JavaFX Project Part 1

Cristian Statescu

Part 1

Introduction

To start off, the first problem of this homework assignment was to make a class hierarchy where the parent class is a class named MyShape (which extends the Java class Object, which every class has as a super class according to Oracle Docs), and then three sub classes, named MyLine, MyRectangle, and MyOval. Along with this hierarchy, we were to create a class MyPoint which was to be used by all classes mentioned previously, and an enum class named MyColor which all classes in the hierarchy and the class MyPoint would use. The point of making MyColor, MyPoint, and the MyShape class hierarchy was to use these classes to draw out a pattern with a sequence of alternating concentric ovals and inscribed rectangles (to be discussed further in the report) 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 2**.

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 2** 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 Class “MyShape”

The Class Variables

The 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 (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). 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 two methods which are to be overridden by all the subclasses in the hierarchy, those being the perimeter and area methods. In MyShape, they simply return 0, because there is no actual geometric shape to return the perimeter and areas of. In the subclasses of this hierarchy, these methods also exist and will get overridden to make sure they work for the subclass they are in and so that it will also be possible to name the methods the same name without worrying about MyShape’s method. The final two methods are also going to be overridden in the subclasses, and those methods are draw and toString. The method draw takes in a GraphicsContext parameter and uses it to help color the JavaFX canvas the same color as the ‘color’ variable. This is done by again using the getFill method to set the color of what will be drawn on the canvas the same way it was done with the MyPoint draw method – by doing “this.Get_MyColor().GetJavaFXColor()”. Then, lastly, the fillRect function is made with GC.fillRect(0, 0, GC.getCanvas().getWidth(), GC.getCanvas().getHeight());, which ensures that from the origin (0,0), the rectangle the size of the canvas – really just the canvas itself – (given the use of the GraphicsContext Canvas methods getWidth and getHeight to get the size of the canvas for the final line of code of the draw method) will be filled with the color received from the first line of this method mentioned Lastly, the toString method will be overridden in all subclasses, and has the purpose of returning a string with information on the instance. This method for MyShape returns the string value of the reference point using the MyPoint.StringPt(excluding the color because there is no reason to get out a string of the color when the general idea of toString will be to use the get information of instances to perform mathematical operations on them on paper so that you can draw out your desired shapes on the JavaFX canvas)

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 of the MyShape super class. 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 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%20th eta%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 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 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 final method draw is an overridden (from the 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 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 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 super class 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 superclass MyShape. The Get methods return the values of the variable their name is associated with (ex. Get_Height(){ return this.h;})

Lastly, the 4 remaining methods are all methods which are overridden because they exist in the superclass MyShape. 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 4 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 4 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 and final method 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 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 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 4 methods are all overridden, as methods of the same name exist in the MyShape class. The first of these remaining 4 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).

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

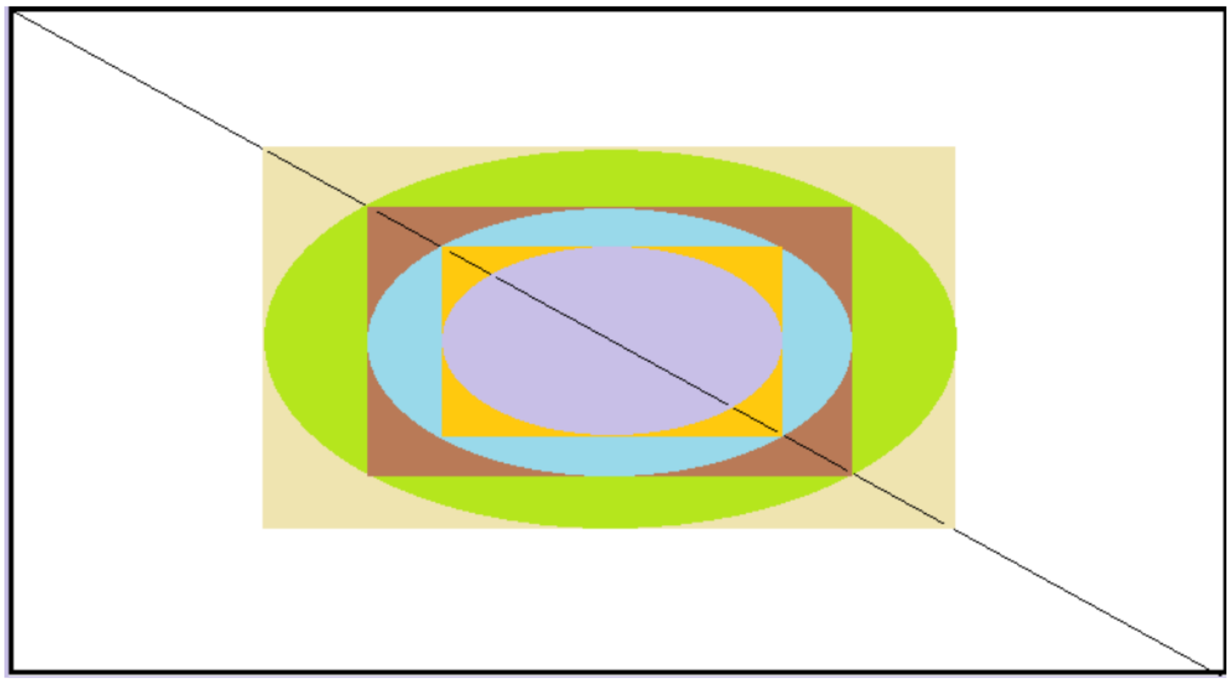
Note that a is the semi-major axis, and b is the semi minor axis, which were gotten through the use of the Get_SemiMajorAxis and Get_SemiMinorAxis methods of MyOval. Using the Math library (for the constant pi and sqrt method), I wrote out the equation into the perimeter method, and the result of this mathematical equation is what would come out. The second of the last 4 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 4 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 final of the 4 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.

Part 2

Introduction

Using the just made classes, we were to make a geometric configuration that is made up of a sequence of concentric ovals and inscribed rectangles on the JavaFX canvas. The code is to maintain this pattern no matter the size of the canvas' proportions, the drawing must be proportional to the dimensions of the canvas, and the ovals and rectangles are to be filled with whatever color we desired (specified using a constant from the enum MyColor). Presented is the general pattern desired to be made by our code:



Solution

Firstly, a Group instance of the JavaFX class Scene was created to keep the whole code organized and allow the result of what was drawn on the canvas to be shown at the end of the code being run. Next, a Scene instance named 'scene' was created, which has the 'root' Root instance we just created passed into it, along with 2 values that represent the height and width of the new scene (the drawings that will pop-up when the whole JavaFX outcome is to be shown on the scene). A Stage instance is then created (named “stage”) to have a window pop-up and show the results of the JavaFX program we are making (using the show method of Stage). Next, a Canvas instance named ‘canvas’ is created (this is where everything is to be drawn) and made a certain size using 2 numbers, again, representing the height and width of the canvas (these values can be anything, the geometric configuration to be drawn out by the code should still hold). Lastly for the setup, a GraphicsContext instance is made named ‘GC’, which gets the context of canvas using the getGraphicsContext2D method (‘GC’ will be used by the compiler and our code to know exactly how to plot all of the shapes out from our code onto the canvas and to get information of the canvas).

Now to get all of the shapes made for the problem. Firstly, I created a MyPoint instance of the center of the canvas using the getCanvas and getWidth/Height methods, and then divided what those methods returned to get the point in the middle of the canvas. The reason this is useful is because now all ellipses made for the problem have a center defined for all general sizes of a canvas.

Next, noticing the outer box of the geometric configuration and noticing that while it had the border outlined but the center not filled in with anything but while, I noticed that the outer box was actually defined as 4 lines with the MyColor.BLACK value that made up a rectangle on the outside. Hence, 4 MyPoint instances were created to signify each corner of the soon to be made rectangle (that was not an instance of the MyRectangle class). Notice in the code that things such as “center.Get_x() - GC.getCanvas().getWidth()/3;” are passed as the ‘x’ coordinates of these points and “center.Get_y() - GC.getCanvas().getHeight()/4;”. Firstly, the reason the center point is used is so that the drawing will stay uniform and centered. Secondly, the reason that the width of the canvas divided by 3 is being subtracted from the ‘x’ value of the canvas and being passed into MyPoint instances as the ‘x’ value is because I wish to make a rectangle for a square sized canvas, and so, dividing the canvas width by 3 and subtracting it from the ‘x’ value of the center point of the canvas will give me a long width since I am offsetting from the center by more (because I am subtracting). For ‘y’ values, I am diving the size of the height of the canvas by 4 and offsetting each point by that value because the less the point is being offset from the center value, the smaller the length will be (since the points will be closer to each other). Why I chose 3 and 4 to divide by is simply a matter of trial and error. This code will work for all sizes of a canvas and, while the outcome won’t always be rectangles and can sometimes be squares, it provides a code for the general case of a canvas. Notice also to keep the points in a neat manner to ensure lines can be formed, all instances are created using the same general inputs, but just changing whether to offput by adding or subtracting depending on whether it is the TLCP (subtracting to offset for the ‘x’ and subtracting to offset for the ‘y’) or the BRCP (adding to offset ‘x’ and adding to offset ‘y’), or the other two points, it is decided based upon how the x and y axis’ work on the canvas (x increases going from left to right, y increases going downwards).

Now, 4 MyLine instances are created (named outerrect*1-4*) using the 4 points created to make the outside rectangle. This was done by simple connecting all appropriate points and using the MyLine parametrized constructor that didn’t take in a MyColor enum constant (since black would be an appropriate color for the outside box). Additionally, notice how there is a diagonal line going from the top left corner point to the bottom right corner point of the rectangle. A line was also created using the respective points.

Now to create the first inner rectangle I simply took the way I made the outer top left corner point and offset the values of the center point by the height or width of the canvas divided by the number 3 * 1.5 for the width, and 4 * 1.5 for the height. The reason for this number (1.5) is to keep a constant ratio so the rectangles are proportional to each other and so that the diagonal line created goes through the top left corner point and bottom right corner points of the rectangle. Notice that the length of width of the rectangle is the length of the outerrect2 length divided by 1.5 (again to keep proportions). The MyColor choice for the constructor was simply because it looked most similar to what was in the example given.

Creating the first encapsulated ellipse was simple. I passed the same width as the rectangle just created (inner1) as well as the same height, and then input the center of the canvas as the center of the inner0 ellipse (so that it would stay centered and be encapsulated properly by the rectangle inner1).

Creating the first rectangle to be encapsulated by an ellipse was more difficult. I had to notice that the ellipse formula was (x/a)^2 + (y/b)^2 = 1 (where a and b are the semi minor and major axis). Additionally, I noticed that the ratio of ‘a’ and ‘b’ will have to be the same as the encapsulated rectangle’s width and height (‘w’ and ‘h’). Then, knowing these two equations allows once to solve a system of equations which ends up showing that the width of the encapsulated rectangle is a * sqrt(2) (a being half the width of the ellipse) (Math library will be used), and for the height it is the same but in this form: b * sqrt(2) (b being half the height of the ellipse). Thankfully with this math done, everything was made a lot easier and it was simply a repition of checking values from previously collected shapes. As you can see in inner2’s constructor, the height and the width are based on the last shape’s (inner1o (oval)) height and width and are plugged into the equations a * sqrt(2) and b * sqrt(2). The values are then offset (by subtraction) from the center point to give the rectangle the proper top left corner point. Note that Get_Height is divided by 2 so that the point doesn’t overshoot, the center will only be half of the height of the ellipse away on the y-axis from where the top left corner point should be, and the same goes for the x-values. For the rest of the problem, this is repeated. MyOval instances are created based upon the values of the height and width of the last rectangle created (with the center always being the center of the canvas to keep the configuration neat), and then the MyRectangle instances are created based upon the value of the last oval created with the offset from the center to the top left corner point of the rectangle being decided by the formula (a or b) * sqrt(2), and having a height and width based upon the height/width of the oval divided by 2 (a or b, since that’s what they stand for in the mathematical formula) and multiplied by square root of 2.

Finally, all the shape instances (minus the MyPoint instances, since they are just used to make other shapes that go over the points either way) are drawn on the canvas using the draw method of each of their resective classes. The, the stage has the setScene method used on it to set up the Scene object ‘scene’, the root object has the getChildren method and then uses the add method to add all the nodes of the canvas that were created (all of the instances created). Lastly, to show the outcome of the code, the line stage.show(); is to be executed at the end of the program, which pops up the scene in a window (with the canvas and all) that shows the geometric configuration.

<u>Part 3</u>

Introduction

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

Solution

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 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, and the constant PI.