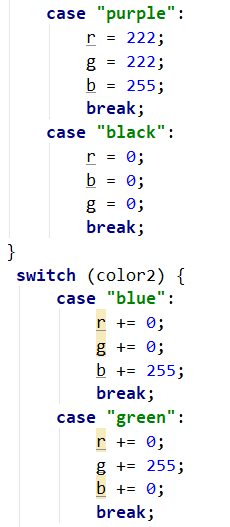
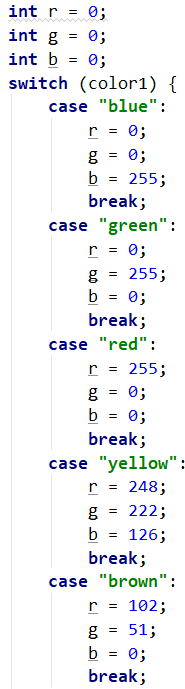
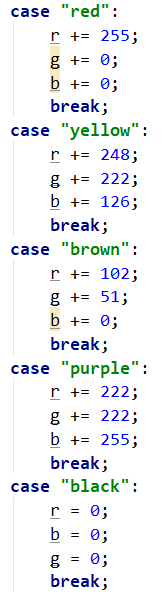
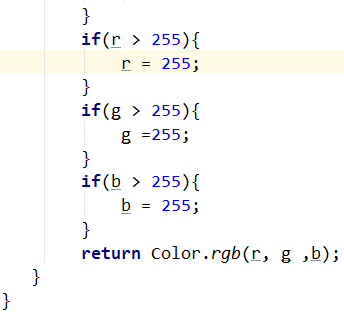
 The method getColor acts as if it is a method in a regular class, and returns color type values that are usable by graphics context. This method was created as a redundancy however, it does simplify passing colors from the Main class through MyShape to MyColor, since now you can write the string name of the desired color, and that string can be sent directly to the MyColor method.

The define method returns a string with the red, green, and blue values of the enum selected color. The code can be seen below and output for the color BROWN is below the code. 



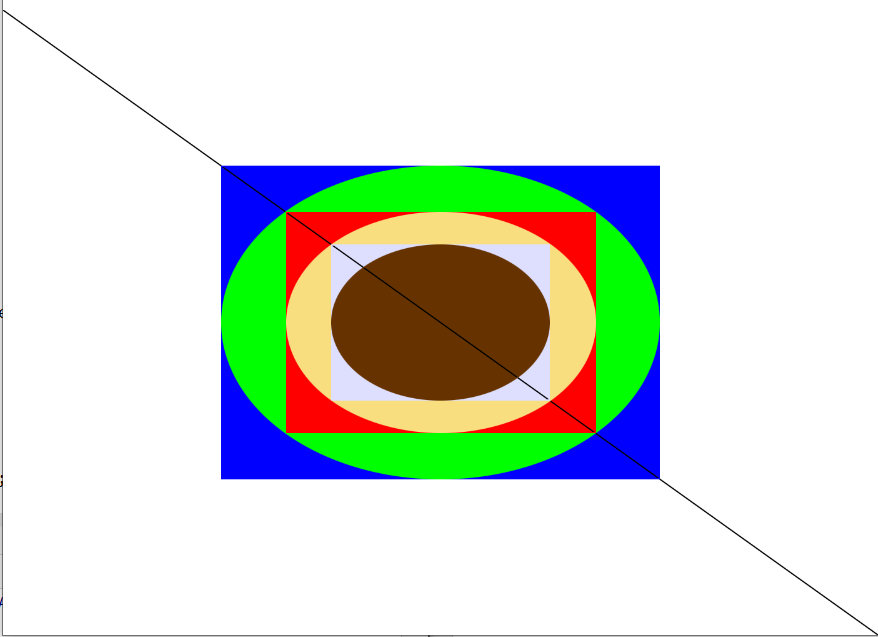
Lastly, the mix method will take to colors, specified by the name in a string value, and add their red, green, and blue values together. If any value goes above 255 it will be set back to 255.



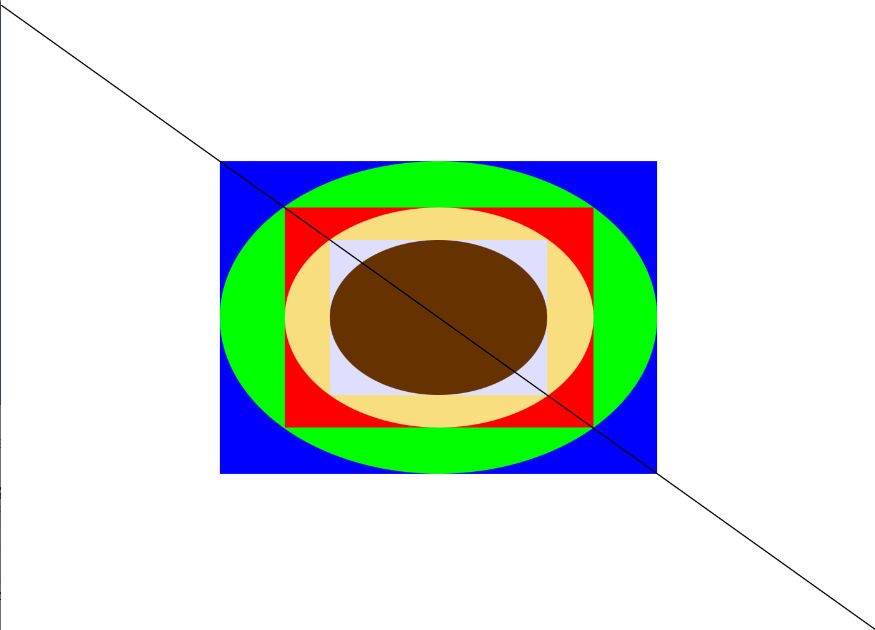
  

# Conclusion

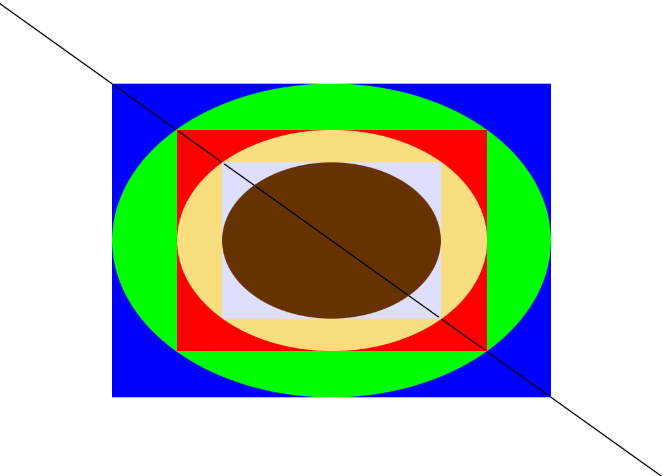
This assignment was a great actualizer for the concepts we have been learning in class. At first, inheritance seemed to be very complicated, but after trial and error, as well as using the Oracle documentation class notes as a reference guide it became much more intuitive. A good example of this would be when using setX and setY. Since the values of w and h were different depending on which object we were calling the methods for, it meant that we needed two different sets of variables, one set for the width and height of that object and another set for the location of the center of the object relative to the canvas. So the values we passed in had to be consistent for all objects being placed at that point, however the widths and heights of those object would changed and depending on if you were calling the method on the correct object, then that abject would be placed properly. Even though this method was inherited from MyShape it would use the local variables that we initiated for the object we were using. Other hurdles came from learning that graphics context will only use a color that comes from the paint class. This mean that we must return paint.Color as a value for MyColor to function with a graphics context object. Part way through the project, after careful inspection of the example picture in the assignment it appeared that the fill shapes were drawn, then the line diagonally across the canvas and then each shape’s outline was drawn on top of that to create an effect where it appears that the diagonal line disappears when it gets to the edge of the shape. In order to create this effect without having to create any extra functions I put an if-else statement in the draw methods for rectangle and oval. Below is the full image created with the outlines on top of the diagonal line.



Below is the image created without the outlines of the shapes on top of the diagonal line.



This project was truly transformative for actualizing our knowledge of the java programming language.



# Works Cited

1. “Perimeter of an Ellipse,” *Perimeter of Ellipse*. [Online]. Available: https://www.mathsisfun.com/geometry/ellipse-perimeter.html. [Accessed: 13-Oct-2019].

# Code - Main

**package** Project1;  
**import** javafx.application.Application;  
**import** javafx.scene.Group;  
**import** javafx.scene.Scene;  
**import** javafx.scene.canvas.Canvas;  
**import** javafx.scene.canvas.GraphicsContext;  
**import** javafx.stage.Stage;  
**import static** java.lang.Math.*sqrt*;  
  
  
**public class** Main **extends** Application{  
 **public static void** main(String[] args){  
 *launch*(args);  
 }  
 **public static final int *w*** = 700, ***h*** = 500;  
 @Override  
 **public void** start(Stage stage) {  
 Group root = **new** Group();  
 Canvas = **new** Canvas(***w***, ***h***);  
 GraphicsContext gc = canvas.getGraphicsContext2D();  
 *//declaration of 1st set of objects* MyShape shape = **new** MyShape(***w***/2, ***h***/2);  
 MyLine line = **new** MyLine(***w***, ***h***);  
 MyRectangle R1 = **new** MyRectangle(***w***/2,***h***/2);  
 MyOval O1 = **new** MyOval(***w***/2, ***h***/2);  
 Double L, M;  
 **int** l, m;  
 *//draw objects* l = ***w***/2;  
 m = ***h***/2;  
 R1.setX(***w***/2);  
 R1.setY(***h***/2);  
 shape.*setColor*(gc, **"blue"**);  
 R1.draw(gc, **false**);  
 shape.*setColor*(gc, **"green"**);  
 O1.draw(gc, **false**);  
 *//1 size in* L = l/*sqrt*(2);  
 M = m/*sqrt*(2);  
 l = L.intValue();  
 m = M.intValue();  
 *// declare 2nd set of objects* MyRectangle R2 = **new** MyRectangle(l,m);  
 MyOval O2 = **new** MyOval(l,m);  
 *// draw objects* R2.setX(***w***/2);  
 R2.setY(***h***/2);  
 shape.*setColor*(gc, **"red"**);  
 R2.draw(gc, **false**);  
 shape.*setColor*(gc, **"yellow"**);  
 O2.draw(gc, **false**);  
 *// 2nd size in* L = l/*sqrt*(2);  
 M = m/*sqrt*(2);  
 l = L.intValue();  
 m = M.intValue();  
 *// declare 2nd set of objects* MyRectangle R3 = **new** MyRectangle(l,m);  
 MyOval O3 = **new** MyOval(l,m);  
 *// draw objects* R3.setX(***w***/2);  
 R3.setY(***h***/2);  
 shape.*setColor*(gc, **"purple"**);  
 R3.draw(gc, **false**);  
 shape.*setColor*(gc, **"brown"**);  
 O3.draw(gc, **false**);  
 shape.*setColor*(gc,**"black"**);  
 line.draw(gc, **false**);  
 *//outlines that overlap line* R1.setX(***w***/2);  
 R1.setY(***h***/2);  
 shape.*setColor*(gc, **"blue"**);  
 R1.draw(gc, **true**);  
 shape.*setColor*(gc, **"green"**);  
 O1.draw(gc, **true**);  
 shape.*setColor*(gc, **"red"**);  
 R2.setX(***w***/2);  
 R2.setY(***h***/2);  
 R2.draw(gc, **true**);  
 shape.*setColor*(gc, **"yellow"**);  
 O2.draw(gc, **true**);  
 shape.*setColor*(gc, **"purple"**);  
 R3.setX(***w***/2);  
 R3.setY(***h***/2);  
 R3.draw(gc, **true**);  
 shape.*setColor*(gc, **"brown"**);  
 O3.draw(gc, **true**);  
 *//display canvas / scene / stage* root.getChildren().add(canvas); *//adds the canvas to the root group* stage.setScene(**new** Scene(root));*//sets the scene on the stage that is using the root with the canvas elements* stage.show();*// tells java to show the stage in the application with all of the things that we put in it* }  
 }

# Code – MyShape

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
**import** javafx.scene.paint.Paint;  
**import** java.lang.Object;  
  
**public class** MyShape **extends** Object{  
 **private int w**, **h**;  
 **public** MyShape(**int** w, **int** h){  
 **this**.**w** = w;  
 **this**.**h** = h;  
 }  
 **protected static int** *x*, *y*;  
 **public void** setX(**int** X){  
 *x* = X - **w**/2;  
 }  
 **public void** setY(**int** Y){  
 *y* = Y - **h**/2;  
 }  
 **public static void** setColor(GraphicsContext gc, String s ){  
 gc.setFill(MyColor.*getColor*(s));  
 gc.setStroke(MyColor.*getColor*(s));  
 }  
 **public static int** getX(){  
 **return** *x*;  
 }  
 **public static int** getY(){  
 **return** *y*;  
 }  
 **public static** Paint getColor(GraphicsContext gc){  
 **return** gc.getFill();  
 }  
 **public** String toString(){  
 String s = **"no object specified"**;  
 **return** s;  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o){  
 gc.fill();  
 }  
}

# Code – MyLine

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
**import static** java.lang.Math.*atan*;  
**import static** java.lang.Math.*sqrt*;  
**import static** java.lang.Math.*pow*;  
  
**public class** MyLine **extends** Project1.MyShape {  
 **private double x1**, **x2**, **y1**, **y2**;  
 *// constructor method* **public** MyLine(**int** w, **int** h){  
 **super**(w, h);  
 **this**.**x1** = 0;  
 **this**.**x2** = w;  
 **this**.**y1** = 0;  
 **this**.**y2** = h;  
 }  
 @Override  
 **public** String toString(){  
 **double** length = *sqrt*(*pow*((**x2**-**x1**), 2) + *pow*((**y2** - **y1**), 2));  
 **double** angle = *atan*((**y2** - **y1**)/(**x2**-**x1**));  
 **return "length: "** + length + **" angle: "** + angle;  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o){  
 gc.strokeLine(**x1**,**y1**,**x2**,**y2**);  
 }  
}

# Code – MyRectangle

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
  
**public class** MyRectangle **extends** MyShape {  
 **private int w**,**h**;  
 **public** MyRectangle(**int** w, **int** h){  
 **super**(w, h);  
 **this**.**w** = w;  
 **this**.**h** = h;  
 }  
 @Override  
 **public** String toString(){  
 **int** P, A;  
 P = **w** + **w** + **h** + **h**;  
 A = **w**\***h**;  
 **return "Width: "** + **w** + **" Height: "** + **h** + **" Perimeter: "** + P + **" Area:"** + A;  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o){  
 **if**(o){  
 gc.strokeRect(*x*,*y*,**w**,**h**);  
 }  
 **else** {  
 gc.fillRect(*x*, *y*, **w**, **h**);  
 }  
 }  
}

# Code – MyOval

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
**import static** java.lang.Math.*sqrt*;  
**import static** java.lang.Math.***PI***;  
  
**public class** MyOval **extends** MyShape {  
 **private double a**,**b**;  
 **private int w**, **h**;  
 **public** MyOval(**int** w, **int** h){  
 **super**(w, h);  
 **this**.**a** = w;  
 **this**.**b** = h;  
 **this**.**w** = w;  
 **this**.**h** = h;  
 }  
 **public** Double getPerimeter(){  
 **return *PI***\*((3\*(**a**+**b**)/2)-(*sqrt*(((3\***a**)+**b**)\*(**a**+(3\***b**))/4)));  
 }  
 **public** Double getArea(){  
 **return a**\***b**\****PI***/4;  
 }  
 @Override  
 **public** String toString(){  
 **return "Vertical Axes: "** + **b**/2 + **" Horizontal Axes: "** + **a**/2 + **" Perimeter: "** + **this**.getPerimeter() + **" Area:"** + **this**.getArea();  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o){  
 **if**(o){  
 gc.strokeOval(*x*,*y*,**w**,**h**);  
 }  
 **else** {  
 gc.fillOval(*x*, *y*, **w**, **h**);  
 }  
 }  
}

# Code – MyColor

**package** Project1;  
**import** javafx.scene.paint.Color;  
  
**public enum** MyColor{  
 ***BLUE***(0, 0 , 255), ***GREEN***(0, 255, 0), ***RED***(255, 0, 0), ***YELLLOW***(248, 222, 126),  
 ***BROWN***(102, 51, 0), ***PUPLRE***(222, 222, 225), ***BLACK***(0,0,0);  
 **private int red**, **green**, **blue**;  
 **private** MyColor(**int** r, **int** g, **int** b) {  
 **red** = r;  
 **green** = g;  
 **blue** = b;  
 }  
 **public** String hex(){  
 String a, b, c;  
 a = Integer.*toHexString*(**red**);  
 b = Integer.*toHexString*(**green**);  
 c = Integer.*toHexString*(**blue**);  
 **return "0x"**+a+a+b+b+c+c;  
 }  
 **public** Color paint() {  
 **return** Color.*rgb*(**red**, **green**, **blue**);  
 }  
 **public static** Color getColor(String S){  
 **int** r = 0;  
 **int** g = 0;  
 **int** b = 0;  
 **switch** (S) {  
 **case "blue"**:  
 r = 0;  
 g = 0;  
 b = 255;  
 **break**;  
 **case "green"**:  
 r = 0;  
 g = 255;  
 b = 0;  
 **break**;  
 **case "red"**:  
 r = 255;  
 g = 0;  
 b = 0;  
 **break**;  
 **case "yellow"**:  
 r = 248;  
 g = 222;  
 b = 126;  
 **break**;  
 **case "brown"**:  
 r = 102;  
 g = 51;  
 b = 0;  
 **break**;  
 **case "purple"**:  
 r = 222;  
 g = 222;  
 b = 255;  
 **break**;  
 **case "black"**:  
 r = 0;  
 b = 0;  
 g = 0;  
 **break**;  
 }  
 **return** Color.*rgb*(r, g ,b);  
 }  
 **public** String define(){  
 **return "Red = "** + **red** + **" Green = "** + **green** + **" Blue = "** + **blue**;  
 }  
 **public static** Color mix(String color1, String color2){  
 **int** r = 0;  
 **int** g = 0;  
 **int** b = 0;  
 **switch** (color1) {  
 **case "blue"**:  
 r = 0;  
 g = 0;  
 b = 255;  
 **break**;  
 **case "green"**:  
 r = 0;  
 g = 255;  
 b = 0;  
 **break**;  
 **case "red"**:  
 r = 255;  
 g = 0;  
 b = 0;  
 **break**;  
 **case "yellow"**:  
 r = 248;  
 g = 222;  
 b = 126;  
 **break**;  
 **case "brown"**:  
 r = 102;  
 g = 51;  
 b = 0;  
 **break**;  
 **case "purple"**:  
 r = 222;  
 g = 222;  
 b = 255;  
 **break**;  
 **case "black"**:  
 r = 0;  
 b = 0;  
 g = 0;  
 **break**;  
 }  
 **switch** (color2) {  
 **case "blue"**:  
 r += 0;  
 g += 0;  
 b += 255;  
 **break**;  
 **case "green"**:  
 r += 0;  
 g += 255;  
 b += 0;  
 **break**;  
 **case "red"**:  
 r += 255;  
 g += 0;  
 b += 0;  
 **break**;  
 **case "yellow"**:  
 r += 248;  
 g += 222;  
 b += 126;  
 **break**;  
 **case "brown"**:  
 r += 102;  
 g += 51;  
 b += 0;  
 **break**;  
 **case "purple"**:  
 r += 222;  
 g += 222;  
 b += 255;  
 **break**;  
 **case "black"**:  
 r = 0;  
 b = 0;  
 g = 0;  
 **break**;  
 }  
 **if**(r > 255){  
 r = 255;  
 }  
 **if**(g > 255){  
 g =255;  
 }  
 **if**(b > 255){  
 b = 255;  
 }  
 **return** Color.*rgb*(r, g ,b);  
 }  
}

1. https://www.mathsisfun.com/geometry/ellipse-perimeter.html [↑](#footnote-ref-1)