The City College of New York

CSC 221 – P || Professor Hesham Auda

Exercise 2

Student: Binyamin Radensky

Contents

[Introduction 2](#_Toc23419957)

[Class Main 3](#_Toc23419958)

[Class MyShape 6](#_Toc23419959)

[Class MyCircle 9](#_Toc23419960)

[Class MyPolygon 11](#_Toc23419961)

[Interface MyPositionInterface 18](#_Toc23419962)

[Interface MyShapePositionInterface 19](#_Toc23419963)

[Class MyLine 20](#_Toc23419964)

[Class MyRectangle 21](#_Toc23419965)

[Class MyOval 23](#_Toc23419966)

[Enum MyColor 25](#_Toc23419967)

[Conclusion 28](#_Toc23419968)

[Works Cited 29](#_Toc23419969)

[Code 30](#_Toc23419970)

[Main 30](#_Toc23419971)

[MyShape 33](#_Toc23419972)

[MyCircle 35](#_Toc23419973)

[MyPolygon 36](#_Toc23419974)

[MyPositionInterface 39](#_Toc23419975)

[MyShapePositionInterface 40](#_Toc23419976)

[MyLine 41](#_Toc23419977)

[MyRectangle 42](#_Toc23419978)

[MyOval 43](#_Toc23419979)

[MyColor 44](#_Toc23419980)

# Introduction

In this project we used interfaces and abstract classes to implement more functionality to our MyShape class hierarchy. Additionally, we added to more classes to the hierarchy, one that creates circles and is an extension of the MyOval class, and another class that creates polygon and is a MyShape. The challenge with creating these classes was that they have other ways of modifying their sizes, so in order to keep all the inherited functions consistent we had to make sure that all our variables were update. After creating the new classes, we made the two interfaces which are a convenient way of forcing implementation into the sub classes of the hierarchy. The other good part of an interface is that we can implement it in different ways for each class, since each one has its own set of variables and methods of implementation. We also made our superclass, MyShape, into an abstract class. This means that the class can no longer be instantiated but can now have abstract methods that can be implemented in its subclasses. This is another great hierarchal tool in Java. It allows us to implement the methods in the superclass that is applicable to all the subclasses without modification and then create abstract methods for the rest of the methods that need to be custom created for each subclass. This was a very informative project that had an intuitive flow for the learning process. By first creating all our new regular class and making MyShape abstract we strengthened the base of knowledge that we gained in project one, then we moved on to doing new things such as dealing with interface classes.

# Class Main

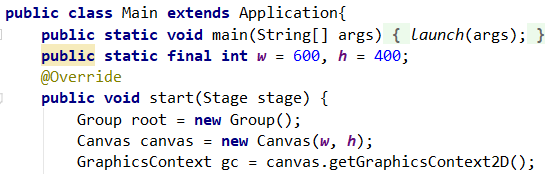
The first thing we did was modify our main method from the last project. It begins by extending the application class, launching our start method, and creating variables for the width and height of our window.

Figure : main header and start method

Our start method is completely the same as the previous project. We create a group called root which will hold all of the canvas elements. Then we create a new canvas with the width and height we specified. Then we instantiate our two dimensional graphics context object which we will use to draw all of our shapes.

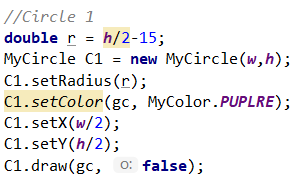
The first shape that we call in order to make the desired geometric configuration is the MyCircle object we created. First, we create a variable which will serve as the radius of our circle. Then we instantiated our circle object with the width and height of the window. As we will see in the MyCircle section of the report, the width and height are used for defaults (radius = (w+h)/4) however they are there as a placeholder and don’t really need to be used. Instead we can see the radius as is shown on the line below. After this we set the color using the MyColor enum type which we created in the previous project. We then set the center of the circle to be in the center of the canvas. Lastly, we draw the circle onto the canvas, and we set the outline option to false so that we get a filled in circle on our canvas.

Figure : Circle 1 instantiation and method calls

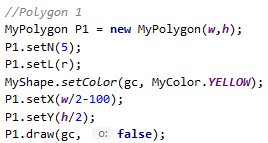
Next, we instantiated one of our MyPolygon objects to create the geometric configuration required for this project. The width and height are again, not required; however, they will set values if they are elected to be used. We set the ‘N’ which is the number of sides our polygon shape will have, and we set the ‘L’ which is the length of each of the equal sides. We set the color to a different color than our last shape and we set the x and y coordinates of where we want the center of our shape to be.

Figure : Polygon 1 instantiation and method calls

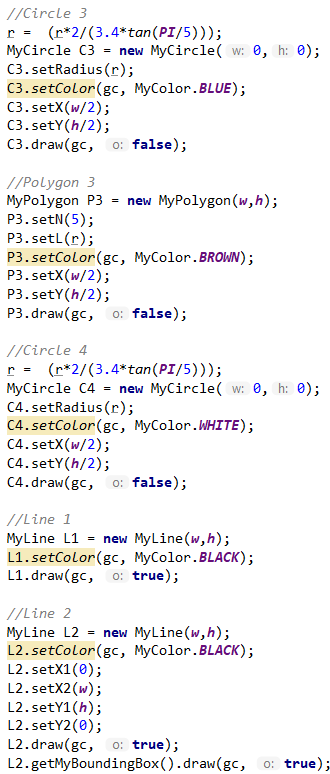
The rest middle of our start method is a repetitive block of code where we instantiate all the objects necessary for our geometric configuration. For the radius we calculated the largest circle that could possible fit into the polygon which is the value of ‘r’. [1] We also instantiated our MyLine objects which cut across our geometric configuration to make an ’X’. For this assignment we added new methods to the MyLine class so that each of the x1, x2, y1, and y2 coordinates could be set. We also used, at the very last line, a bounding box of the MyLine object which created the black outline around our geometric configuration.

Figure : Instantiation of various objects to create geometric configuration and lines

Below are all the methods we imported for our main class. As well as the command declaring the main class part of the package project one, even though we are in project 2.

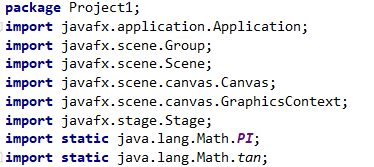


Figure : imported methods for main class

Lastly, we have our code which adds the canvas to the root group, and then sets the scene with the root and then shows the stage.



Figure : final snippet of code from main class start method

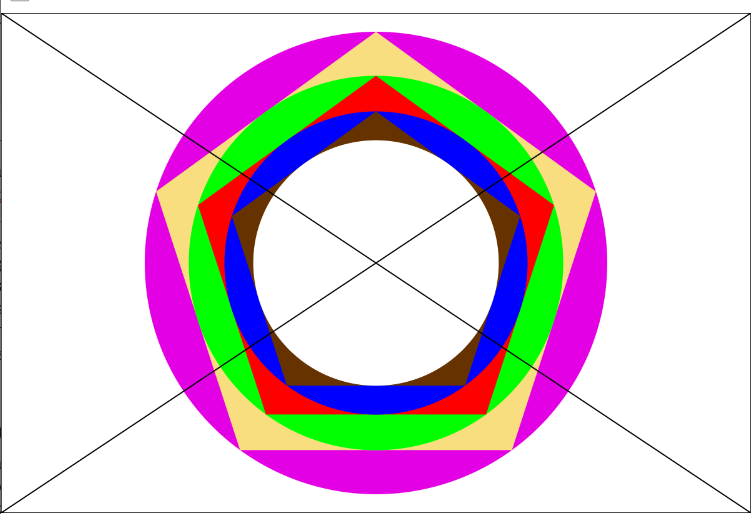


Figure : Output of our Program

# Class MyShape

Our MyShape class is now an abstract class that extends object and implements the MyShapePositionInterface.



Figure : header for MyShape

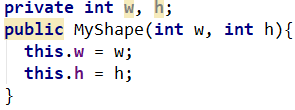
The constructor is the same as last time where it takes the width and height of the object and makes them into local instance variables.

Figure : constructor, myshape

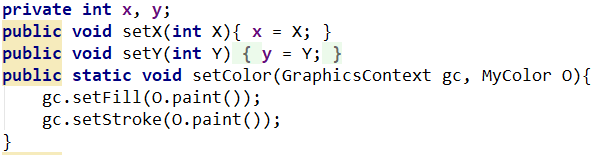
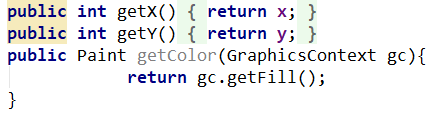
We then have the three methods that set the middle point of the object (x, y) and the color of the object. We improved these from last project since now you can pass the MyColor enum directly into the set color function, additionally the x,y point is now calculated in each individual subclass instead of a general definition since that way as we begin to get different shapes, such as MyCircle, which has a different variable for calculating the middle of the object in relation to the x,y point specified. We also have the corresponding three methods that will retrieve what we set, which we will use in our subclasses.

Figure : get methods myshape

Figure : set methods myshape

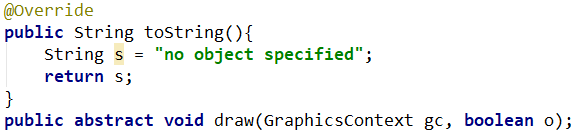
We also have the same to string method as before. Now we have a draw method that is abstract, so it is forced to be overridden in every subclass.

Figure : toString and draw myshape

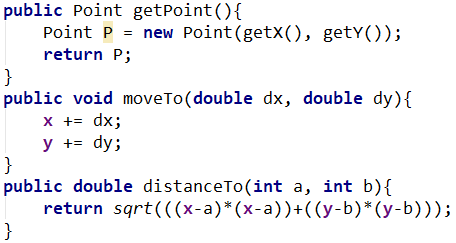
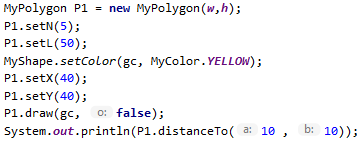
Next, we have the implementation of the MyPositionInterface methods. Since they are not subclass specific, we were able to implement them in the MyShape superclass. The distanceTo method uses the Pythagorean theorem to calculate distance between two points. An example and verification of the distanceTo method can be seen below: 

Figure : Implementation of methods from MyPositionInterface in myshape

Figure : Input for distanceTo method

Figure : Google Chrome calculator results for the same operation as distanceTo implementation showing the numbers are equal.



Figure : Output for distanceTo method

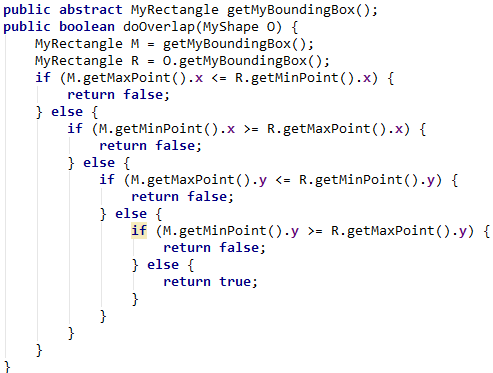
 Lastly, we finish the class with our implementation of the MyShapePositionInterface implementation where the getMyBoundingBox becomes an abstract method so that it can be implement uniquely for every subclass and doOverlap is implemented in the MyShape superclass itself. For doOverlap we took every case where the shapes would not overlap and if that case was true then it would return false, otherwise the method will return true. So, if the shapes are out of bounds of each other in the x or y direction then it would return false. To be able to get the coordinates of the shape objects we used the opposite corner coordinates of the getMyBoundingBox by creating two new methods in our MyRectangle class. Once we have the two corner coordinates, we then know the maximum and minimum values of the x and y directions of the bounding box of the shape. The only issue is that since this method using the bounding box, it is possible, especially with a circle and a polygon that the corners of the Bounding boxes will be overlapping and therefore the method will return true (only in our implementation) but they will not actually be overlapping.

Figure : Implementation of MyShapePositionInterface, myshape

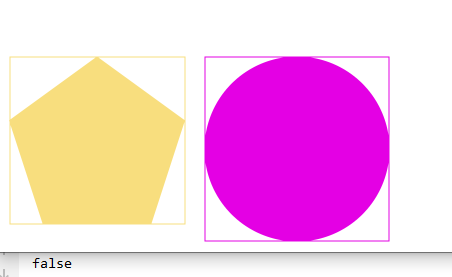


Figure : Showing the doOverlap method returning false when the shapes are not overlapping

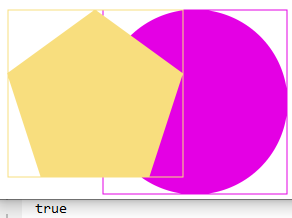


Figure : Showing the doOverlap method returning true when the shapes are overlapping

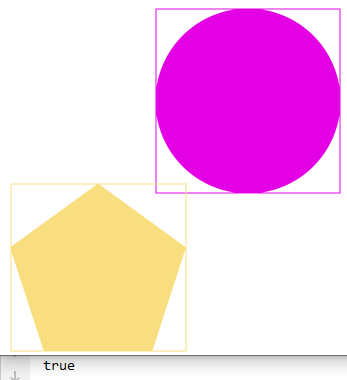


Figure : Illustrating the issue with doOverlap since it is using the bounding box to estimate whether not they overlap, it is possible for the method to return the wrong answer

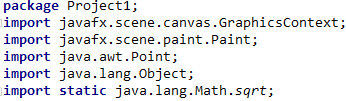
 To the right are the methods we imported for the MyShape class. Also, on top is the declaration of the package that this class is part of.

Figure : imported methods, myshape

# Class MyCircle

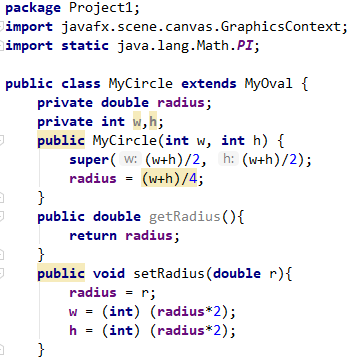
 The MyCircle class is an extension of the MyOval class that we made in the previous project. This makes sense since they are very similar, and we could potentially inherit and use the methods from the MyOval class. The difference between the two is that the MyCircle class uses the radius variable as the way of manipulating the size of the object while the MyOval class uses the w, h variables that are passed into it when it is instantiated. To keep consistency and allow for extended functionality, we have implemented the MyCircle class in such a way that you can use both the MyOval methods with a MyCircle object and use the methods that we have written into the MyCircle class. Both will work since the w, h variables will continue to be updated as twice the radius so that they can be used and are not irrelevant if someone were to elect to use them. To start the w, h variables will define the radius in the constructor so that once the object is instantiated it can be used right away. Also included in our snippet of code are the get and set radius methods that will set the radius of the circle. As mentioned, whenever the radius is set, the width and height variables are also set to allow them to be used as will be seen bellow.

Figure : Implementation of the first half of the MyCircle class

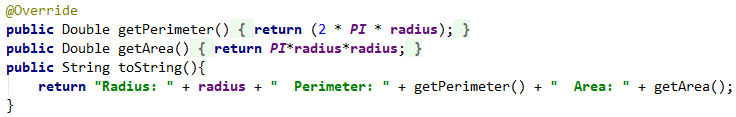
Our next methods are all used to create the toString method for the MyCircle class. In order to keep the toString method clean looking we mimicked the way that we created the method in the MyOval class from the previous project and we even overrode those methods. Since the MyCircle class uses the set radius method to set its radius, and this is not done in the constructor, we are unable to properly inherit those methods without overriding them since we would need to call super() which can only be done in the constructor. If use you sue the default radius which is set when the width and height of the circle are passed into the constructor, then we can in fact use those methods. 

Figure : toString method with other methods that it calls. MyCircle

Figure :Output of toString method for MyCircle

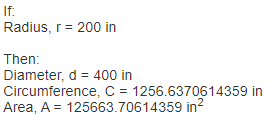


Figure : Output of https://www.calculatorsoup.com/calculators/geometry-plane/circle.php with specified radius. Confirming results of output.

Below are the outputs of the overriding methods in MyCircle and below that, the outputs of the inherited methods from the MyOval class when called on the same circle. In order to get the “radius” output like in the tope figure we can just override the toString method and inherit the other two as seen in the last figure. We generated these by commenting out the overriding methods in the MyCircle class and calling “System.out.println(C1.toString());” in the start method of the main class.



Figure : output when the methods in MyCircle override the methods from MyOval.



Figure : Output when all three of the toString method's methods inherited from MyOval



Figure : Overridden toString method using inherited get perimeter and get area methods from MyOval.

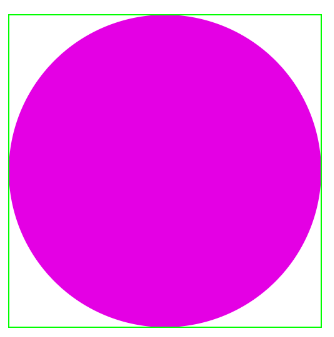
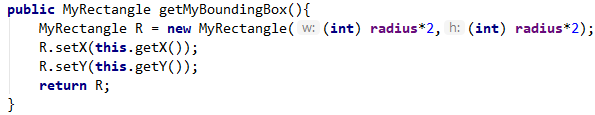
 Next, we overrode the getMyBoundingBox method which we inherited from the interface MyShapePositionInterface. This method returns the rectangle that would exactly fit the circle, called the bounding rectangle, by setting the width and height to twice the radius. We also set the middle point of the bounding box to be the same as the middle point of the circle. The bounding boxes are MyRectangle objects so that would make custom methods for it and call it easily since it is already part of the package.

Figure : code for getMyBoundingBox method

Figure : MyCircle object (purple) with bounding box drawn around it (green).

Lastly, we overrode the draw method for the MyCircle object, and instead of using width and height to specify the size of the object and where the corner of the object is in relation to the point x,y (which we retrieve using the inherited methods getX() and getY()) we used the radius.

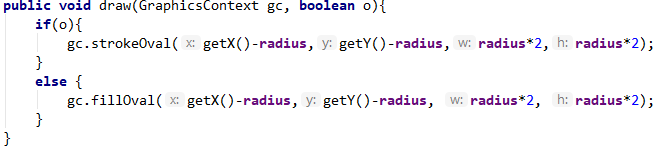


Figure : draw method for the MyCircle class

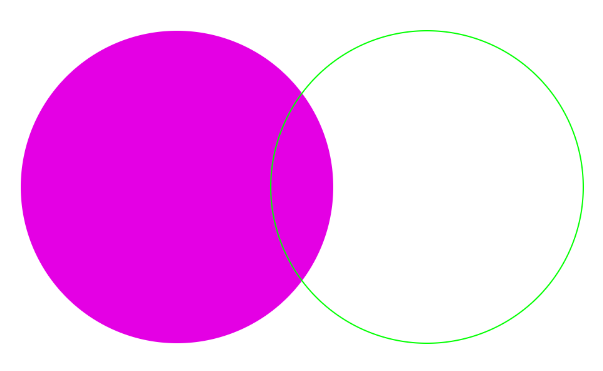


Figure : Output of the draw method when outline is false (left) and when outline is true (right)

# Class MyPolygon

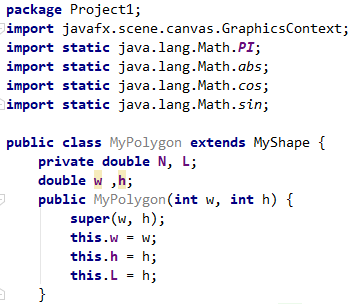
 The MyPolygon class inherits from our MyShape superclass and uses the parameters w and h as initial values in its constructor. It is part of the package project1 and in order to calculate the location of the points where the vertices of the polygon are as well as the width and height of the bounding box around the polygon, we used several methods from the Math class in the java.lang library.

Figure : Snippet of first part of the MyPolygon class

We have the two methods to set the number of sides in our polygon as well as the method to set the length of each of the equal side in our polygon.

Figure : setN and setL methods.

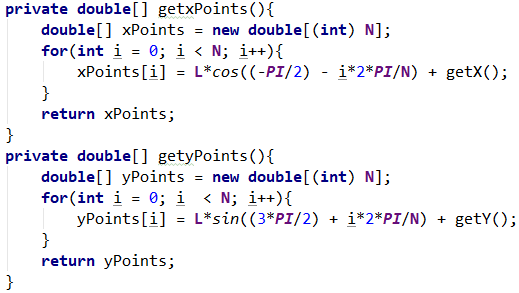
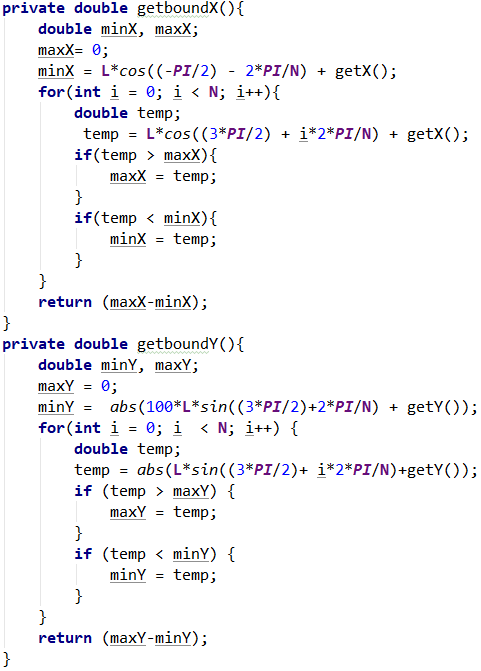
 Next, we have our two methods that will generate the x and y values of the points where the vertices of the polygon will be located. When we originally create this method, we used the formula without the -PI/2, however in order to rotate it so that the point of the polygon is oriented to the top we shifted it by PI/2. The equation is iterative which was perfect for using a for loop with, and then we store the results in an x and y double type array which the function returns when called.

Figure : methods to get arrays filled with the x and y coordinates of the vertices of the polygon

After, we used the same formulas to find the maximum and minimum coordinates of the vertices in the polygon so that we could get the width and height of the polygon to make a bounding box. It would go through the value and if the max was less than the current value it would store the current value, if the min was greater than the current value it would store the current value. Although we could have had the for loop just go through the array that already has the values stored, because the center of the polygon is different than the center of the square we wanted to create a slightly different equation to try and find a more accurate bounding box. In the end we just stuck with what we had since in order to really get the bounding box perfectly around the polygon we would have to change the x and y value of the bounding box, however the amount that the bounding box is insignificant enough that it was not worth messing with it since when we have more than 5 sides there is no issue.

Figure : method for getting the width and height of the area of the polygon to be used for the bounding box

The toString method outputs the side length, the interior angle in degrees and the perimeter.

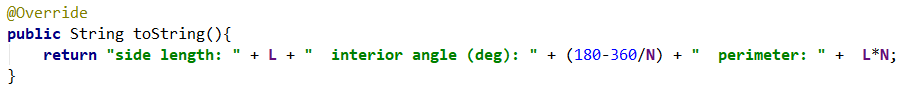


Figure : toString method for MyPolygon



Figure : output of toString

For the getMyBoundingBox method we used the functions above to get the maximum width and length of the polygon. We the create a new MyRectangle object with that width and height. After that we set the middle of that rectangle to have the same canter coordinates as our polygon.

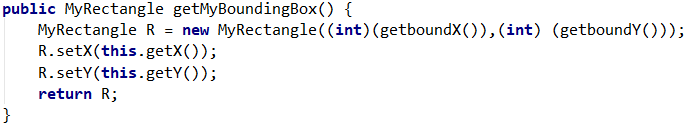
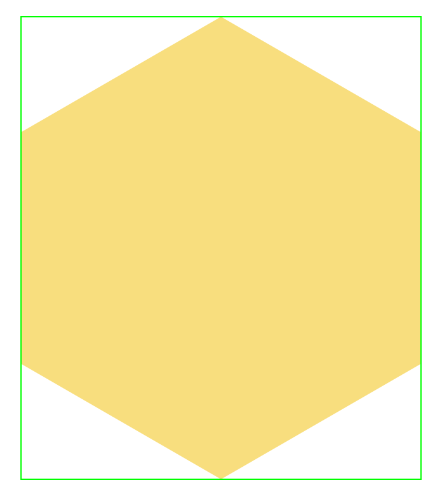
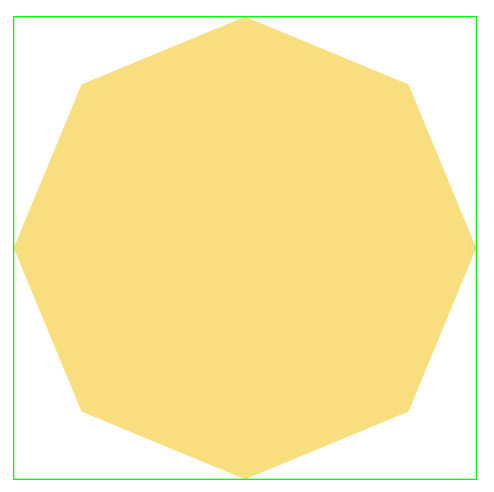


Figure : Bad code for getMyBoundingBox method in MyPolygon, not used in the actual project

As mentioned above the getMyBoundingBox method for MyPolygon is not perfect when it comes to odd side numbered polygons. Bellow you can see examples of other polygons where the method works and some where it does not.



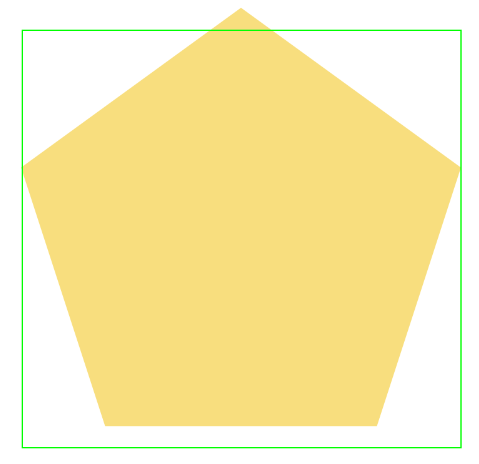
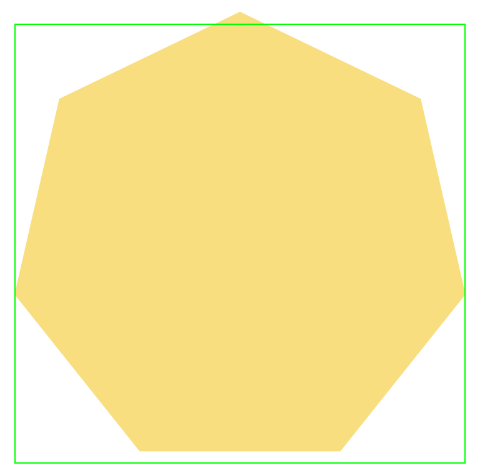


Figure : 6-sided polygon perfectly fits into bounding box

Figure : 8-sided polygon perfectly fits into bounding box

Figure : We can see that the as we add more sides the gap gets smaller and smaller (7 sides)

Figure : 5-sided polygon is slightly outside of the bounding box

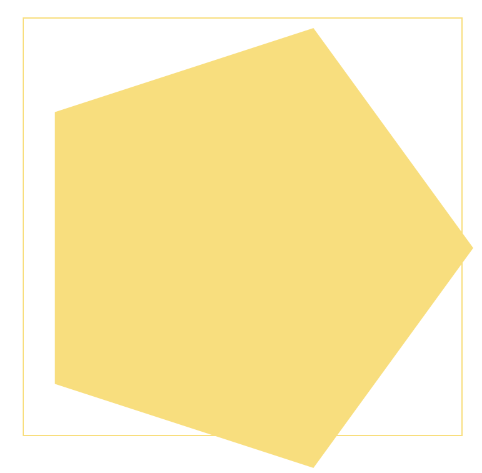
In order to fix this error, we had to account for the fact that we had rotation of the shape so that the point would be on top. We used the equation L\*sin(i\*2\*PI/N) where ‘i’ goes from zero N, the number of sides. We did this by rotating the polygon by 270 degrees. This also meant that we were going to have a slight offset from where the vertices of the polygon were going to be resting in the y direction as can be seen from above. In order to correct the change of rotation we had to multiply the offset by the offset caused by the fact that not all point have the same with and height. When the shape is even this isn’t an issue because the longest distance is the distance between the top and bottom vertices. This is never an issue in the x direction because the x direction always contains that longest distance. As we can see in the image of the 5 side polygon above, there is only a vertex at the top that protrudes from the polygon, however the two vertices at the bottom are parallel and are not maximizing the height of the polygon as they would if they were rotated. So to correct al of this we subtracted the radius of the inner circle (called the apothem) by the radius of the outer circle and then we multiplied that by sqrt(N)/N to correct for the rotation (since that is the y value of the offset caused by the rotation). We also put in an if statement that will detect where there are an even number of sides or not. We can see that the even number of sides has not issues since the widest part of the polygon where the vertices are sticking out the most exist in the top and bottom of the shape. Below is the corrected version of the getMyBoundingBox method.

Figure : The way the polygon prints with any correction

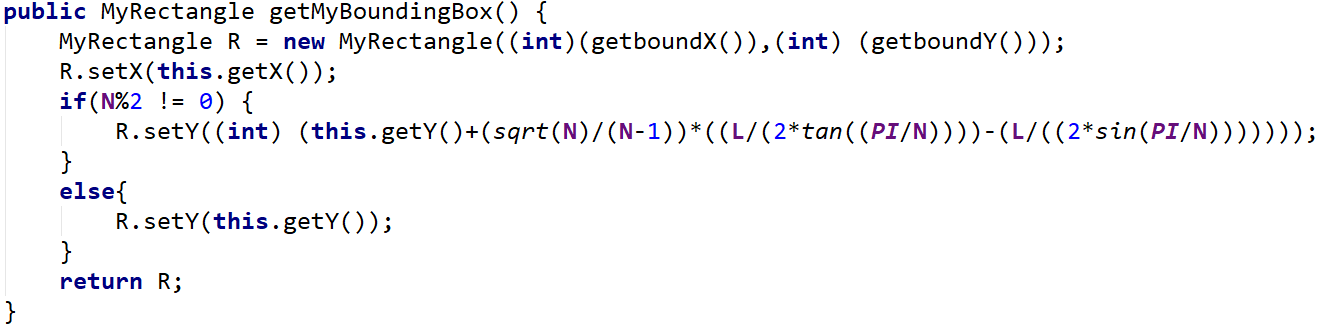
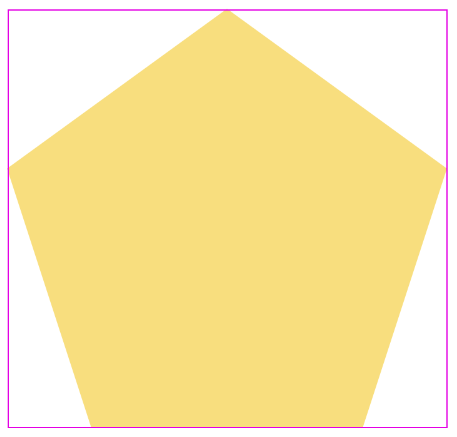


Figure : code for getMyBoundingBox that corrects the location when there is an odd number of sides for the polygon



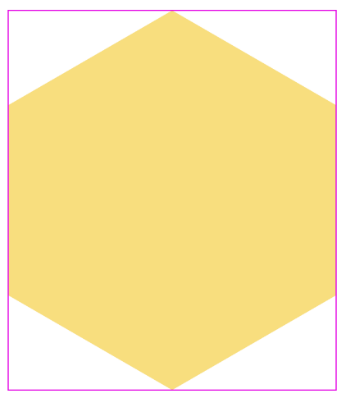
Figure : The same shape object but with the 270-degree correction

Figure : 6-sided polygon with correct bounding box, consistent as before because of the if statement in the method

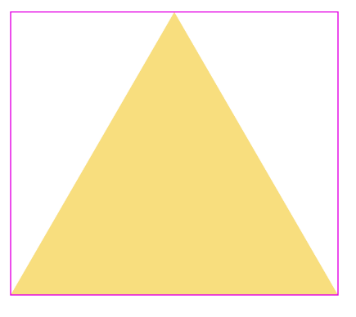
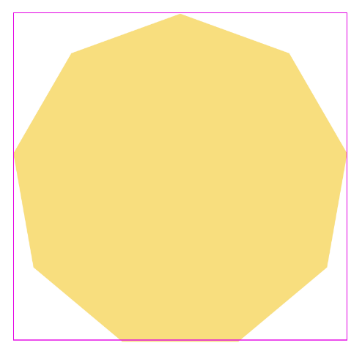


Figure : 3-sided polygon (triangle) with corrected bounding box

Figure : 9-sided polygon with corrected bounding box



Lastly, we have the draw method of our MyPolygon class. This method is unique that you do not give it the coordinates of the middle of that shape and its height and width. You don’t even give it a radius or the length of a side. Rather, we pass in an array with the x and y coordinates of the vertices of the polygon and the number of sides the polygon has. As is standard we also gave the capability of either fill in in the polygon or just having an outline of the polygon drawn on the canvas.

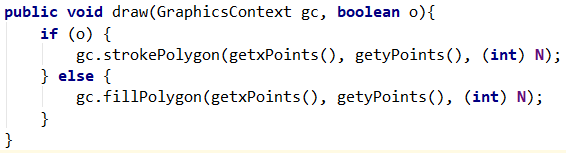


Figure : draw method for the MyPolygon class

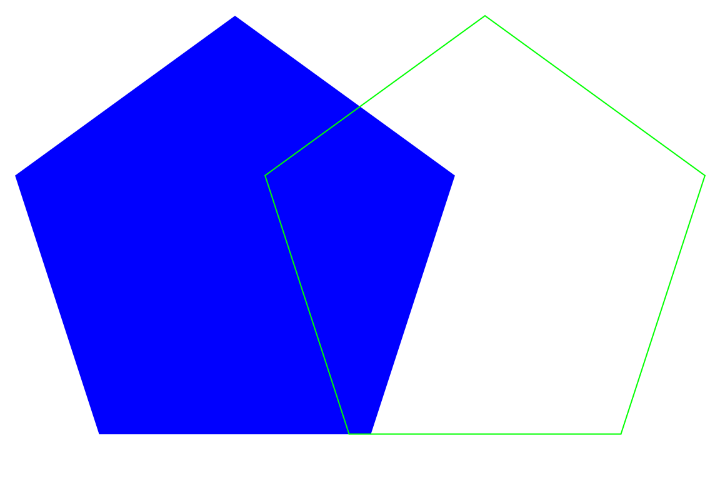


Figure : Output of the draw method when outline is false (left) and when outline is true (right)

# Interface MyPositionInterface

This is the first interface that we have created. Due to the way an interface is structured we do not include a body of the method, and all that we include is the header of each method that will need to be implemented by and class that inherits the interface, as well as a brief description of how the method should be implemented. These kinds of files are reminiscent of header files in C++.

Figure : package declaration, imports and header for the interface

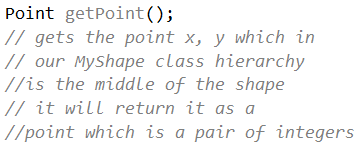
 For our first method we made it so that the return type is a point which is a pair of integers that are considered one object. A description of the method can be seen below the header for the method so that if someone else were to use this interface they would know how to implement it properly. This method returns the coordinates x, y as a point object. In our MyShape class that is the coordinates for the center of the object.

Figure : getPoint method in the MyPositonInterface interface

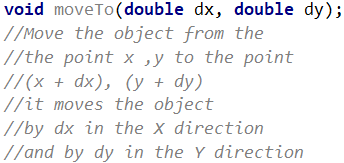
 For our moveTo method we have two parameters which will be used to move the object in those directions by the amount passed into those parameters. The first parameter is the amount which we want to move the object in the x direction (where 0,0 is the top left corner of the axes), and the second is the amount which we want to move the object in the y direction.

Figure : moveTo method in interface

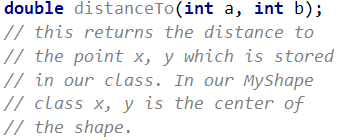
 Our last method in the interface is the distanceTo method which has the parameters a and b which are a pair of coordinates; the method should return a double value with the distance from those coordinates to the x, y coordinates stored in our class. In our MyShape class x, y are the coordinates for the center of the shape, so this method would return a single value stating the distance between (a, b) the specified point and (x, y) the middle of our shape.

Figure : distanceTo method in interface

# Interface MyShapePositionInterface

The MyShapePositionInterface interface is an interface that inherits from the MyPositionInterface and is made specifically for our MyShape class. When it inherits from the MyPositionInterface that means that all the methods from that interface is inherited which means that when this interface is implemented, all the unimplemented methods from that interface must also be implemented.



Figure : Header and package declaration for the interface

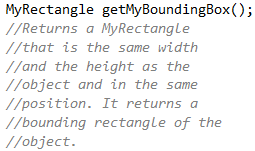
 The method getMyBoundingBox returns the MyRectangle object that would bound the MyShape object it is being called on, meaning that would have the same width, height and position as that object. As can be seen above this method creates a rectangle that completely covers every pint of that object, so that if it was drawn the object would fit perfectly within the rectangle. This will be extremely useful for our next method in this interface.

Figure : getMyBoundingBox method for the interface

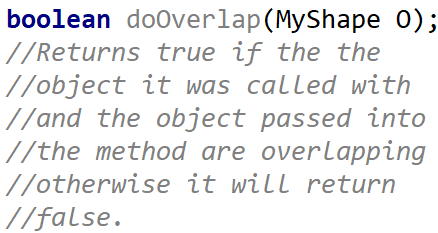
 The method doOverlap returns true if two of our shape objects overlap. The way that we implemented this in our project was by using the bounding boxes and checking to see if the bound boxes were out of range of each other in the x and y directions. The way that would use this method is by calling it using a shape and then we pass in a different shape object and our method will return true if they are overlapping. As mentioned above, since this method is using the bounding box, it is possible, especially with a circle and a polygon that the corners of the Bounding boxes will be overlapping and therefore the method will return true (only in our implementation) but they will not actually be overlapping as illustrated above.

Figure : doOverlap method for the interface

# Class MyLine

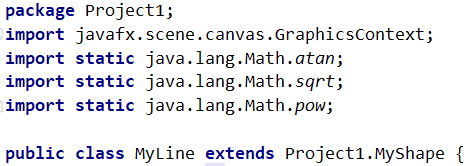
 Class MyLine extends the class MyShape which means that it inherits all the methods of MyShape and includes it functionality. Methods that MyLine needs that it does not have such as the mathematical functions and graphics context for draw(). We also declare that this is part of Project1.

Figure : beginning of MyLine class

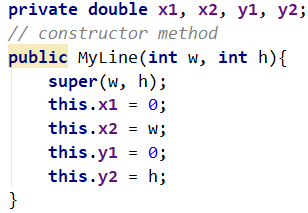
 The constructor initiates all the variables and sets the 4 coordinates of the line. For our implementation we set the values in the constructor itself. We also insert super which takes the width and height that was passed into the constructor of our subclass and uses it on the constructor of the superclass, 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. We can change the values of x1, y1, x2, y2 as seen bellow.

Figure : constructor for MyLine class

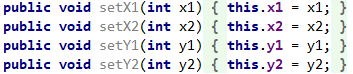
 The set methods allow us to set the two points which make up the line. Each method corresponds to one of the two coordinates in each point. So, for our project we set them to (0, h) and (w, 0) so that we could make the x across our geometric configuration as was asked of us.

Figure : Group of set function in the MyLine class

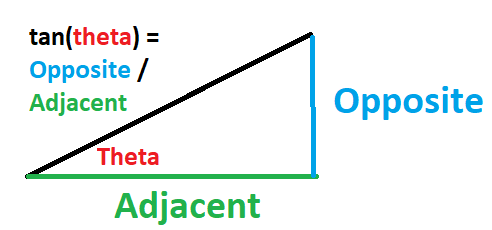
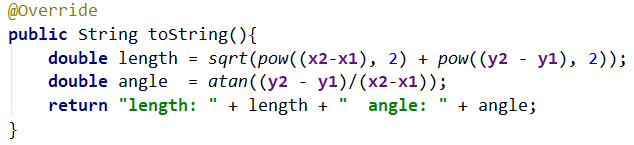
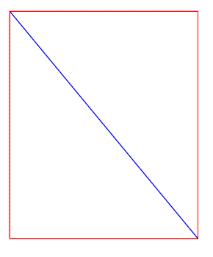
 The toString method overrides MyShape toString and returns a string containing the description of the shape, in our case the length and angle of our line. To find the length we use Pythagorean’s theorem which is A­­­2 + B­2 = C2 where x2 - x1 is the length of the x component and y2 – y1 is the length 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.

Figure : toString method for MyLine class



Our newest method for this class is the overriding getMyBoundingBox method for the MyLine class. This method returns the MyRectangle object that occupies the same width, height, and location as the instantiated line object it is called on.

Figure : MyLine object (blue) with bounding box (red) around it

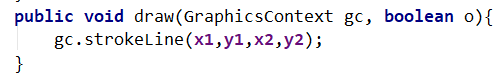
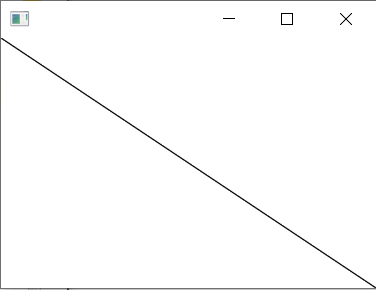
 Lastly, the overridden draw method which has the parameters of the graphics context and boolean o will draw the stroke line onto the canvas based off the values of the variables set in the constructor. For this class boolean o does not serve a purpose, however since it is overriding the inherited draw 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.

Figure : Line object on canvas

Figure : Draw method for MyLine class

# Class MyRectangle

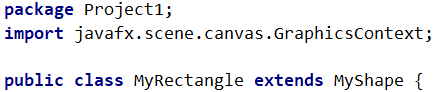
 MyRectangle extends MyShape, is part of the package Project1 and uses graphics context for its draw method as seen on the right.

Figure : Beginning of MyRectangle class

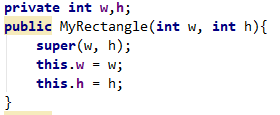
 The constructor for MyRectangle takes in the width and height passed in when the object was instantiated and sets the that as the values for the local variables of the width and height of the object. It also calls the constructor of the MyShape class with those parameters using the super() method.

Figure : Constructor for MyRectangle class

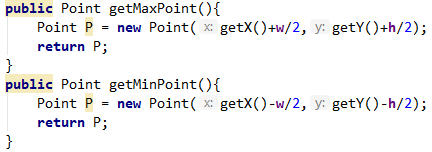
 These two methods retrieve the coordinates of the corners of the MyRectangle object. They were created so that we could use these coordinates to see if they were within the range of each other so that we could tell if they were overlapping. To get these coordinates we retrieved the directional coordinate for the middle of the object and for the X coordinate we added half the height of the object to get the largest valued corner coordinate and we subtracted half the width to get the smallest valued corner coordinate for the x direction. We then did the same thing using the height for the Y direction. These methods are critical for our implementation of doOverlap in our MyShape class.

Figure : getMaxPoint and getMinPoint methods that assist in the doOverlap method for the MyShape hierarchy

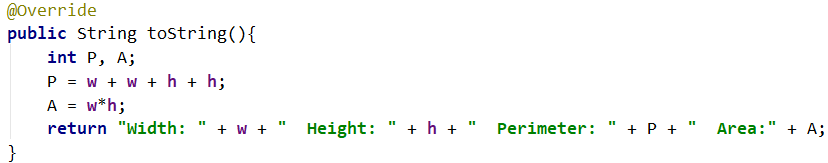
 The overridden method toString returns a string with the width, height, perimeter, and area of the rectangle. We calculated perimeter by adding the length of each side of the rectangle together and the area by multiplying the width by the length of the rectangle. The output of the method can be seen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

Figure : toString method

Figure : Output of toString method

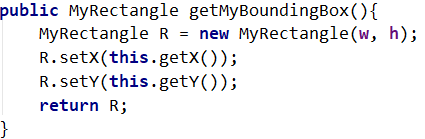
 For our second project we had to implement the getMyBoundingBox method in each of our classes. For our MyRectangle class we created a copy of the current MyRectangle object and sets its width, height, x, and y values as the same.

Figure : getMyBoundingBox implementation for the MyRectangle class

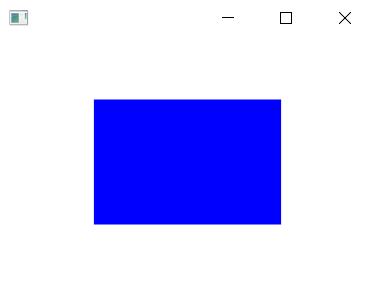
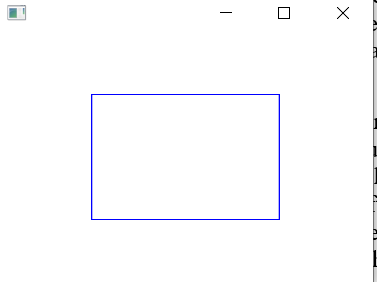
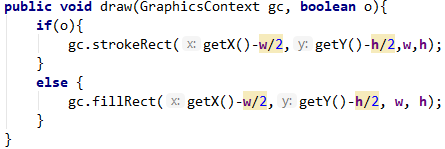
 The overridden method draw has parameters graphics context and boolean o. If the parameter o is true, then draw will put the outline of a rectangle of width w and height h with being positioned based off where the top left corner is. To calculate where the top left corner is in relation to our middle point(x, y) we did that component minus half the size of the object in that direction which would result in the corner. 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.

Figure : draw method

Figure : Draw output with O = true

Figure :draw output with O = false

# Class MyOval

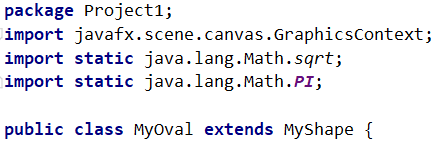
MyOval extends MyShape, is part of the package Project1, uses graphics context for its draw method, sqrt and PI from the lang.Math for its getPerimiter and getArea method as seen on the right. Since MyOval extends MyShape this means it inherits all the methods of the MyShape class and if there are any abstract methods in MyShape, such as draw, the MyOval class must implement them. This also means that any classes that inherit from the MyOval class, such our MyCircle class, will inherit all the methods from MyShape in addition to the methods of MyOval. This inheritance by the MyCircle class also includes inheriting the implemented versions of the abstract methods inherited from MyShape that were implemented in MyOval.

Figure : Snippet of beginning of the MyOval class

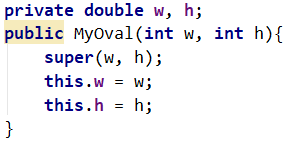
The constructor for MyOval initiates two local variables to set the width and height of the oval, which are twice the value of the vertical and horizontal axes of the oval. In our old implementation of this class we had four variables, however this was unecessary since we will consitently use width and height and do operations on them as needed when we want the vertical and horizontal axes for example.

Figure : Constructor for MyOval class

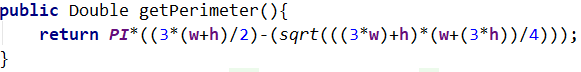
getPerimeter is a method that returns the perimiter of the oval as a double type value. We calculate the perimiter of the oval using an approximation made by the mathmatician Ramanujan.2 This approximation is fairly accurate while only requiring basic algebraic calculations.

Figure : getPerimeter method

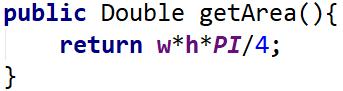
For getArea we used oval Area equation which is the vertical\*horizontal axes \* PI. It is similar to the area equation of a circle expet instead of having one radius we have to different ones that we multiply together. Since both w and h in our method is double the value of the axes they are assosciated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

Figure : getArea method

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 w and h 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.



Figure : toString method

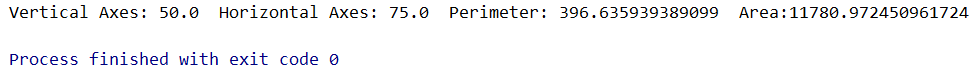


Figure : Output of toString method for MyOval

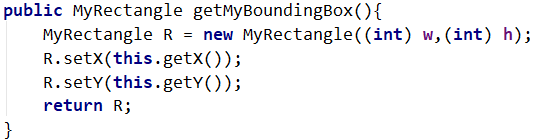
 The getMyBoundindBox method returns a MyRectangle object that has the same width height and location as the oval object. We do this by passing in the same width and height values and we set the x and y values for the center of the object to be the same as the oval object it was called on.

Figure : getMyBoundingBox method for MyOval

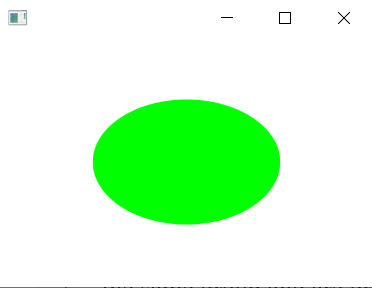
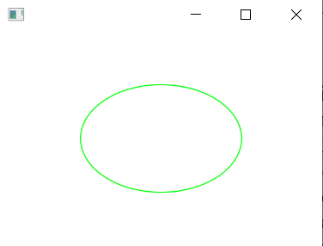
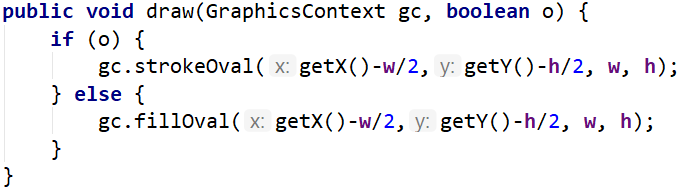
 The draw method is an overridden method that can either draw the outline of the shape onto the canvas or put a solid filled shape onto the canvas. The parameters are graphics context gc and boolean o. Graphics context gc is the instance that was created in main so that they are all part of the same group 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. For this project we also had ot change he way the we implemented the coordinates of where the shape is placed. To calculate where the top left corner is in relation to our middle point(x, y) we did that component minus half the size of the object in that direction which would result in the corner.

Figure : draw method for MyOval

Figure : Output of draw method onto canvas with 0=true

Figure : Output of draw method onto canvas with O = false

# 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.



Figure : First part of MyColor enum

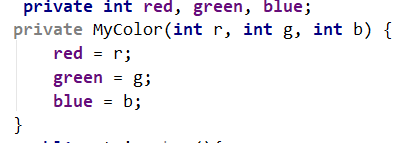
 The constructor for the enum type can take red, green, and blue integer values and use them as a custom color. We can then use different methods that will make the custom color that was passed in usable for graphics context objects.

Figure : Constructor for MyColor enum

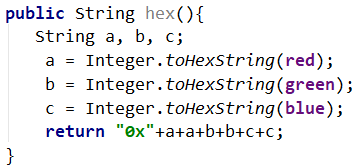
 Our first method returns a string with the hex number of the enum color chosen. It is designed to mimic the way that javafx.scene.paint.Color returns chosen colors. The output of brown, red = 102; green = 51; blue = 0, can be seen below. 

Figure : hex method

Figure : output of hex method

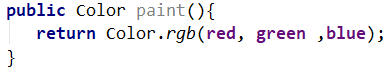
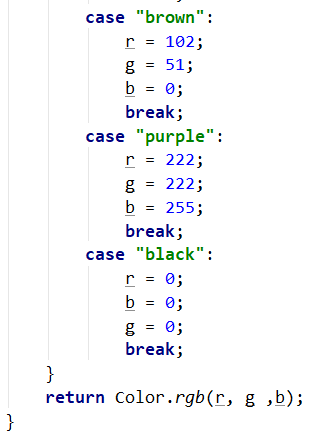
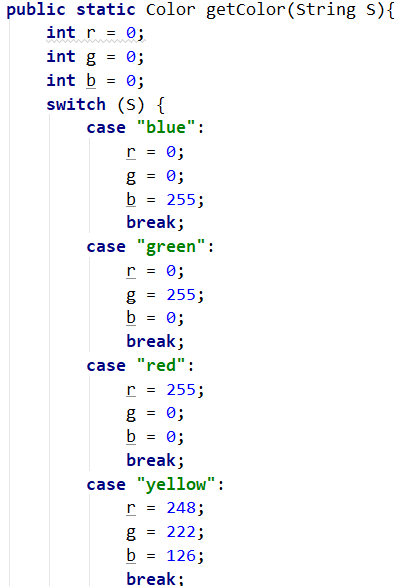
 The method paint uses the enum value and will return the JavaFX color type associated with that value. That way we could call mycolor.BLUE.paint() and It would be like calling color.BLUE. This is the most important method of the enum type as it is what we use in order to paint the canvas and return color types that graphics context can use.

Figure : paint method, most important for our use

 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.

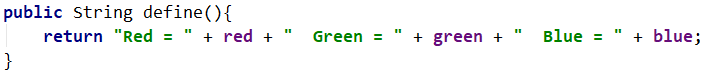
The define method returns a string with the red, green, and blue values of the enum selected color. The code can be seen below and output for the color BROWN is below the code. 

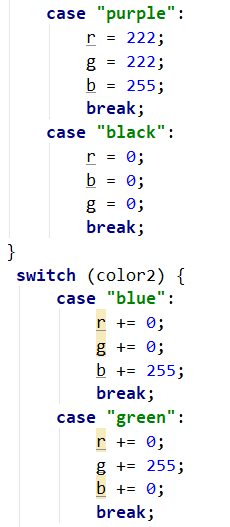
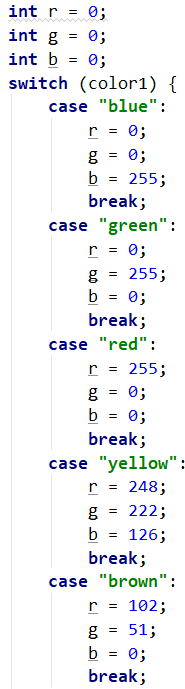
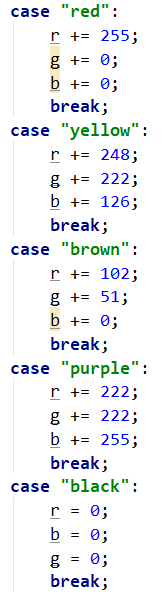
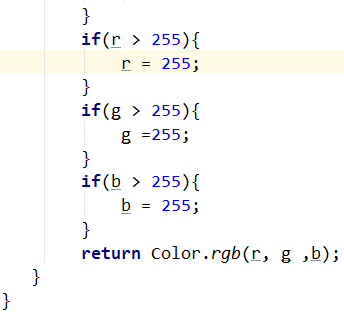
Figure : define method



Figure : Output of define method

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

In this assignment we dealt more with abstract classes and interfaces. This forced us to use inheritance more and understand multiple inheritances. Abstract classes are important because they allow us to have a set of methods that work for most subclasses, but also contain critical methods that must be unique for each subclass. This better than just overloading in some cases because there are some methods that just can’t work for both classes, and this will ensure that the implementer will tailor the method to that subclass. We saw this specifically with our draw method. The method had to be different for each subclass because they were drawing different shapes which meant that they must call different methods from graphics context.

Similar to abstract classes, are interfaces which we also used in this project. The core difference between interfaces and abstract classes is that all the methods in an interface must be implemented in another class, but the interface can inherit multiple interfaces. We saw this with the MyPositionInterface which was inherited by the MyShapeInterface which was then implemented by MyShape, which mean that the MyShape class was now implementing two interfaces while also inheriting the object class. The interfaces are also useful since they add structure to the hierarchy. They allow us to have a list of methods that are accessible from this package that can be used for any of the object that inherit the interface, since they are forced to implement if they inherit the interface.

The way in which we implemented doOverlap was not perfect. AS mentioned above in the report when the corners of the bounding boxes were overlapping but the shapes themselves were not overlapping our method still returned true. One way to fix this would be to make doOverlap into an abstract method in MyShape and create a unique implementation for each subclass, as well as helping methods in each subclass so that they can return useful information to be used for each different kind of object. One implementation of this could be creating a method that returns a certain number of equally spaced points that were mapped to be in the same places of different shapes. Then we could do what we did in our implementation, except that we wouldn’t be comparing two points of a rectangle but let’s say one hundred points that go around the perimeter of the object. There are also issues with this method and it is not clear that it would work at all.

This project was extremely interesting and helped us reinforce and improve up what we already created while also learning new concepts. Using the base of our previous project was extremely helpful in creating a bridge for learning the new concepts and implementing the new methods. This report contains the processes that were undergone in order to create the code, as well as what each piece of code does and why it is important, and most importantly, develops the concepts which were asked for us to understand by doing this project.

# Works Cited

|  |  |
| --- | --- |
| [1] | J. Page, "Incircle of a Polygon," https://www.mathopenref.com/polygonincircle.html, California, 2011. |

**[2]** “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.***PI***;  
**import static** java.lang.Math.*tan*;  
  
  
**public class** Main **extends** Application{  
 **public static void** main(String[] args){  
 *launch*(args);  
 }  
 **public static final int *w*** = 600, ***h*** = 400;  
 @Override  
 **public void** start(Stage stage) {  
 Group root = **new** Group();  
 Canvas canvas = **new** Canvas(***w***, ***h***);  
 GraphicsContext gc = canvas.getGraphicsContext2D();  
  
 *//Circle 1* **double** r = ***h***/2-15;  
 MyCircle C1 = **new** MyCircle(***w***/2,***h***/2);  
 C1.setRadius(r);  
 C1.*setColor*(gc, MyColor.***PUPLRE***);  
 C1.setX(***w***/2);  
 C1.setY(***h***/2);  
 C1.draw(gc, **false**);  
 C1.getMyBoundingBox().draw(gc, **true**);  
 System.***out***.println(C1.toString());  
  
 *//Polygon 1* MyPolygon P1 = **new** MyPolygon(***w***,***h***);  
 P1.setN(9);  
 P1.setL(r);  
 MyShape.*setColor*(gc, MyColor.***YELLOW***);  
 P1.setX(***w***/2);  
 P1.setY(***h***/2);  
 P1.draw(gc, **false**);  
 P1.*setColor*(gc, MyColor.***PUPLRE***);  
 P1.getMyBoundingBox().draw(gc, **true**);  
  
 *//Circle 2* r = (2\*r/(3.4\**tan*(***PI***/5)));  
 MyCircle C2 = **new** MyCircle(0,0);  
 C2.setRadius(r);  
 C2.*setColor*(gc, MyColor.***GREEN***);  
 C2.setX(***w***/2);  
 C2.setY(***h***/2);  
 C2.draw(gc, **false**);  
  
 *//Polygon 2* MyPolygon P2 = **new** MyPolygon(***w***,***h***);  
 P2.setN(5);  
 P2.setL(r);  
 P2.*setColor*(gc, MyColor.***RED***);  
 P2.setX(***w***/2);  
 P2.setY(***h***/2);  
 P2.draw(gc, **false**);  
  
 *//Circle 3* r = (r\*2/(3.4\**tan*(***PI***/5)));  
 MyCircle C3 = **new** MyCircle(0,0);  
 C3.setRadius(r);  
 C3.*setColor*(gc, MyColor.***BLUE***);  
 C3.setX(***w***/2);  
 C3.setY(***h***/2);  
 C3.draw(gc, **false**);  
  
 *//Polygon 3* MyPolygon P3 = **new** MyPolygon(***w***,***h***);  
 P3.setN(5);  
 P3.setL(r);  
 P3.*setColor*(gc, MyColor.***BROWN***);  
 P3.setX(***w***/2);  
 P3.setY(***h***/2);  
 P3.draw(gc, **false**);  
  
 *//Circle 4* r = (r\*2/(3.4\**tan*(***PI***/5)));  
 MyCircle C4 = **new** MyCircle(0,0);  
 C4.setRadius(r);  
 C4.*setColor*(gc, MyColor.***WHITE***);  
 C4.setX(***w***/2);  
 C4.setY(***h***/2);  
 C4.draw(gc, **false**);  
  
 *//Line 1* MyLine L1 = **new** MyLine(***w***,***h***);  
 L1.*setColor*(gc, MyColor.***BLACK***);  
 L1.draw(gc, **true**);  
  
 *//Line 2* MyLine L2 = **new** MyLine(***w***,***h***);  
 L2.*setColor*(gc, MyColor.***BLACK***);  
 L2.setX1(0);  
 L2.setX2(***w***);  
 L2.setY1(***h***);  
 L2.setY2(0);  
 L2.draw(gc, **true**);  
 L2.getMyBoundingBox().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* }  
 }

## MyShape

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
**import** javafx.scene.paint.Paint;  
**import** java.awt.Point;  
**import** java.lang.Object;  
**import static** java.lang.Math.*sqrt*;  
  
**public abstract class** MyShape **extends** Object **implements** MyShapePositionInterface{  
 **private int w**, **h**;  
 **public** MyShape(**int** w, **int** h){  
 **this**.**w** = w;  
 **this**.**h** = h;  
 }  
 **private int x**, **y**;  
 **public void** setX(**int** X){ **x** = X; }  
 **public void** setY(**int** Y){  
 **y** = Y;  
 }  
 **public static void** setColor(GraphicsContext gc, MyColor O){  
 gc.setFill(O.paint());  
 gc.setStroke(O.paint());  
 }  
 **public int** getX(){  
 **return x**;  
 }  
 **public int** getY(){  
 **return y**;  
 }  
 **public** Paint getColor(GraphicsContext gc){  
 **return** gc.getFill();  
 }  
 @Override  
 **public** String toString(){  
 String s = **"no object specified"**;  
 **return** s;  
 }  
 **public abstract void** draw(GraphicsContext gc, **boolean** o);  
 **public** Point getPoint(){  
 Point P = **new** Point(getX(), getY());  
 **return** P;  
 }  
 **public void** moveTo(**double** dx, **double** dy){  
 **x** += dx;  
 **y** += dy;  
 }  
 **public double** distanceTo(**int** a, **int** b){  
 **return** *sqrt*(((**x**-a)\*(**x**-a))+((**y**-b)\*(**y**-b)));  
 }  
 **public abstract** MyRectangle getMyBoundingBox();  
 **public boolean** doOverlap(MyShape O) {  
 MyRectangle M = getMyBoundingBox();  
 MyRectangle R = O.getMyBoundingBox();  
 **if** (M.getMaxPoint().**x** <= R.getMinPoint().**x**) {  
 **return false**;  
 } **else** {  
 **if** (M.getMinPoint().**x** >= R.getMaxPoint().**x**) {  
 **return false**;  
 } **else** {  
 **if** (M.getMaxPoint().**y** <= R.getMinPoint().**y**) {  
 **return false**;  
 } **else** {  
 **if** (M.getMinPoint().**y** >= R.getMaxPoint().**y**) {  
 **return false**;  
 } **else** {  
 **return true**;  
 }  
 }  
 }  
 }  
 }  
}

## MyCircle

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
**import static** java.lang.Math.***PI***;  
  
**public class** MyCircle **extends** MyOval {  
 **private double radius**;  
 **private int w**,**h**;  
 **public** MyCircle(**int** w, **int** h) {  
 **super**((w+h)/2, (w+h)/2);  
 **radius** = (w+h)/4;  
 }  
 **public double** getRadius(){  
 **return radius**;  
 }  
 **public void** setRadius(**double** r){  
 **radius** = r;  
 **w** = (**int**) (**radius**\*2);  
 **h** = (**int**) (**radius**\*2);  
 }  
 @Override  
 **public** Double getPerimeter() {  
 **return** (2 \* ***PI*** \* **radius**);  
 }  
 **public** Double getArea(){  
 **return *PI***\***radius**\***radius**;  
 }  
 **public** String toString(){  
 **return "Radius: "** + **radius** + **" Perimeter: "** + getPerimeter() + **" Area: "** + getArea();  
 }  
 **public** MyRectangle getMyBoundingBox(){  
 MyRectangle R = **new** MyRectangle((**int**) **radius**\*2,(**int**) **radius**\*2);  
 R.setX(**this**.getX());  
 R.setY(**this**.getY());  
 **return** R;  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o){  
 **if**(o){  
 gc.strokeOval(getX()-**radius**,getY()-**radius**,**radius**\*2,**radius**\*2);  
 }  
 **else** {  
 gc.fillOval(getX()-**radius**,getY()-**radius**, **radius**\*2, **radius**\*2);  
 }  
 }  
}

## MyPolygon

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
  
**import static** java.lang.Math.***PI***;  
**import static** java.lang.Math.*abs*;  
**import static** java.lang.Math.*cos*;  
**import static** java.lang.Math.*sin*;  
**import static** java.lang.Math.*sqrt*;  
**import static** java.lang.Math.*tan*;  
  
**public class** MyPolygon **extends** MyShape {  
 **private double N**, **L**;  
 **double w** ,**h**;  
 **public** MyPolygon(**int** w, **int** h) {  
 **super**(w, h);  
 **this**.**w** = w;  
 **this**.**h** = h;  
 **this**.**L** = h;  
 }  
 **public void** setN(**double** sides){  
 **N** = sides;  
 }  
 **public void** setL(**double** l) {  
 **L** = l;  
 }  
 **private double**[] getxPoints(){  
 **double**[] xPoints = **new double**[(**int**) **N**];  
 **for**(**int** i = 0; i < **N**; i++){  
 xPoints[i] = **L**\**cos*((-***PI***/2) - i\*2\****PI***/**N**) + getX();  
 }  
 **return** xPoints;  
 }  
 **private double**[] getyPoints(){  
 **double**[] yPoints = **new double**[(**int**) **N**];  
 **for**(**int** i = 0; i < **N**; i++){  
 yPoints[i] = **L**\**sin*((3\****PI***/2) + i\*2\****PI***/**N**) + getY();  
 }  
 **return** yPoints;  
 }  
 **private double** getboundX(){  
 **double** minX, maxX;  
 maxX= 0;  
 minX = **L**\**cos*((-***PI***/2) - 2\****PI***/**N**) + getX();  
 **for**(**int** i = 0; i < **N**; i++){  
 **double** temp;  
 temp = **L**\**cos*((3\****PI***/2) + i\*2\****PI***/**N**) + getX();  
 **if**(temp > maxX){  
 maxX = temp;  
 }  
 **if**(temp < minX){  
 minX = temp;  
 }  
 }  
 **return** (maxX-minX);  
 }  
 **private double** getboundY(){  
 **double** minY, maxY;  
 maxY = 0;  
 minY = *abs*(100\***L**\**sin*((3\****PI***/2)+2\****PI***/**N**) + getY());  
 **for**(**int** i = 0; i < **N**; i++) {  
 **double** temp;  
 temp = *abs*(**L**\**sin*((3\****PI***/2)+ i\*2\****PI***/**N**)+getY());  
 **if** (temp > maxY) {  
 maxY = temp;  
 }  
 **if** (temp < minY) {  
 minY = temp;  
 }  
 }  
 **return** (maxY-minY);  
 }  
 @Override  
 **public** String toString(){  
 **return "side length: "** + **L** + **" interior angle (deg): "** + (180-360/**N**) + **" perimeter: "** + **L**\***N**;  
 }  
 **public** MyRectangle getMyBoundingBox() {  
 MyRectangle R = **new** MyRectangle((**int**)(getboundX()),(**int**) (getboundY()));  
 R.setX(**this**.getX());  
 **if**(**N**%2 != 0) {  
 R.setY((**int**) (**this**.getY()+(*sqrt*(**N**)/(**N**-1))\*((**L**/(2\**tan*((***PI***/**N**))))-(**L**/((2\**sin*(***PI***/**N**)))))));  
 }  
 **else**{  
 R.setY(**this**.getY());  
 }  
 **return** R;  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o){  
 **if** (o) {  
 gc.strokePolygon(getxPoints(), getyPoints(), (**int**) **N**);  
 } **else** {  
 gc.fillPolygon(getxPoints(), getyPoints(), (**int**) **N**);  
 }  
 }  
}

## MyPositionInterface

**package** Project1;  
**import** java.awt.Point;  
  
**public interface** MyPositionInterface {  
 Point getPoint();  
 *// gets the point x, y which in  
 // our MyShape class hierarchy  
 //is the middle of the shape  
 // it will return it as a  
 //point which is a pair of integers* **void** moveTo(**double** dx, **double** dy);  
 *//Move the object from the  
 //the point x ,y to the point  
 //(x + dx), (y + dy)  
 //it moves the object  
 //by dx in the X direction  
 //and by dy in the Y direction* **double** distanceTo(**int** a, **int** b);  
 *// this returns the distance to  
 // the point x, y which is stored  
 // in our class. In our MyShape  
 // class x, y is the center of  
 // the shape.*}

## MyShapePositionInterface

**package** Project1;  
  
**public interface** MyShapePositionInterface **extends** MyPositionInterface{  
 MyRectangle getMyBoundingBox();  
 *//Returns a MyRectangle  
 //that is the same width  
 //and the height as the  
 //object and in the same  
 //position. It returns a  
 //bounding rectangle of the  
 //object.* **boolean** doOverlap(MyShape O);  
 *//Returns true if the the  
 //object it was called with  
 //and the object passed into  
 //the method are overlapping  
 //otherwise it will return  
 //false.*}

## MyLine

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
  
**import static** java.lang.Math.*abs*;  
**import static** java.lang.Math.*atan*;  
**import static** java.lang.Math.*pow*;  
**import static** java.lang.Math.*sqrt*;  
  
**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;  
 }  
 **public void** setX1(**int** x1){  
 **this**.**x1** = x1;  
 }  
 **public void** setX2(**int** x2){  
 **this**.**x2** = x2;  
 }  
 **public void** setY1(**int** y1){  
 **this**.**y1** = y1;  
 }  
 **public void** setY2(**int** y2){  
 **this**.**y2** = y2;  
 }  
 @Override  
 **public** MyRectangle getMyBoundingBox(){  
 MyRectangle R = **new** MyRectangle((**int**) *abs*(**x1**-**x2**), (**int**)*abs*(**y1**-**y2**));  
 R.setX((**int**) (**x1**+**x2**)/2);  
 R.setY((**int**) (**y1**+**y2**)/2);  
 **return** R;  
 }  
 **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**);  
 }  
}

## MyRectangle

**package** Project1;  
**import** javafx.scene.canvas.GraphicsContext;  
  
**import** java.awt.Point;  
  
**public class** MyRectangle **extends** MyShape {  
 **private int w**,**h**;  
 **public** MyRectangle(**int** w, **int** h){  
 **super**(w, h);  
 **this**.**w** = w;  
 **this**.**h** = h;  
 }  
 **public** Point getMaxPoint(){  
 Point P = **new** Point(getX()+**w**/2,getY()+**h**/2);  
 **return** P;  
 }  
 **public** Point getMinPoint(){  
 Point P = **new** Point(getX()-**w**/2,getY()-**h**/2);  
 **return** P;  
 }  
 @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** MyRectangle getMyBoundingBox(){  
 MyRectangle R = **new** MyRectangle(**w**, **h**);  
 R.setX(**this**.getX());  
 R.setY(**this**.getY());  
 **return** R;  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o){  
 **if**(o){  
 gc.strokeRect(getX()-**w**/2,getY()-**h**/2,**w**,**h**);  
 }  
 **else** {  
 gc.fillRect(getX()-**w**/2,getY()-**h**/2, **w**, **h**);  
 }  
 }  
}

## 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 w**, **h**;  
 **public** MyOval(**int** w, **int** h){  
 **super**(w, h);  
 **this**.**w** = w;  
 **this**.**h** = h;  
 }  
 **public** Double getPerimeter(){  
 **return *PI***\*((3\*(**w**+**h**)/2)-(*sqrt*(((3\***w**)+**h**)\*(**w**+(3\***h**))/4)));  
 }  
 **public** Double getArea(){  
 **return w**\***h**\****PI***/4;  
 }  
 @Override  
 **public** String toString(){  
 **return "Vertical Axes: "** + **h**/2 + **" Horizontal Axes: "** + **w**/2 + **" Perimeter: "** + **this**.getPerimeter() + **" Area:"** + **this**.getArea();  
 }  
 **public** MyRectangle getMyBoundingBox(){  
 MyRectangle R = **new** MyRectangle((**int**) **w**,(**int**) **h**);  
 R.setX(**this**.getX());  
 R.setY(**this**.getY());  
 **return** R;  
 }  
 **public void** draw(GraphicsContext gc, **boolean** o) {  
 **if** (o) {  
 gc.strokeOval(getX()-**w**/2,getY()-**h**/2, **w**, **h**);  
 } **else** {  
 gc.fillOval(getX()-**w**/2,getY()-**h**/2, **w**, **h**);  
 }  
 }  
}

## MyColor

**package** Project1;  
**import** javafx.scene.paint.Color;  
  
**public enum** MyColor{  
 ***BLUE***(0, 0 , 255), ***GREEN***(0, 255, 0), ***RED***(255, 0, 0), ***YELLOW***(248, 222, 126),  
 ***BROWN***(102, 51, 0), ***PUPLRE***(228, 0, 228), ***BLACK***(0,0,0), ***WHITE***(255, 255, 255);  
 **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);  
 }  
}