JavaFX Project Part 2

Cristian Statescu

Parts 1 & 2 (setting up the hierarchy and interface)

Introduction

To start off, the first problem of this homework assignment was to make a class hierarchy where the parent(super) class is an **abstract** class named MyShape (which extends the Java class Object, which every class has as a super class according to Oracle Docs). Then, in the hierarchy, there are 5 subclasses, 4 of which extend the MyShape abstract class, while one of them extends one of the 4 just mentioned subclasses. The 4 subclasses which extend directly off of MyShape are named MyLine, MyArc, MyRectangle, and MyOval. Finally, the last subclass that is part of the hierarchy extends off of MyOval and is called MyCircle. Along with this hierarchy, we were to create a class MyPoint which was to be used by all classes mentioned previously, along with an enum class named MyColor which all classes in the hierarchy and the class MyPoint would use, and finally, an interface which is named MyShapeInterface. The point of making MyColor, MyPoint, and the MyShape class hierarchy with the interface MyShapeInterface is to use the whole "system" to draw a geometric configuration of a circular pizza pie arbitrarily (random **OR** chosen) sliced (to be discussed further in the report), along with the intersection of 2 different rectangles and the intersection of a rectangle and a circle shape, and bounding rectangles of different subclasses (MyLine, MyArc, and MyCircle) of MyShape, using JavaFX.

Solution

The Enum Class "MyColor"

The Enum Constants and Class Variables

First, to begin solving the solution to Part 1 of this assignment, I made the enum class MyColor first, which defines the color of all shapes. All constants of colors created were taken from the link:
https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/Colors.html.
It provided me with the RGB values of all the different types of colors I needed for my JavaFX. All the color constants has 5 variables: 'r', 'g', 'b', 'a' and 'rgba'. The first four variables stand for red, green, blue, and alpha, and range from values 0 to 255. The variables 'r', 'g', and 'b' define what level of 'intensity' each of these colors is in the final color (0 means none, 255 means the most 'intensity' of the color). The variable 'a' (alpha) stands for the opacity of the color (0 means transparent, and 255 means opaque, with the middle values being intermediate values between transparent and opaque). The 'rgba' variable is a string which makes up the hexadecimal value of colors as such: 0xRRGGBBAA. Each variable 'r', 'g', 'b', and 'a' have 2 hexadecimal digits to signify their value. The reason this is the case is because the maximum value you can have with 2 hexadecimal digits is 255, which as mentioned previously is the maximum of all the single variables 'r', 'g', 'b', and 'a'.

Class Attributes, Constructors, and Methods

This class only has one method which took in values for ‘r’, ‘g’, ‘b’, and ‘a’. It uses 5 methods - setR, setG, setB, setA, setRGBA - to set the values of ‘r’, ‘g’, ‘b’, and ‘a’ that was passed into the constructor, and then uses the setRGBA to make the hexadecimal string for the enum constant.

To explain the constructor further, the setR, setG, setB, setA took in the values of the respective variable that they worked with (‘r’ for setR and so on) and set the value of the constant by the use of the ‘this’ keyword (by the code, for example, this.r = red; (red being value passed into the method). If the value passed into one of these 4 methods was not in the range 0-255, the value of the respective variables would be set to 0 (just for the sake of consistency and making the values secure and accurate). setRGBA worked in a more complex manner because it has to work with hexadecimal values. It took in no parameters and instead just took the ‘r’, ‘g’, ‘b’, ‘a’ values (already previously set) by using the GetR, GetG, GetB, and GetA methods (to be explained after this one). Then, using a wrapper String classes instance for all ‘r’, ‘g’, ‘b’, and ‘a’ values to use other methods on the individual strings, the method format (of class String, which I learned about from https://www.geeksforgeeks.org/java-string-format-method-with-examples/) is used to format all instances of the String wrapper classes previously mentioned. The reason this method of the String class is used to format all new instances of the String objects is to ensure that there are only 2 hexadecimal digits in each respective value set for the 4 variables, hence why each initialization of each string hexadecimal variable has String.format("%02X", (0xFF & ‘variablename’)), since it takes the variable taken from the Get methods and transforms it into a

2 digit hexadecimal string which is what is needed based on what the description of the ‘rgba’ variable was. Once all of the hexadecimal (wrapper) String instances of the ‘r’, ‘g’, ‘b’, ‘a’ instances are made using the String.format method, the Strings are all combined into a single String instance called ‘rgbalower’ using the + operator for Strings to concatenate them. Lastly, the final string called ‘newrgba’ is produced by combining “0x” with the rgbalower string (which becomes capitalized by using the .toUpperCase() String wrapper class method to make it look like a typical set of hexadecimal numbers: with only uppercase letters to represent values over 9). Then the String ‘rgba’ is set to the variable of the enum instance using (once again) the ‘this’ keyword: “this.rgba = newrgba;” .

NOTE: All Set methods are void methods because they do not return any value and instead only set values of variables of the instance they are “working on”.

The “Get” methods – which comprise of the methods GetR, GetG, GetB, GetA, GetRGBA, and GetJavaFXColor – of the enum class MyColor take no parameters and use the ‘this’ keyword to return the respective value of the “Get” method. For example, GetR() just does “return this.r;” and GetRGBA() just does “return this.rgba;”. The one that is slightly more complex is the GetJavaFXColor, which uses the Color JavaFX class method .valueOf. The function has one line of code: “return Color.valueOf(this.GetRGBA());” What the .valueOf method of the JavaFX class Color does is returns the respective Color constant from the JavaFX class based on the hexadecimal string of style 0xRRGGBBAA. In the GetJavaFXColor method, I made it so that the RGBA string of each MyColor constant was passed inside of the .valueOf method. Doing so makes the GetJavaFXColor return the JavaFX color of the same value as the hexadecimal string passed inside of the .valueOf method, which will be used to make JavaFX canvas drawings in **problem 3.**

The Class “MyPoint”

The Class Variables

The class MyPoint consists of double variables ‘x’, and ‘y’, and a MyColor variable named ‘color’. ‘x’ and ‘y’ signify the cardinal points of any instance of MyPoint (which will be used for **problem 3** in addition with the JavaFX Canvas). The MyColor variable ‘color’ will be used to give the point a color using the MyColor class.

Class Attributes, Constructors, and Methods

The class MyPoint has 4 constructors: one default constructor (no parameters passed), 2 parametrized constructors, and a copy constructor. The default constructor takes no parameters and defaults the instance of MyPoint to have ‘x’ and ‘y’ set to (0, 0) and the color to be set to MyColor.BLACK, all done with the use of the Set methods of MyPoint (explained further). The first parametrized constructor for MyPoint takes values for ‘x’, ‘y’, and ‘color’. It sets the value of the double ‘x’ and ‘y’, and the MyColor ‘color’ by using the respective Set methods of MyPoint. The color for this constructor actually has an extra bit inside of the Set_MyColor method: “Set_MyColor(Optional.ofNullable(newcolor).orElse(MyColor.BLACK));”. The Optional… part of this line of code ensures that no bizarre value is input into the color part of the parameter, and if there is a bizarre value (of null), the ‘color’ variable is defaulted to BLACK to prevent issues with the object (this is used throughout the whole project). The second parametrized constructor for MyPoint only takes in values for ‘x’ and ‘y’ (only 2 values are taken in by the method). For this constructor, no MyColor value is passed and the MyColor value color for the instance of MyPoint created by the second parametrized constructor is set to MyColor.BLACK. All variables are set in the second parametrized constructor with the use of the Set methods of the MyPoint class. The last constructor that the class MyPoint has is the copy constructor. This constructor takes in one value, that being an instance of MyPoint that will be copied into a new instance of MyPoint. This constructor uses the Set methods of MyPoint to set all the variables of the new instance to the object that was passed in, and inputs the values to be passed to the Set methods using the Get methods (covered in this section) of the class MyPoint to get values of ‘x’, ‘y’, and ‘color’ from the instance that is to be copied to the new instance of the MyPoint class.

Next, the Get and Set methods of the MyPoint class. The methods were: Get_x, Get_y, Get_MyColor, Set_x, Set_y, and Set_MyColor. All of the Get methods take one parameter and are one line, which uses the ‘this’ keyword to set the value of the instance they are working on to the value passed into them (for example: this.x = xnew;). For the Set methods, all of them are void functions (return nothing) because they only set values passed into them (those being the respective types of the Get function – for example, the Set_MyColor function will have a MyColor enum constant as the input). All Set functions use the ‘this’ keyword to set the values of the MyPoint instance.

Finally, the final methods of the MyPoint class is Add, Sub, StringPt and draw. The Add method returns a new MyPoint instance created from the addition of the ‘x’ and ‘y’ variables of two inputted MyPoint instances (the Add method takes in two MyPoint instances to add the ‘x’ and ‘y’ values of both inputted instances, and gets the values from each inputted instance using the Get_x and Get_y method). The method uses the parametrized constructor without the color input to make the new instance, rather the ‘color’ variable of the sum instance defaults to MyColor.BLACK. If someone wishes to change the color of the sum instance, they can do so using the Set_MyColor method. The Sub method works the same as the Add method, but instead subtracts the second inputted instance from the first. It is made using the same parametrized constructor and uses the Get_x and Get_y methods to get the ‘x’ and ‘y’ values from the inputted instances. Both Add and Sub are static methods because they do not need instances to base off of. Rather, they need two instances inputted to be worked on, and because of this should belong to the class rather than to each individual instance. The StringPt method prints out an inputted MyPoint instance in the format: (x,y). It uses the Get methods to get the ‘x’ and ‘y’ variable values of the instance passed into it. It is a static method because, while it would work as a non-static method, one can just pass the instance inside of the method and get the string of the point without needing direct access to the instances’ attributes, and so should belong to the class rather than to all individual cases. StringPt accepts a point parameter (because it is a static method), and is a string method because it returns a string of the coordinates (excluding the color because it is only a single, and there would be too much information in a string otherwise, hence people can just use the Get_MyColor method to see the color of their point) of the MyPoint instance. Lastly, the draw method: this method takes in a JavaFX GraphicsContext instance names ‘GC’ to set up for drawing on a JavaFX canvas. The method uses the setFill method from the GraphicsContext class (not a static method) to pick the color that will be drawn on the JavaFX canvas. The line of code for the first part of the draw method is: GC.setFill(this.Get_MyColor().GetJavaFXColor());. The ‘this’ keyword is used to use get the instance that the draw method is being used on. Get_MyColor() is used to get the instances ‘color’ value. Lastly, the GetJavaFXColor method is used to get the JavaFX color which is to be used in the setFill method as the input, since only JavaFX Color values are inputted into the method setFill. Then, for the second part of the draw method is the method from the GraphicsContext class called fillRect, which takes in 4 values: the ‘x’ and ‘y’ value of the top left corner point of a rectangle that is to be drawn by the method, and then for the last two parameters, they take in the height and the width of the rectangle. To draw the pixel out, the ‘x’ and ‘y’ coordinates are taken from the instance the draw method is being used on by using the respective Get methods for both variables, and then passing in 1 for both the height and the width parameters of the fillRect method. Doing so makes a single pixel – or point in our case of graphing – (height and width of 1 pixel) at the coordinate (x,y) (whatever the ‘x’ and ‘y’ values may be).

The Interface “MyShapeInterface”

The Interface Variables

Because of how I decided to make this interface, there are no variables present in it, since this interface (for my solution) is a collection of 4 methods.

The Interface Methods

An interface is a reference type of many methods which can be implemented in different ways (if abstract) and act as a “link” for a general use of methods between objects of different types that are in the same hierarchy (which is what two of these methods are used for). Two of the four methods are abstract methods, those being getBoundingRectangle (returns a MyRectangle object that encapsulates the shape on which it is called) and pointInMyShape (takes in a MyPoint object and returns a Boolean value for wether or not the MyPoint object passed in is in the shape the method is working/being called on). These two methods, being abstract, are to be overridden in every subclass of the hierarchy (Note the word subclasses, the abstract superclass MyShape will not need to override these classes because it implements MyShapeInterface for the entire hierarchy, hence it gives the subclasses a connection to the interface). The last two methods are static methods (one of which – drawIntersectMyShapes – was supposed to be a default method, however, it did not seem fitting to have to dot reference from another object of a subclass in the heirarchy to call on this method, so instead it is static and can be called directly from the interface), and they are the intersectMyShapes method (takes in two subclasses of the heirarchy, a GraphicsContext instance, and a MyColor instance, and returns an ArrayList of MyPoint objects that have all the points that are in both shapes passed into the function) and drawIntersectMyShapes. IntersectMyShapes works by using the pointInMyShape method for each respective shape passed into the method and iterates through all points in the canvas (which is also input into the method) using 2 for loops (one representing x's values for each point, and one representing y's values). Then, using the ArrayList (that contains objects of MyPoint as each element) instance that was created at the start of this method running (named intersectingpoints), if the point that an iteration of the loop in the method is in both of the shapes input into the method, then that point is added to intersectingpoints ArrayList. Once the loop is done iterating through the whole canvas' possible points, the method returns the ArrayList intersectingpoints. Lastly, the drawIntersectMyShapes method creates a new Canvas instance and new GraphicsContext instance, draws the 2 shapes that are input into it onto the new canvas, and then iterates through a for loop that draws all the elements created by the IntersectMyShapes method. Every iteration of the loop remakes the IntersectMyShapes ArrayList (because of a lack of a simple way to fully copy an ArrayList) and checks the respective element that the loop variable is on so that it may draw it to the canvas. Lastly, the canvas of the GraphicsContext is returned.

The Abstract Super (or Parent) Class “MyShape”

The Class Variables

The abstract class MyShape has only two variables: a MyPoint instance ‘p’ and a MyColor ‘color’. The ‘p’ acts as a reference point, and the ‘color’ defines the color of each instance of MyShape.

Class Attributes, Constructors, and Methods

The class MyShape is an abstract class and (the superclass of the class hierarchy being formed) has 6 constructors (which all use Set methods like all previous constructors from the previously mentioned classes and Get methods from previous classes to get data from ‘p’ and ‘color’ or parameters passed inside of the methods). Firstly, an abstract class (such as MyShape) is a class in which an object cannot be made directly out of it (the only way to access it would be through inheritance from another subclass, making it a good type of class to have for a superclass in our class hierarchy). Note that despite the fact this class is abstract, constructors should still be used as good practice (super constructors are needed in all subclasses, which will use there constructors). The default constructor sets ‘p’ to a MyPoint instance with coordinate (0,0) and the default MyColor.BLACK using the default MyPoint constructor. Additionally, the color for the MyShape instance is set to MyColor.BLACK. One parametrized constructor only takes in a MyPoint instance as a parameter and sets the variable ‘color’ by default to MyColor.BLACK. Another parametrized constructor takes in both a MyPoint instance and a MyColor instance. It follows the same code as the last constructor mentioned, but just uses the code Optional.ofNullable(newcolor).orElse(MyColor.BLACK); once again (as the previous constructors from the previous classes have done) to ensure that the color is kept as a correct and proper value. If an improper value is passed, the color of the instance will be set to MyColor.BLACK – **from here on out in this report, this will be common for all constructors from ANY class that take in a MyColor parameter).** Then, 2 constructors take in ‘x’ and ‘y’ and make new MyPoint ‘p’ instances that default to MyColor.BLACK as the color and have ‘x’ and ‘y’ as the coordinate point values. One of these constructors takes in a MyColor parameter, while the other does not (and so defaults ‘color’ to MyColor.BLACK). Finally, the copy constructor. This last constructor takes in a MyShape instance as a parameter and forms a new MyShape object with the same data as the instance passed into the constructor (using Set methods).

The Set methods in the MyShape class (Set_Point, Set_MyColor) take in the respective value for its function (such as Set_Point(MyPoint p) taking in a MyPoint instance). They are both void functions since they only set variables and do not return anything. They use the ‘this’ keyword to set values for the instance they are working on.

The Get methods in the MyShape class (Get_Point, Get_MyColor) return the value associated with their name using the ‘this’ keyword (ex. Get_Point{ return this.p;}

There are four abstract methods which are to be overridden by all the subclasses in the hierarchy, those being the perimeter, area, draw, and toString methods. In MyShape, they are abstract methods, meaning they are a method that are just declared but do not have any implementation (given what the class of MyShape consists of and the fact that it's an abstract class). This is done because the MyShape class be used to lay a groundwork for all classes in the hierarchy (which is its purpose as an abstract class in this hierarchy). In the subclasses of this hierarchy, these methods also exist and **MUST** get overridden (by the rules of abstract methods and hierarchy according to ORACLE Docs) to make sure they work for each of the individual subclass they are in because they are abstract methods from an abstract class that all classes in the hierarchy draw from (or "extend" from).

The Class "MyLine"

The Class Variables

Each instance of MyLine has 3 variables, 2 MyPoint instances 'p1' and 'p2' (which hold the start and end points of the line) and a MyColor instance 'color' (which holds the color of the line).

Class Attributes, Constructors, and Methods

This class extends off the MyShape abstract superclass. All of the 4 constructors use the line "super(new MyPoint(), null);" because all subclasses must get the superclass methods and access the superclass constructors (and in these subclasses, all the MyShape variables are defaulted to a default MyPoint and null (in the "super" line) because the variables do not matter to the MyLine instance). All constructors also use the Set methods of the MyLine class, and the copy constructor uses Get methods of the MyLine class aswell. The first constructor takes in no parameters and defaults both MyPoint variables of the instance to default constructor MyPoint instances (no inputs, (0,0) coordinates, MyColor.BLACK as the color). The next 2 constructors take in 2 MyPoint instances. The only difference between the two is that one of them does not take in a MyColor instance while the other does. For the one that does, it uses the Optional line (like previous constructors from other classes with MyColor as a variable) to ensure a valid color is assigned. The constructor that doesn't have any MyColor instance passed into it makes the instance have the color MyColor.BLACK. Lastly, the copy constructor takes in a MyLine instance and uses Get and Set methods of MyLine to make a new instance that has the same values as the MyLine instance passed into the method.

The class MyLine has some methods which are overridden using the @Override feature to override methods from the abstract superclass MyShape, and some which are entirely unique to the MyLine class. Its first method is xAngle, which returns a double value, that being the degrees of the angle from the x-axis to the line. The angle is measured using the atan2 Math class method, which turns rectangular coordinates into polar coordinates (r, theta) and returns the angle theta (https://www.programiz.com/java-programming/library/math/atan2#:~:text=The%20Java%20Math%20atan2(),the%20angle%20theta%20(%CE%B8).&text=Here%2C%20atan2()%20is%20a%20static%20method.). Input into the function is the difference of the 'x' and 'y' (which is taken from using Get methods of MyLine and MyPoint) of the two end points of the MyLine instance as the rectangular coordinate point. However, the 'x' and 'y' values are placed the in the other variable location (like so (y,x)) because JavaFX canvas' work by having the y-axis go positive in the downwards direction, and so must be flipped to get the correct angle measurement from the axis' used by the canvas. The difference is used from 'p1' to 'p2' because it will get me the angle directly from the x-axis to the line if the line instance is positive in the sense of the canvas axes. Otherwise, if it is not, doing this ensures that the correct point is given for atan2 (since on the JavaFX canvas, Q1 is shown in the sense that y and x are positive, however, the y axis goes downwards as its values increase, and giving the difference essentially gives the point that would be in the regular cartesian plane). Before returning the angle, the return value of the tan2 is subtracted by 180 degrees because of the way the JavaFX canvas is measured. Essentially, in most mathematical 2D cartesian planes, the angle is measured from the positive x-axis in quadrant 1, but because of the way the canvas plane is, the angle is measured from the 2nd quadrant on the x-axis and hence atan2 will give me an answer for a regular cartesian plane rather than the JavaFX plane, which is why the method subtracts 180 from the atan2 method's return value. The return of the xAngle method is always going to be positive, indicated by the if statement inside of it, to avoid confusion about the ambiguity of angles in the JavaFX canvas. This is mainly because I wanted the angle of the lines to be measured from the x-axis directly to the line (in a clockwise fashion) because it would provide a simpler view of the angle.

Next, the Get methods of MyLine use the 'this' keyword to return the value associated with their name. (ex. Get_MyColor() { return this.color;}) The Get methods are Get_MyColor, Get_Point1, Get_Point2, and Get_Line. Get_MyColor is overridden because there is a method of the same name in the abstract superclass MyShape.

The next methods are the perimeter and area methods (which return doubles because perimeter and area values are rarely integers in math), which are overridden from the abstract superclass MyShape. Although they are overridden, both return a value of 0 for MyLine because lines do not consist of a perimeter nor an area. Rather they build up other shapes which do have perimeters and areas.

The other method that measures and returns a double type value of a MyLine instance is the length method. This method uses the Distance Formula method of 2 points, which is coded by using the sqrt and pow methods from the Math class. The method also uses MyPoint Get methods to get the x and y values of each endpoint from the MyLine instance (for reference, the Distance Formula: https://www.khanacademy.org/math/geometry/hs-geo-analytic-geometry/hs-geo-distance-and-midpoints/a/distance-formula).

The Set methods of the MyLine class are void methods which set the value of each variable their name is associated with by using the 'this' keyword. The Set methods of MyLine are Set_Point1, Set_Point2, and Set_MyColor(which is overridden using @Override in this class because there exists a method in the superclass named Set_MyColor).

The toString method returns a description in the form of a String of the instance of MyLine that it is working on. It uses wrapper classes of Double and assigns the coordinates of both endpoints to Double instances using the MyPoint methods to get all cooridnates of both endpoints, and wrapping the length and angle values into Double instances as well. The reason of using wrapper instances of Double is to be able to use the toString method of wrapper classes so as to make the toString method of MyLine output the desired string with all the information of the instance (as seen in the code).

The method draw is an overridden (from the abstract superclass MyShape) void method that draws the MyLine instance. It takes in an instance of a GraphicsContext. Then, using the GraphicsContext method setStroke, along with Get methods Get_MyColor (class MyLine) – which is overridden because a method of the same name exists in abstract superclass MyShape –

and GetJavaFXColor (class MyColor), it sets the color of the line to be drawn by passing in a JavaFX color from the GetJavaFXColor method of MyColor. The setLineWidth method of the GraphicsContext class is used to set the width of the MyLine instance to be drawn to 1 (the value input is the thickness of a stroke on the JavaFX canvas, and for that input, 1 was passed into it to make the soon to be drawn line be 1 pixel thick). Lastly, the GraphicsContext method strokeLine is used to draw the line. It takes in the ‘x’ and ‘y’ coordinates of the 2 endpoints of the line that is to be drawn. To use this method, the Get_Point1 and Get_Point2 methods are used (from class MyLine), along with Get_x and Get_y (from class MyPoint). The final methods are overridden from the abstract methods from the interface MyShapeInterface. Those two methods are getBoundingRectangle and pointInMyShape. For getBoundingRectangle, what the method in MyLine does is create a new MyRectangle instance, a new top left corner point MyPoint instance, and two double variables for the height and width of the new rectangle. Then, using the two endpoints of the MyLine instance the method is called on from (and 2 if-else blocks), the top left corner point of the new rectangle is created out of the smallest x and y values of the twopoints of the MyLine instance (because of the way the axes work in JavaFX, the more to the top left of the screen, the smaller the x and y values). Addtionally, in the same if-else blocks, the height and width of the rectangle is determined, and is gotten by taking the difference between the two points’ x and y values and adding 2 to each difference so that the rectangle will actually encapsulate the line fully and not have its corners on the rectangle itself (adding 2 because I removed 1 from the smallest value of x and y for the TLCP for the exact same reason, to ensure an actual bounding rectangle). The rectangle is then set using set methods and is then returned. The pointInMyShape method (overridden because it is an abstract class in the interface) takes in a point to test if it is on the line the method is called from. How it works is by checking to see if the sum of the lengths of the line segmented into 2 from the starting point to the called on point, and then from the called on point to the end point of the line equals the same as the length of the original line (using the length method of MyLine for all measurements), which would ensure that the point is actually on the line, because otherwise, if the point is not on the line, the sum will not be the same as that of the original line. If it is equal, the method returns true, otherwise, it returns false.

The Class “MyRectangle”

The Class Variables

Each instance of the class MyRectangle has 4 variables: doubles ‘h’ and ‘w’ (standing for height and width respectively), an instance of MyPoint named TLCP (standing for the top left corner point of the rectangle that the MyRectangle instance is representing), and an instance of MyColor named ‘color’ (which represents the color of the rectangle).

Class Attributes, Constructors, and Methods

The class MyRectangle extends off of the abstract superclass MyShape. The MyRectangle class consists of 4 constructors: 1 default constructor, 2 parametrized constructors, and 1 copy constructor. All of these constructors use the “super(new MyPoint(), null);” line because all subclasses must get the superclass methods and access the superclass constructors. It defaults both “super” values to default points because they are irrelevant to the MyRectangle instances. Additionally, all constructors use Set methods of the class MyRectangle to work. The default constructor sets ‘h’ and ‘w’ both to 0, sets ‘TLCP’ to a default constructor made instance of MyPoint, and sets ‘color’ to MyColor.BLACK. The two parametrized constructors both take in values for height and width, along with a MyPoint instance. The difference between them is one takes in a MyColor instance while the other does not. The constructor without a MyColor instance passed into it sets ‘color’ of the instance it is creating to MyColor.BLACK, while the one that does take in a MyColor instance uses the Optional method (used in previously mentioned constructors) to ensure that an actual MyColor value is stored inside of ‘color’. Finally, the copy constructor takes in an instance of the MyRectangle class and creates a new instance of MyRectangle with the same values for all variables as the instance that was passed into the copy constructor.

All Set methods (which are all void because they only set variables to values rather than returning anything) of MyRectangle class use the ‘this’ keyword to set the value of the variable its name corresponds to, to the value that is passed into each respective Get method. The Get methods of the class MyRectangle are Get_Height, Get_Width, Get_TLCP, and Get_MyColor. Get_MyColor is overridden by using the @Override line in MyRectangle because a method of the same name exists for the abstract superclass MyShape. The Get methods return the values of the variable their name is associated with (ex. Get_Height(){ return this.h;})

Lastly, the 6 remaining methods are all methods which are overridden because they exist in the abstract superclass MyShape or the interface MyShapeInterface (the last two discussed will be from the interface). The first of these methods is perimeter, which returns the perimeter of the MyRectangle instance by calculating its value using the Get methods (to retrieve them first) of MyRectangle to get the values of ‘h’ and ‘w’ for the MyRectangle instance, then multiplying both values by two and adding them to get the perimeter of the MyRectangle instance. That value that

that was just mentioned is returned in this function (which is a value of type “double” to account for non-integer perimeters). The second of the 6 methods is area, which calculates and returns the area (as a double to account for non-integer areas) of the respective MyRectangle instance by multiplying the height and width values of the instance of MyRectangle it is working on (it gets the ‘h’ and ‘w’ values by using the MyRectangle Get methods). The third of the 6 methods is the toString method, which returns the information of the MyRectangle instance it is working on by using wrapper classes to use the toString method of wrapper classes. It works similarly to the MyLine toString method, but instead it returns a string that has the coordinates of the top left corner point, height, width, perimeter, and area of the MyRectangle instance. The fourth of the 6 methods is draw, which takes in a GraphicsContext instance and draws out the rectangle of the MyRectangle instance. It works the same way that the draw method worked for the MyPoint class. It uses the setFill method of GraphicsContext and the Get_MyColor (this time of class MyRectangle) and Get_JavaFXColor (of class MyColor) to set the color of the rectangle that is to be drawn. Then, it uses the method fillRect of the GraphicsContext class to draw the rectangle of the instance. The method takes in the individual ‘x’ and ‘y’ values of the ‘TLCP’ of the instance (with the use of Get_TLCP from MyRectangle and Get_x and Get_y of the MyPoint class), and then also the width and the height of the respective rectangle through the use of the Get_Height and Get_Width of the MyRectangle class. The fillRect class works by using the coordinate point values it takes in to make a point that acts as the top left corner point of the rectangle that is to be drawn. Then, off of that corner point, using the width and height values input into it, it makes a rectangle with those 2 values, starting off from the top left corner expanding to the height and width given to the method. The fifth of the 6 methods is getMyBoundingRectangle, which is from the MyShapeInterface (an abstract method in the interface). What getMyBoundingRectangle does in MyRectangle is take the corner point of the rectangle that it is being called from and creating a new TLCP for the rectangle by subtracting 1 from the x and y values of the point gotten from the rectangle that is to be bounded. This is done so that the new rectangle will bound the old rectangle and not just overlap it. Lastly, the width and height of the new rectangle is made by adding a value of two to the current rectangles height and width and setting the values to those, once again to ensure an actual bounding rectangle. The rectangle is created using the MyRectangle parametrized constructor without any MyColor enum constant taken into it. The final method is pointInMyShape (another abstract method from the interface MyShapeInterface). This method takes in a MyPoint instance and checks if the point is in the MyRectangle instance from which the method is called. It does this by a simple if statement, which checks to see if the x and y points are in the range of the TLCP of the rectangle plus the width or height of the rectangle (depending on if it is looking at the x value of the point of the y value). If the MyPoint instance's values are in the range (including on the perimeter of the rectangle), then the method returns true, otherwise it returns false.

The Class "MyOval"

The Class Variables

The variables of the MyOval class are doubles ‘h’ and ‘w’ (representing height and width), a MyPoint instance ‘cntr’ (representing the center of the oval/ellipse), and a MyColor instance ‘color’ (representing the color of the oval/ellipse).

Class Attributes, Constructors, and Methods

The class MyRectangle extends off of the abstract super class MyShape. The MyOval class consists of 4 constructors: 1 default constructor, 2 parametrized constructors, and 1 copy constructor. All of these constructors use the “super(new MyPoint(), null);” line because all subclasses must get the superclass methods and access the superclass constructors, while also setting the 2 values of the super class to a default MyPoint instance and a null MyShape ‘color’ value because they are irrelevant to the MyOval class. Additionally, all constructors use Set methods of the MyOval class. The default constructor sets variables of the instance they are making like so:‘h’ and ‘w’ to 0, MyPoint ‘cntr’ to a new MyPoint instance created by a default MyPoint constructor, and the ‘color’ variable set to MyColor.BLACK. The 2 parametrized constructors take in values for ‘h’, ‘w’, and ‘cntr’, but one of them doesn’t take a MyColor instance in as a parameter, while the other does. The one that doesn’t set’s the color of the MyOval instance to MyColor.BLACK, while the one that does take an instance of MyColor in as a parameter uses the “Optional” line used in previous constructors for other classes to ensure that an actual MyColor constant is assigned to ‘color’. Finally, the copy constructor takes in an instance of MyRectangle that is to have all of its information retrieved using the Get methods of the MyOval class, and then copied over to the new instance of MyOval.

The Get methods of the class MyOval use the ‘this’ keyword to return the respective value that their name is associated with. The Get methods of MyOval are Get_Center, Get_Height, Get_Width, Get_MyColor (which is overridden because a method of the same name exists in class MyShape), Get_SemiMinorAxis, and Get_SemiMajorAxis. The last two are a bit unique, since they do not return a variable in an instance, but rather calculate and see what the value of the semi-minor and semi-major axes are. The way it does this is by first seeing if ‘h’ is higher than ‘w’ by first getting these variables using the Get_Height and Get_Width methods of the MyOval class, and then using an if, else, statement to return the correct value (semi-minor axis is the smaller of height or width divided by 2 (half of the height or width, whichever is smaller – this is a general definition for ellipses).

The Set methods of the MyOval class are all void methods, since they only set the variables of an instance rather than returning any value. The Set methods take in a value that is to be set inside of an instance (say, for Set_MyColor, the parameter it takes is a constant of the enum MyColor), and use the ‘this’ keyword to set the value that is input into the method to the appropriate variable of the instance of MyOval. The Set methods of MyOval are Set_Height, Set_Width, Set_MyColor (which is overridden because a method exists in MyShape that has the same name), and Set_Cntr.

The last 6 methods are all overridden, as methods of the same name exist in the MyShape class. The first of these remaining 6 methods is perimeter. The method perimeter of the MyShape class returns the perimeter (of type double) of the MyOval instance. Given there is no actual simple method to find the perimeter of an ellipse, I used an approximation I found searching through the internet (https://www.cuemath.com/measurement/perimeter-of-ellipse/ - approximation formula 3).

$$P \approx \pi\left[\frac{3}{2}(a+b) - \sqrt{ab}\right]$$

Note that a is the semi-major axis, and b is the semi minor axis, which were gotten through the use of the Get_SemiMajorAxis and Get_SemiMinorAxis methods of MyOval. Using the Math library (for the constant pi and sqrt method), I wrote out the equation into the perimeter method, and the result of this mathematical equation is what would come out. The second of the last 6 methods is area, which returns the area of the MyOval instance. For ellipses, there is a clear "plug-and-chug" equation that gets the area (that being PI * a * b). I used the Math class constant PI and the Get_SemiMajorAxis and Get_SemiMinorAxis methods of MyOval to code this formula. The result of the formula is what the area method of MyOval returns. The third of the last 6 methods is the toString method, which uses wrapper classes (and Get methods) to wrap up all variables ('cntr' is wrapped up as two individual Double instances signifying 'x' and 'y' in the MyPoint 'cntr' variable of an instance of MyOval) in an instance of MyOval, along with the perimeter, semi-major/minor axis values, major/minor axis values. The method then uses the .toString method of wrapper classes and returns a string that is concatenated with all the information of the instance of MyOval. The fourth of the 6 methods is the draw method, which draws the MyOval instance on the JavaFX canvas. Firstly, a GraphicsContext instance is passed which will be used in the draw method. The setFill method is used to set the color of the ellipse that is to be drawn (by passing the MyOval Get_Color method, and then the Get_JavaFXColor method of MyColor). Lastly, the fillOval method of the GraphicsContext class is used, which fills in an oval bounded by a rectangle that is defined in the values passed inside the method (https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html). Inside of this method, the MyOval center's x value minus half of the width (retrieved using the Get methods of MyOval and MyPoint) of the ellipse is input as the 'x' value of the top left corner of the rectangle that the bounded oval is to be drawn inside of (note, the rectangle isn't drawn, this is simply just how the function works). The ‘y’ value of the top left corner of the rectangle that the bounded oval is to be drawn inside of is retrieved the same way, by getting the ‘y’ value of the ‘cntr’ variable and subtracting it by half of the height (retrieved using the Get methods of MyOval and MyPoint). This is actually done because the last two values taken in by the fillOval method is the width and height of the (essentially imaginary) rectangle that will encapsulate the drawn oval (since the rectangle is defined by the top left corner point, half of the width must be subtracted from the ‘x’ value of ‘cntr’ because of how the JavaFX canvas is made, as well as half of the height must be subtracted from the ‘y’ value of ‘cntr’). Hence, using the Get methods of MyOval to get the height and width of the MyOval instance, those values are input into the fillOval method and so draw the oval with the color that was previously set. The fifth of the 6 remaining methods is getMyBoundingRectangle (abstract class in the MyShapeInterface interface). This method works by creating a new top left corner point for the rectangle that is to be made by taking the center of the oval that this method is called from, and then subtracts half the width and half the height from the respective x and y values of the center. Lastly, the rectangle is made by simply inputting in the newly created point along with the height and width of the oval the method was called from into the parametrized constructor (the one that doesn’t take an MyColor enum constant). The reason this method is simple is because all the information is provided from the oval itself, since the draw method of oval is similar to that of the rectangle, most of the code was inspired from what was in the draw method. The method then returns the newly created rectangle. Lastly, the final method in this class is the pointInMyShape method, which takes in a point and checks to see if the point is in the oval or not. What this method does is use the ellipse formula to check if the point is in the ellipse or not. It plugs in the input points x and y values into the ellipse formula and checks to see if the outcome is less than or equal to 1 (which all ellipse equations are equal to). Using the if statement, if the condition is met, the method returns true, otherwise, the method returns false.

The Class "MyCircle"

The Class Variables

The variables of the MyCircle class are quite simple, having a MyPoint instance named 'cntr' for the center of the circle, a MyColor instance named 'color' for the color of the circle, and a double variable named 'r' for the radius of the circle.

Class Attributes, Constructors, and Methods

MyCircle actually extends off of the MyOval class in the hierarchy and so will be slightly different, but will still have a connection to the interface MyShapeInterface, and the MyShape abstract superclass. The MyCircle class has 4 constructors: 1 default constructor, 2 parametrized constructors, and a copy constructor. All of the methods use the Set methods of the class to set the values in each instance to be created (the Set methods will be discussed shortly), also have the line "super();" to initialize the default MyOval constructor (which is common good practice). The default constructor takes in no parameters and creates a circle that has a point at the origin (point made using the MyPoint default constructor), a radius of 5 (to make a circle and not a dot), and a color of MyColor.BLACK. The 2 parameterized constructors are identical except for the fact that one takes in a MyColor enum constant while the other does not, and rather defaults the color of the created MyCircle instance to MyColor.BLACK. Otherwise, the two constructors do the same thing: set the center of the to be created circle to the 'center' MyPoint instance passed into the constructor, and set the radius of the to be created circle to the 'newr' double value passed into the constructor. The copy constructor just takes in a MyCircle instance and takes all the values inside of the passed in MyCircle instance using the Get methods of MyCircle(to be discussed shortly) and sets the values of the soon to be created MyCircle instance to what was taken from the passed in MyCircle instance.

The Get and Set methods of the MyCircle class just use the “this” keyword to get and set values of a MyCircle instance. The Get methods are Get_Center, Get_MyColor (both of which are overridden because they are in the MyOval class), and Get_Radius. The Set methods are Set_Radius (not overridden because MyOval doesn’t have this method), Set_MyColor, and Set_Cntr (both of the last two methods are overridden because they are in the MyOval class).

The perimeter and area methods of MyCircle are overridden because they were part of the MyOval class, as well as the MyShape abstract class. To calculate the perimeter of the MyCircle instance, the perimeter method just does the formula of “2 * pi * r” using the Math.PI constant and the Get_Radius method to get the radius of the MyCircle instance. The method then returns the calculated perimeter. To calculate the area of the MyCircle instance, the area method simply uses the equation “pi * r^2” using the Math.PI constant and the Get_Radius method to get the radius of the MyCircle instance. The method then returns the calculated area. The toString method is overridden from the MyOval super class (which was also overridden from the MyShape abstract superclass). In the MyCircle class, it makes 5 instance of the Double wrapper class (for the perimeter – gotten from the method previously mentioned, area– gotten from the method previously mentioned, radius – get method, and x and y values of the center of the MyCircle instance – get method). The method then returns a string that neatly organizes all of these values by using the wrapper class’ method of toString.

The getMyBoundingRectangle method and pointInMyShape method are both from the MyShapeInterface interface (where they are abstract methods for the class hierarchy), as well as from the MyOval class that MyCircle extends from. The getMyBoundingRectangle method makes a new MyRectangle instance using the default constructor of MyRectangle, and then sets the height, width, and TLCP of the newly created rectangle by using the MyRectangle class Set methods. The height and width of the rectangle (which will actually be a square), are just the radius of the circle times 2 (which is also just the diameter of the circle). The TLCP of the new rectangle is created by taking the x and y values of the center of the MyCircle instance from which this method I called from and then subtracting the radius from both values. This is done because of how the axes on JavaFX work, where the further you go towards the top left of the screen, the lower the x and y values are. This way of finding the top left corner point also ensures the newly created rectangle will actually bound the MyCircle instance. The pointInMyShape takes in a point and checks to see if the input point is in the MyCircle instance from which the method was called from. It does this by plugging in the x and y values from the point that was passed into the method into the left hand side of the equation of a circle (**(x-h)^2 + (y-k)^2** = r^2). ‘h’ and ‘k’ are the x and y values of the center of the circle (gotten using the Get methods of the MyCircle class). The exponents of the equation are done using the Math.pow function. Then, with a simple if else statement, if the left hand side of the circle equation is less than or equal to that of the r^2 (calculated using the Math.pow and radius (gotten using the Get method of MyCircle)), then the method returns true, otherwise, it returns false. Finally, the draw method of MyCircle is actually identical to that of the MyOval class. It takes in a GraphicsContext instance and uses the setFill method of GraphicsContext to set the color of the soon to be drawn circle. Then, using the fillOval method, the top left corner of the method is set using the x or y –

radius way talked about in the getMyBoundingRectangle method, while the width and height of the fillOval method is set by doing the radius of the MyCircle instance multiplied by 2.

The Class “MyArc”

The Class Variables

The variables of the MyArc class are p1, p2 (both MyPoint instances which mark the two points of the arc), color (MyColor enum constant variable), xanglep1, xanglep2 (both double variables which represent the angle from the x-axis to the respective point on the arc (angles are measured increasingly as they move counterclockwise), and lastly, main (a MyOval instance that the arc is based on).

Class Attributes, Constructors, and Methods

The MyArc class has 4 constructors which all use the Set methods of the MyArc class (to be discussed shortly). The class has one default constructor, 2 parametrized constructors, and one copy constructor, and have angles inside of them that represent the angle size from the x-axis to the line created from the center of the MyOval instance, to one of the points in the MyArc instance – essentially, the angles are where the arc spans from (from xangle1 to xangle2). All constructors of MyArc use the line “super(new MyPoint(), null);” to use the MyShape constructor (as that is what the MyArc class extends from). The default constructor sets main, p1, p2 to default constructed instances of each respective class. It sets the color of the arc to MyColor.BLACK, and sets anglep1 and anglep2 to 0. One of the parametrized constructors takes in a MyOval instance, 2 MyPoint instances, and a MyColor instance. Using the proper set methods, all values are set directly as they are input. Once done, the angles of the MyArc instance are calculated and set by making a new MyLine instance from the center to each of the two points input into the constructor. Then, using the xAngle method of the MyLine class, the angle is returned. Because of how the draw method of the MyArc class is made, the angle that was just calculated is used to subtract from 360 to get the angle that would be if the xAngle method measured the angle counterclockwise instead of clockwise. The other parametrized constructor takes in a MyOval instance, 2 double values for the angles of the arc, and a MyColor enum constant.the MyOval, MyColor, and angle instances/values are set directly from what was passed into the constructor. The p1 and p2 MyPoint instances are set by calculating the x and y values through the use of Math.sin and Math.cos methods and the Math.toRadians method (used to translate the angles input to radian angles). This is done through trigonometric calculations and using the rules of sine (opposite over hypotenuse) and cosine (adjacent over hypotenuse). Then, the points are set by creating new MyPoint instances that have the calculations done inside of the constructor pass in's. The MyPoint constructor used for these is the parametrized constructor that takes in a value for x and y, but no value for the MyColor variable of MyPoint. The last constructor is the copy constructor which just takes in a MyArc instance and creates a copy of the MyArc passed into it by using the Get and Set methods of Arc (to be discussed shortly).

The Get and Set methods just use the "this" keyword to set or return the values that are passed. The methods are Set_MyOval, Set_P1, Set_P2, Set_MyColor (overridden because it is part of the abstract MyShape superclass), Set_Angle1, Set_Angle2, Get_MyOval, Get_P1, Get_P2, Get_MyColor (overridden because it is part of the abstract MyShape superclass), Get_Angle1, and Get_Angle2.

The method length returns the arclength of the MyArc instance. It is calculated by using the angles inside of the MyArc instance, and approximates the arc length by making multiple MyLine instances and doing the length method of the MyLine class. The method first checks to see if which of the 2 angles in the MyArc instance is larger and so then creates a temporary MyArc objects that is only 1 degree apart (based on the loop it is in) and then iterates through all possible angles in the range of the MyArc instance, measuring all the small lines between each of the two points of the temporary MyArc instances (the temporary MyArc instances are made by using the parametrized constructor that takes in angles and calculates the points on the arc). At the end, the sum of these lengths (a double variable named "sum") is returned from the method.

The method perimeter returns the perimeter of the MyArc instance. It does this by measuring the arclength (using the previous function) and also making 2 temporary lines from the center of the arc to the endpoints of the arc, and using the length function of the MyLine class (since the arc is the curved bit of the arc itself, as well as the two lines that connect it to the center of the oval from which it originates). The values of these three calculated lengths is added and is the returned as the perimeter of the MyArc instance.

The method area returns the area of the MyArc instance by using proportions of the MyOval object form which the MyArc instance is based upon. To do this, the first thing the method does is look for which of the two angles input into the MyArc instance is smaller. Then once that is decided and marked using 2 temporary variables (one for the big and the other for the small angle), the equation is calculated for the area using the semimajor and semiminor axes of the MyOval instance that the MyArc instance is based upon, along with the Math.atan and Math.tan methods (to use the angles for the calculation based upon the proportions of the MyOval instance). The value that is calculated is then returned as the area from the method.

The toString method is overriden from the abstract superclass MyShape,and simply creates a bunch of Double wrapper instances of the values of the coordinates of the MyArc instance, the angles of the MyArc method, the area, the arclength, and the perimeter of the instance. The method then returns a neatly organized string with all the values of the MyArc instance well organized, along with the information of the MyOval instance that the MyArc instance is based upon (using the toString method of MyOval to concatenate it with the string that will be returned from the method).

The draw method is overriden from the abstract superclass MyShape and takes in a GraphicsContext instance. Firstly, the method uses the setFill method to set the color of the MyArc instance that is about to be drawn (the color is received using the get method). Then using the fillArc method, the way this method works is actually how the fillOval method works, however it takes in 3 more values, those being the angle that the arc starts at in the oval, and how many degrees the arc spans from the angle that it starts from (calculated by subtracting the first angle of the MyArc instance from the second angle), and also the ArcType constant that tells the jdk how the arc is to be drawn (in what style it is to be drawn). I chose to put it as ArcType.ROUND since it looks closest to the “pizza pie” needed in **problem 3**. Hence, to get the TLCP of the rectangle bounding the main oval of the MyArc instance, the method takes the center of the main oval of the MyArc instance and calculates the point by subtracting the width/2 and height/2 from the x and y values of the center of the MyOval instance inside of the MyArc method. The width and height passed into the method is the width and height of the MyOval (gotten using the get method of MyOval). Doing this draws the MyArc instance.

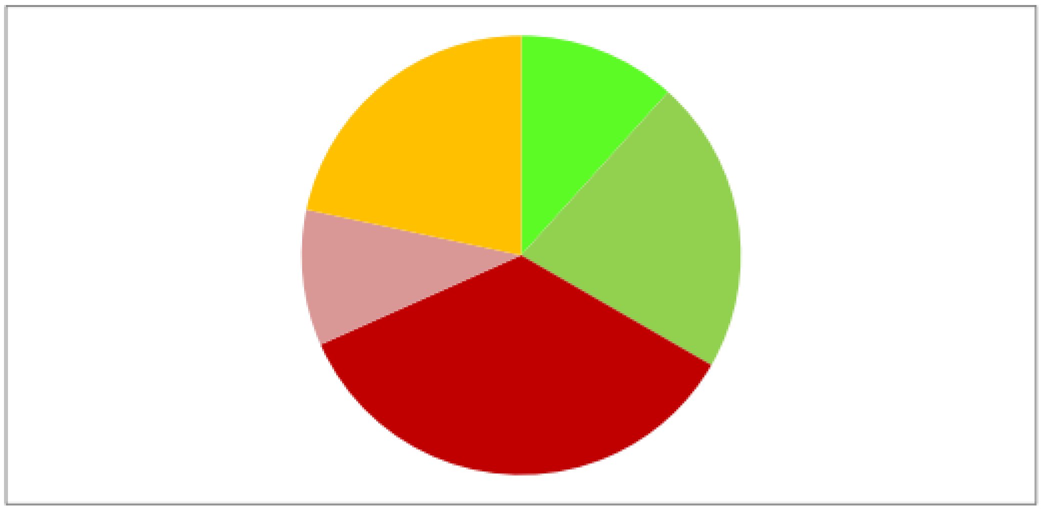
The getBoundingRectangle method of MyArc is overriden as it is a part of the MyShapeInterface interface (as an abstract method). This method simply returns the bounding rectangle of the MyOval instance that the MyArc instance is based on (using the getMyBoundingRectangle method of the MyOval class).

Finally, the pointInMyShape class takes an instance of MyPoint and checks to see if it is in the MyArc instance or not. Firstly, 2 temporary double values are made which keep track of which angle in the MyArc instance is smaller (assignments of these two variables are done by using an if-else block – these variables are used in the next if statement coming up). Then, an if statement first checks if the MyPoint instance that was passed into the method is in the MyOval instance of the MyArc instance first. If it is not, then the method returns false. If it is, then further analysis is done by a nested if statements, which checks if the angle of the line made from the center of the MyOval instance to the MyPoint (doing 360 – the length of the new line by using the xAngle method of new line. Note that 360 was done because xAngle measures the angle clockwise, while the input of MyArc has angles that go counterclock wise) instance is between the two angles that are a part of the MyArc instance (done in the if block). If the calculated angle is between or at that of the MyArc angles, the point is in the MyArc instance, and the method returns true. Otherwise, if that is not the case, the method will return false.

Part 3

Introduction

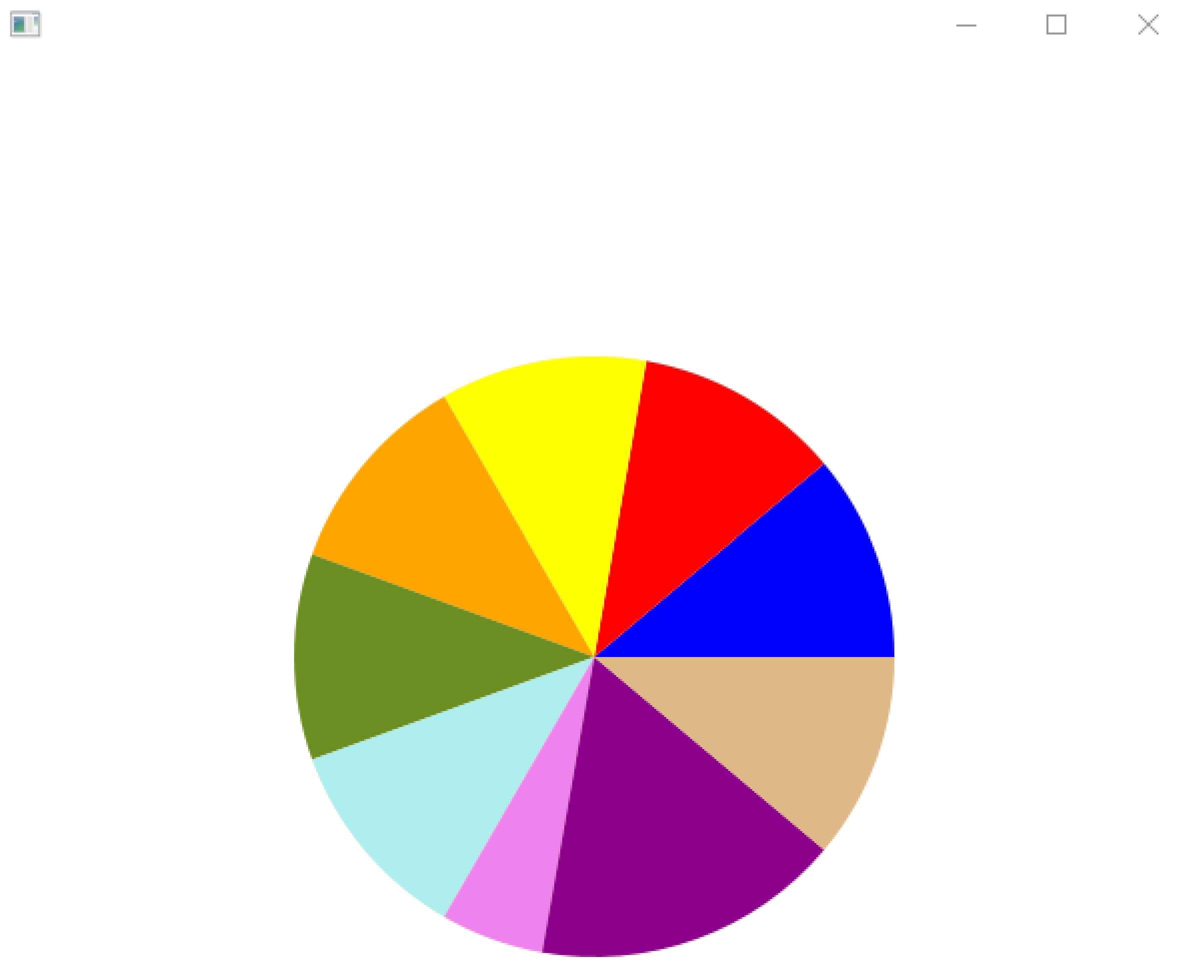
Problem 3 tells us to use the hierarchy and interface we made to draw (for any size of canvas) for a geometric configuration of a circular pizza pie arbitrarily (random **OR** chosen, according to the definition of arbitrarily) sliced, as shown in the sample image in the project pdf:



Additionally, we are also to draw a bounding rectangle of a MyLine instance, a MyArc instance, and a MyCircle instance (or object) of our choice and draw the area of intersection between two MyRectangle objects, and the area of intersection of a MyRectangle object and a MyCircle object.

Solution

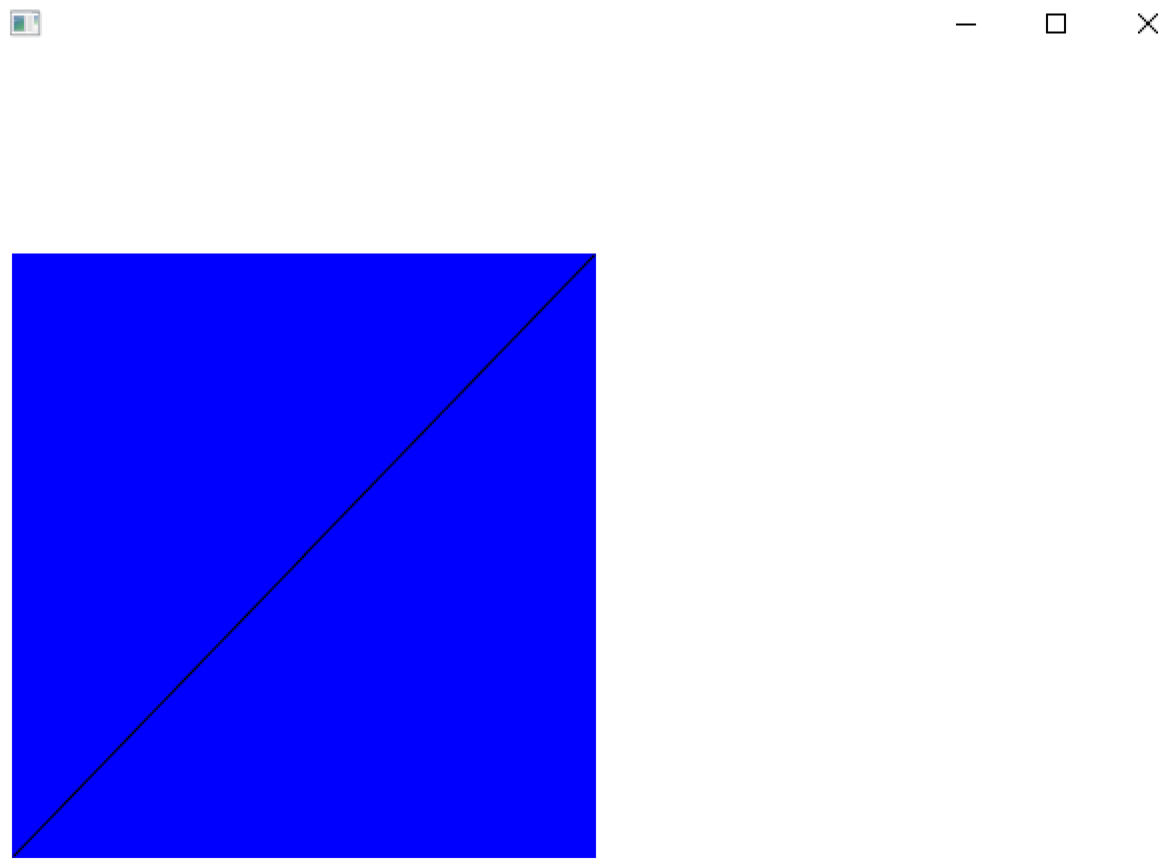
The solution for this part of the project was quite simple as all it really was, was testing what was already made in the hierarchy. I first made Group, scene, stage, canvas, and GraphicsContext instances (boilerplate for drawing on a canvas in JavaFX). To make the "pizza pie" part, I randomly decided that I would find the center point of the canvas using the getWidth and getHeight methods of the Canvas class, make a MyOval instance that will have the center of the canvas as its center, along with the height and width of the MyOval object being half the size of the height and width of the canvas (making a circular MyOval object). Then I created 9 MyArc instances that were mainly of a 40 degree angle size (except for arc7, which was of an angle size of 20, and arc8, which was of an angle size of 60). Additionally, to test out that putting angles backwards in the constructor of a MyArc instance works, I did arc9 by inputting the larger angle first (360), and the smaller angle last (320). Then, using the draw methods of all the shapes created, and the .setScene, .getChildren.add, and the .show methods, the program came out with this result:



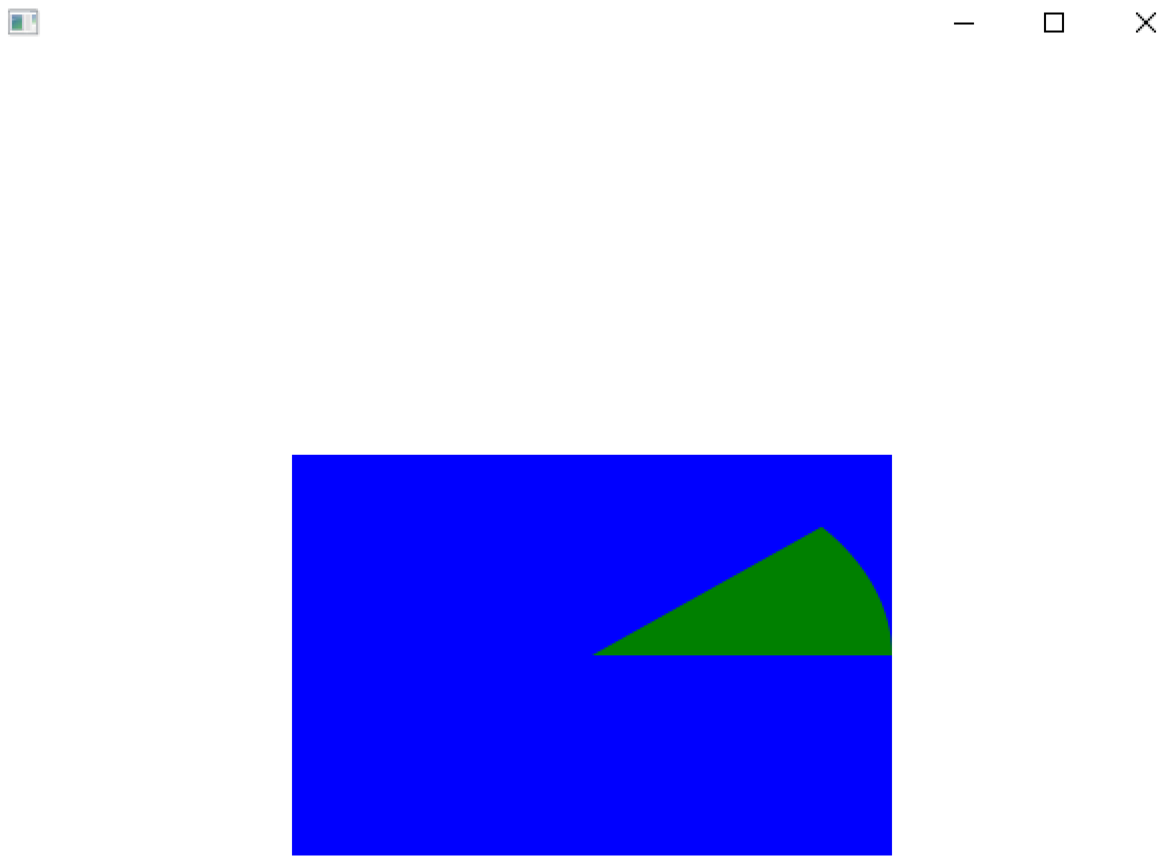
Note, in the code, each part requested for part 3 of this project, individual parts are “commented” out and so must be uncommented and then the other part that was not commented turned into a comment to ensure that the program makes specifically what part of the project was wanted.

The next part of the 3rd part of the project was to draw bounding rectangles of a MyLine instance, a MyArc instance, and a MyCircle instance (or object(s)). All of the above instances were randomly chosen since the whole point of this part was to test and see if the getMyBoundingRectangle methods worked. Hence, in the code, I just used random colors, points, and such to see if the resulting MyBoundingRectangle was correct. I only used the Set_MyColor method for the created bounding rectangles because I wanted the color to not be BLACK and show more neatly in the outcome of the code. Here are the outcomes of the code in the order of which they are asked for in the project pdf:

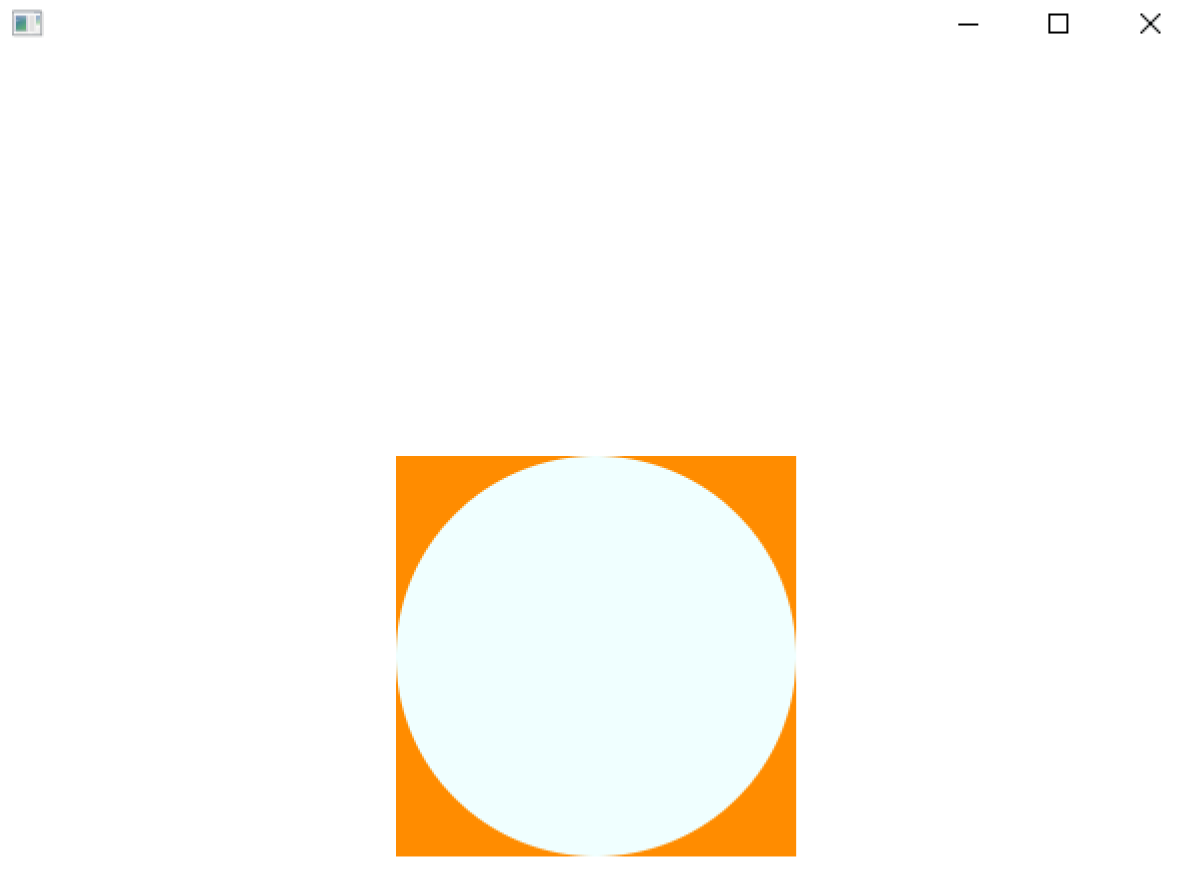
Bounding rectangle for MyLine:



Bounding rectangle for MyArc:

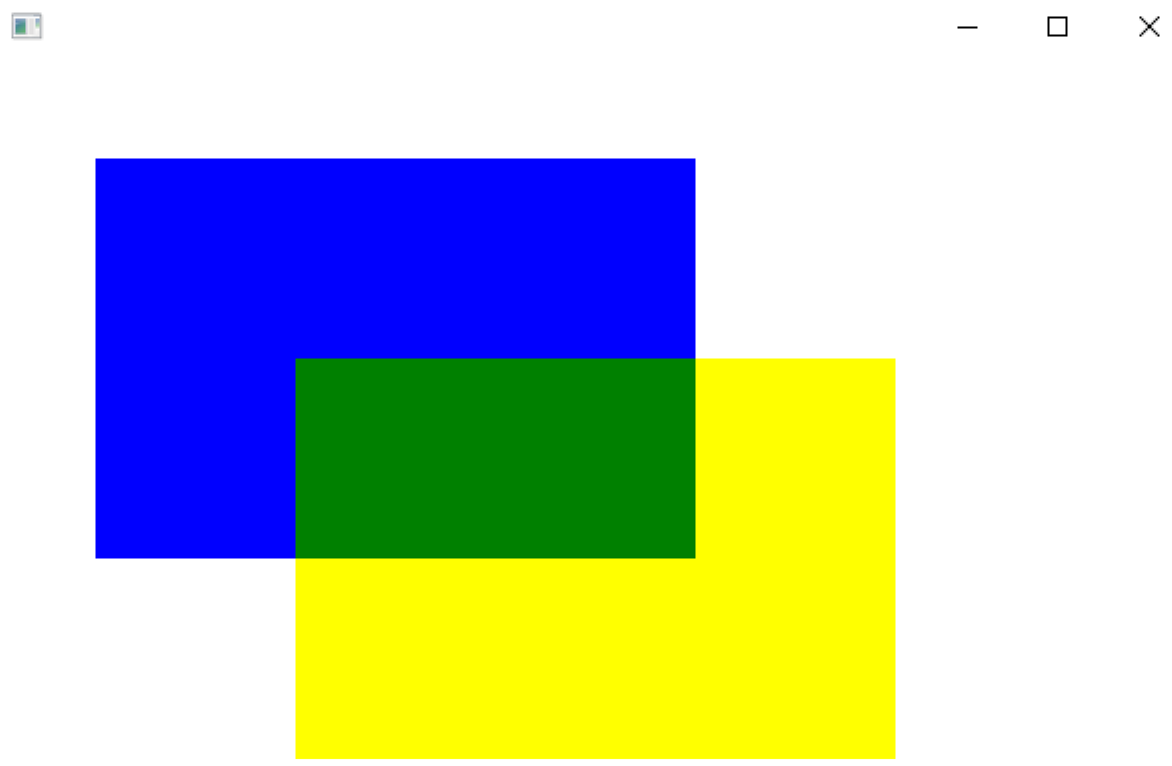


Bounding rectangle for MyCircle:

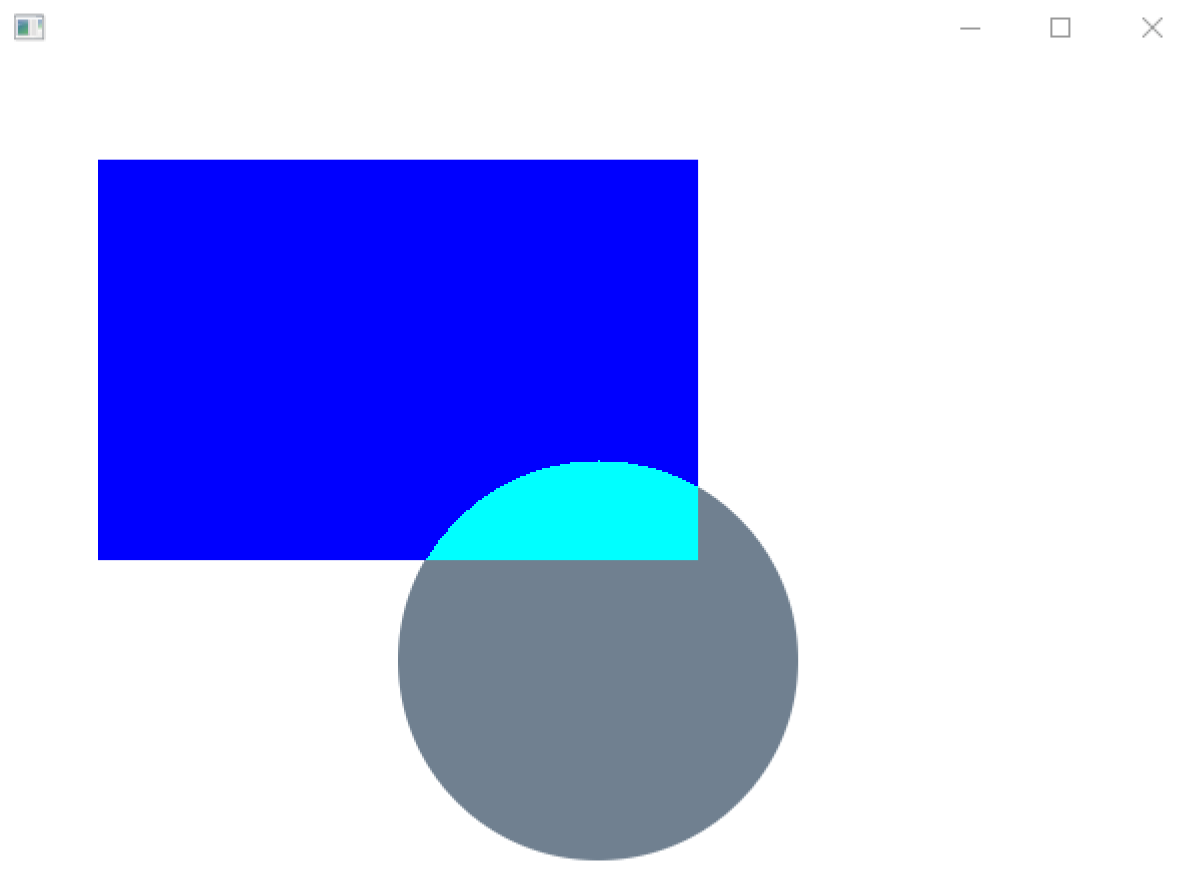


The final part of this project was to draw the area of intersection of two MyRectangle objects and that of a MyRectangle object and a MyCircle object. Once again, the shapes were arbitrarily chosen by me to be as I wished, the only requirement was that the shapes had to touch, which is why the rectangles in the rectangle intersection solution are both of the same dimensions, but just have different TLCP MyPoint instances. The same goes for the intersection of the MyRectangle and MyCircle objects. I just had to ensure that the objects intersected to show that the intersection drawn out (which is why the TLCP of the rectangle and the center of the circle are somewhat close to each other – and why the circle's radius is 100 – to ensure some overlapping). Here are the outcomes of the solution:

Intersection of two MyRectangle objects:



Intersection of MyRectangle and MyCircle objects:



Part 4

Introduction

This part of the project simply tells us to explicitly specify all classes imported and used in our code.

Solution

The ArrayList class imported so that ArrayLists could be used for the intersectMyShapes and drawIntersectMyShapes method in the MyShapeInterface (used to store MyPoint instances that resided in both shapes).

The Optional class was used to use the Optional.ofNullable method and the .orElse method when setting values of MyColor in instances to ensure that the variable ‘color’ in all instances were being properly set to real enum MyColor constants.

The JavaFX Application class was imported because it is the class from which the JavaFX can produce a stage and scene, as well as launch JavaFX in the first place. It also explains the “extends” word for the public class App at the run of the program.

The Group class of JavaFX was imported because it was used to make a Group instance in Part 2 of this project, which allowed for all of the instances of shapes to be added together by using the getChildren method.

The Scene class of JavaFX was imported because it set the Scene for the canvas to be on in the first place in Part 2 of the project. The scene holds all content in the scene graph and essentially holds all the parts of the JavaFX application.

The Canvas class of JavaFX was imported because that is where the shape instances were all drawn out. The Canvas class allows for drawing to happen on the JavaFX scene.

The GraphicsContext class of JavaFX was imported because it worked directly with the Canvas class and provided the program with information about the canvas so that Part 2 would end up working smoothly. Additionally, it allowed the shapes to be drawn out on the canvas with the use of methods setFill, fillOval, fillRect, setStroke, setLineWidth, and strokeLine.

The Color class of JavaFX was imported because it had to work in accordance with the MyColor class I created so that setFill and setStroke would have proper JavaFX colors (which was retrieved through the use of the Get_JavaFXColor of the MyColor class).

The ArcType class of JavaFX was imported because it was needed for the fillArc method in the draw method of the MyArc class (the method needed to know the way the ArcType was to be drawn and this class provides the constants for it).

The Stage class of JavaFX was imported because it provides the main platform to show the results of Part 2 of this problem (https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html). It also was used to use the method setScene to bring up the Scene instance 'scene' on the Stage in the outcome of Part 2. Also the show method was used from the Stage class to reveal the geometric configuration finally produced.

NOTE: Although not directly imported, I had used the Math class of Java (which is in the java.lang package and does not need to be imported - https://www.knowprogram.com/java/import-math-class-java/) for methods pow, sqrt, constant PI, tan, atan, and atan2.