AP Computer Science A Name_____
Lab 07B date _____ / _____

As programmers write software they like to reuse code as often as possible. Often you can predict that several different (but related) classes will contain a fair amount of identical code but will also have some differences. For example, suppose you are writing a painting program that uses a variety of shapes on a graphics screen. Three of the many possible shape classes ar e shown below:

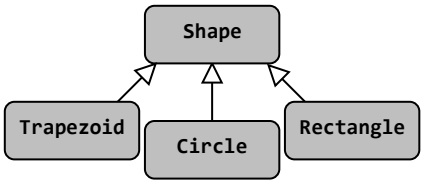
Circle	**Rectangle**	**Trapezoid**
x, y, r, color	*x, y, w, h, color*	*x, y, b_1, b_2, h, color*
public methods	public methods	public methods

`int getX()`
`int getY()`
`Color getColor()`
`void setColor(Color newC)`
`String getName()`
`double getArea()`
`void moveTo(int newX, int newY)`
`void drawOnto(Graphics g)`
	`int getX()`
	`int getY()`
	`Color getColor()`
	`void setColor(Color newC)`
	`String getName()`
	`double getArea()`
	`void moveTo(int newX, int newY)`
	`void drawOnto(Graphics g)`
		`int getX()`
		`int getY()`
		`Color getColor()`
		`void setColor(Color newC)`
		`String getName()`
		`double getArea()`
		`void moveTo(int newX, int newY)`
		`void drawOnto(Graphics g)`

List here all the private instance variables that you think will be common to all three classes:

Circle the **five** public methods that will likely have ***the exact same*** implementation (code) for each of the three classes shown above:

`getX()`	`getY()`	`getColor()`	`setColor(Color newC)`
`getName()`	`getArea()`	`moveTo(int newX, int newY)`	`drawOnto(Graphics g)`



One possible solution to this situation is to write a **`Shape`** class that will be the superclass of the **`Circle`**, **`Rectangle`**, and **`Trapezoid`** classes (and possibly other future shape classes). The **`Shape`** class will contain the instance variables that are common to all shapes, and also contain the methods that all shapes would need. However, there's a small problem: how can the **`Shape`** class have the correct code for a method that will be different for each shape? For example, the **`getArea()`** method will be different for each subclass of shape, so how could the **`Shape`** class include code for this method? One option is for the **`getArea()`** method in the **`Shape`** class to **`return 0.0;`** and to allow the subclasses to override this method to return the correct area. Note: a better option is to use abstract classes, but that is a topic which we do not discuss in this course.

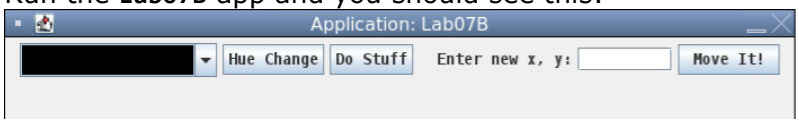
The **`Shape`** class will have these features:

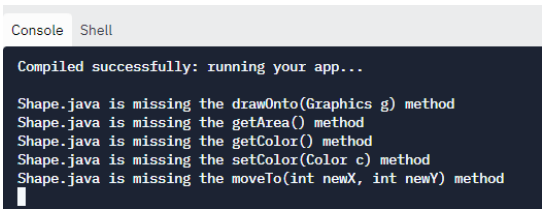
(1) **`Shape`** includes the private instance variables that are shared by all future subclasses of **`Shape`**.

(2) **`Shape`** "factors out" the duplicate method code to be written **<u>once</u>** in the most generic **`Shape`** class.

(3) **`Shape`** provides "placeholder" methods (**`getArea()`** and **`getName()`**) that are to be present in all subclasses of **`Shape`**, but will still need to be overridden by the individual subclasses.

1. Run the **Lab07B** app and you should see this:





2. In the **Shape.java** class declare all the instance variables you wrote down on the first page of this lab.

Make sure the instance variables are (and stay!) `private`

3. Uncomment lines 10 and 13 in **Lab07B.java** and observe how the client wants to instantiate a **Shape**:

```
 9     // x = 10, y = 60
10     menu.add( new Shape(10, 60, Color.ORANGE) );
11
12     // x = 10, y = 80
13     menu.add( new Shape(10, 80, Color.RED) );
```

   In **Shape.java** write a **Shape** constructor that will be compatible with the client's code shown above.

4. In the **Shape** class write the five methods that will have the same implementation (code) for all future shapes (these are the methods **getX()**, **getY()**, **getColor()**, **setColor(Color)**, and **moveTo(int, int)** that you circled on the first page of this lab). Although future subclasses are free to override these methods, they probably won't need to.

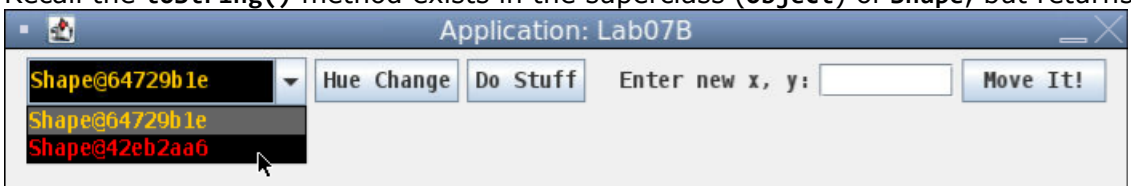
5. The methods **getName()** and **getArea()** will be different for each shape. Therefore, these methods are ***placeholders*** in **Shape.java** that will be revised in the subclasses (that you will write in steps 8 – 12).

   Add these two methods to the **Shape** class. Every future subclass of **Shape** will need to **@Override** these two methods in a way that will make sense for that shape.

```
public String getName()
{
  return "Shape";
}

public double getArea()
{
  return 0.0;
}
```

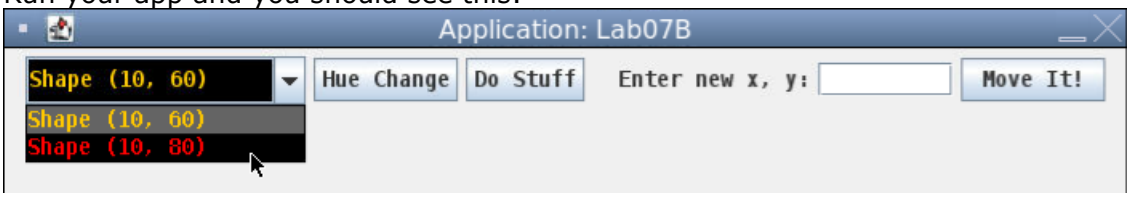
6. Recall the **toString()** method exists in the superclass (**Object**) of **Shape**, but returns a garbage value:



   Override the **toString()** method by returning **this.getName()+" ("+this.getX()+","+this.getY()+")";**.

   Although the **getName()** method of the shape class returns **"Shape"**, at runtime this method will return the name of the actual ***run time*** shape because the future subclasses will override the **getName()** method.

Run your app and you should see this:



Pressing the ***Hue Change*** button will allow you to change the selected shape's color (the code that I wrote for that button calls the **`setColor(Color newC)`** that you wrote in the **`Shape`** class).

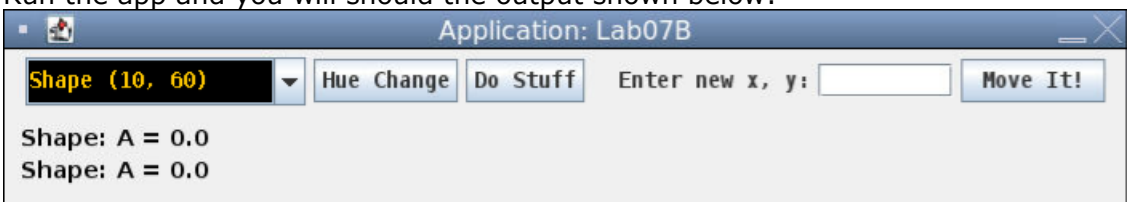
If you enter two integers separated with a comma (or a space), pressing the ***Move It!*** button will allow you to change the (*x*, *y*) of the selected shape (the code that I wrote for that button calls the **`moveTo`** method you wrote in step#4). Note: the ***Do Stuff*** button doesn't do anything until you complete step #13 of this lab.

7. The **`Shape`** class will contain a *partially written* **`drawOnto`** method, and every future subclass that you write will (1) use its built-in functionality, and (2) override it to include new functionality.

   Add this code to the **`Shape`** class:

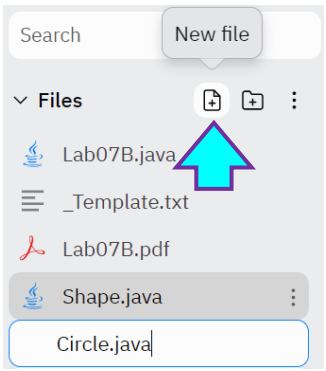
```
public void drawOnto(Graphics g)
{
  g.setColor(Color.BLACK);
  g.setFont(new Font(Font.SANS_SERIF, Font.BOLD, 14));
  g.drawString(this.getName()+": A = "+this.getArea(), this.getX(), this.getY());
}
```

   Run the app and you will should the output shown below:



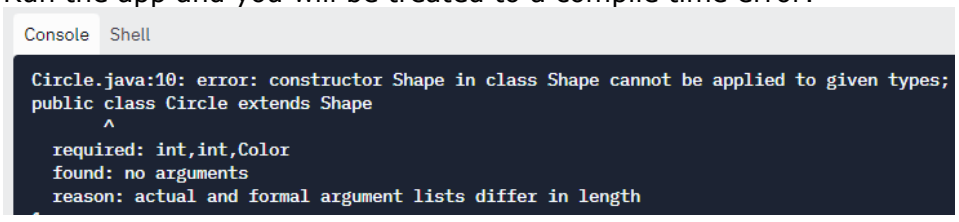
8. You are now ready to begin writing the **`Circle`** class, which is a subclass of the **`Shape`** class.

   In the **Files** menu on the left side of your screen press the **New File** button and type **`Circle.java`**.



   Open the **`_Template.txt`** file in your **`replit`**, copy all the code, and paste it into **`Circle.java`**. Be sure to change all **`XYZ`** to **`Circle`**.

   Run the app and you will be treated to a compile time error:



   This error occurs *because you did not yet write a constructor* for **`Circle.java`**, so *Java* will attempt to provide you with a (hidden and not very good) ***default*** **`Circle`** constructor which also includes a hidden call to **`super();`** (see next page).

Java compiler incorrectly attempted to give you this...

`public Circle()`
`{`
`  super();`
`}`

… but the super class (**Shape**) does **not** have a constructor with zero explicit parameters!

If the superclass did have a zero explicit parameter constructor, then **`super();`** would work. However, the **Shape** superclass does ***not*** have a zero explicit parameter constructor, so you get a compile error.

Uncomment line 16 in **Lab07B.java** to see how the client code would like to instantiate circles:

```
15      // x = 500, y = 300, r = 100
16      menu.add( new Circle(500, 300, 100, Color.CYAN) );
```

Add this constructor to your **Circle** class:

```
public Circle(int x, int y, int r, Color c)
{
  super(x, y, c);  // Calls your superclass (Shape) constructor
  // Now add more code specific to the Circle constructor...
}
```

A **Circle** may declare its own instance variables, but it should ***NOT!*** duplicate / shadow the (*x*, *y*) or *color* instance variables contained the **Shape** class.

9. In this step you will complete the **Circle** class:

   A) You should not override the **getX()**, **getY()**, and **getColor()** methods (i.e.: do not write **public int getX()**, etc.). The **Circle** class automatically *inherits* them as they were written in the **Shape** class.

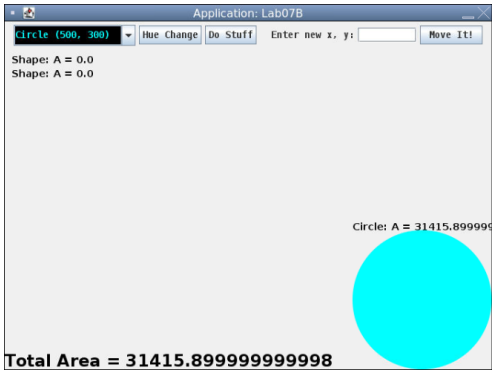
   B) Override and provide the correct implementations for the **getName()** and **getArea()** methods (I used 3.14159 for π).

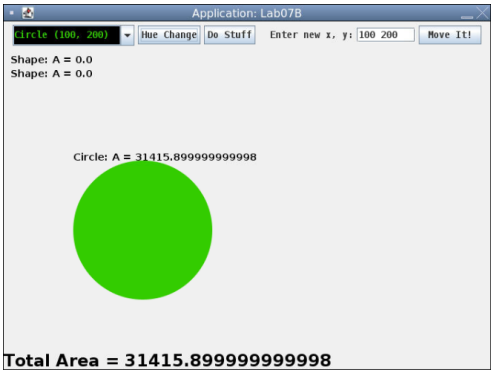
   C) Finally, the **drawOnto(Graphics g)** method is a hybrid because you will inherit its default behavior (written in the **Shape** class) and then write the additional code that is specific to circles.

```
@Override
public void drawOnto(Graphics g)
{
  super.drawOnto(g);  // Calls your superclass drawOnto method...
  // You will need to use g.fillOval(x, y, w, h) where w & h are diameters
}
```

   If you accidentally left off the **super.** here, the compiler would interpret it as **this.drawOnto(g)** which would lead to "infinite" recursion ☹ at run time and your app crashing with a stack overflow.

Test your circle class by running the app. Take note of the size and location of the circle in the first screenshot. You should also be able to change the color of the circle and its location.





10. Write the **`Rectangle`** class. Then uncomment lines 19 and 22 of **`Lab07B.java`** and run the app.

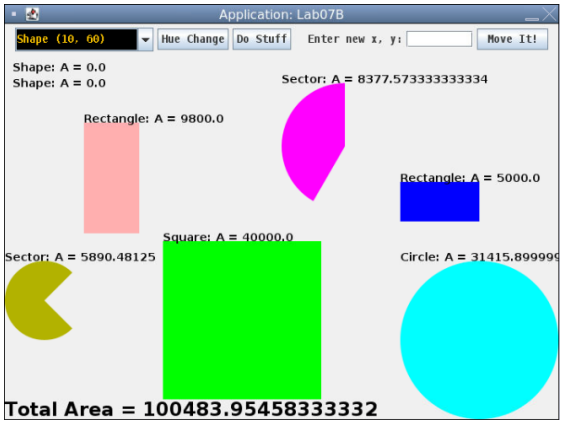
11. Write the **`Square`** class so that it extends **`Rectangle`** (note that you are **not** directly extending **`Shape`** here). Do this with the *least amount of code possible*. Test it by uncommenting line 25 of **`Lab07B.java`** and running the app.

12. Write the **`Sector`** class. In the **`drawOnto`** method you will use

    `g.fillArc(x, y, w, h, startAngle, shadeDegreesCCW)`

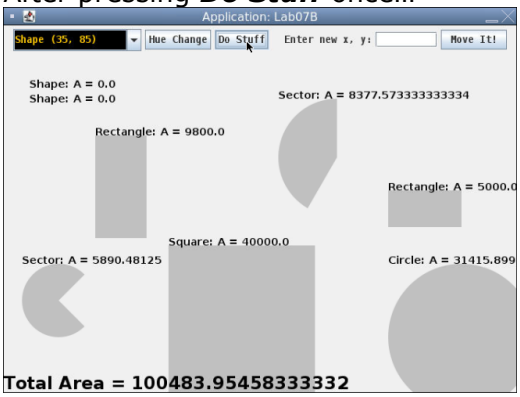
    Look up the formula for the *area of a sector* if you forgot it.

    Uncomment lines 28 and 31 of **`Lab07B.java`** and run the app.

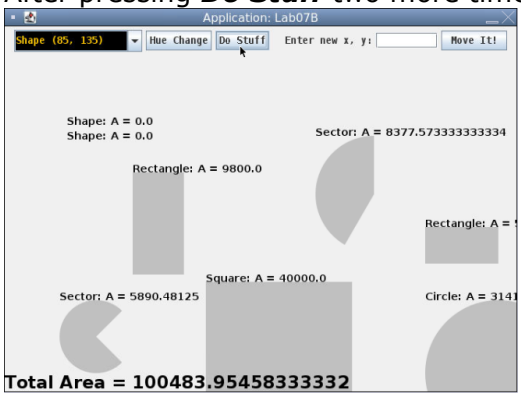


13. Complete the client method **`doStuff(Shape something)`** in **`Lab07B.java`**. This method changes the color of **`something`** to **`Color.LIGHT_GRAY`** and changes its position by 25 pixels to the right and 25 pixels down.

After pressing ***Do Stuff*** once…



After pressing ***Do Stuff*** two more times…



14. When you are finished go through this checklist before you submit your lab.

    _____ **`Shape`** instance variables private and no shadowing in subclasses

    _____ Subclasses call getter methods of their super class

    _____ **`Square`** extends **`Rectangle`**, minimal code

    _____ Button functionality indicates methods are correct

    _____ Area ≈ correct

    _____ Circles are the correct size

    _____ Sectors are the correct size and shape