AP Computer Science A Name_____
Lab 04A date _____ / _____

In this lab you will learn about how to **write a class** and how to **test a class**. The class you will be writing is a model for a tulip. At run time you will create several objects of the **`Tulip`** class, and you will modify these objects and cause them to interact with each other.

When you write a class you need to first think about what exactly you are modelling. There are two different things to think about: the state of an object, and the methods that operate on an object.

The **state** of the object	What data will each of the individual objects need to store about themselves? Will that data change over the life of the object? The data are called **instance variables**.
The **interface** of the object	Which methods will you need to write to make it possible for the objects to change over time and to interact with each other? The **public methods** are the interface.

The state of a Tulip:
Each **`Tulip`** object that you create will need to store data about itself: its position (*x, y*), and its color. Each of these three data will be stored in an instance variable, which means that each **`Tulip`** object has their own copies of these three instance variables.

The interface of a Tulip:
Below is a listing of the methods that allow other java files to interact with **`Tulip`** objects. You will write all the methods in this assignment. The last column indicates the step in which you will write that method.

Return Type	Method	Description	Step #
`---`	`Tulip(int x, int y, Color c)`	Constructs a **`Tulip`** object with the indicated (*x*, *y*) location and color	**3**
`String`	`toString()`	Returns a string in the format **`"(x, y)"`**	**4**
`int`	`getX()`	Returns the *x*-value of the tulip	**5**
`int`	`getY()`	Returns the *y*-value of the tulip	**5**
`Color`	`getColor()`	Returns the color of the tulip	**5**
`void`	`moveTo(int x, int y)`	Changes the tulip's location to the indicated (*x*, *y*)	**6**
`Image`	`getImage()`	Returns a simple drawn image of the tulip	**7**
`Tulip`	`clone(int x, int y)`	Creates and returns a new tulip that has the same color as the implicit parameter tulip at the indicated (*x*, *y*) value	**8**
`Tulip`	`combine(Tulip t)`	Creates and returns a new tulip that has a color that is the arithmetic average of the implicit and explicit tulips, and a location that is the arithmetic average of the implicit and explicit tulips	**9**

1. Open the **`Lab04A`** project in the Teams section of **`replit`**.

   When you press the ▸Run button, you should see the app shown at right.

2. You will write the code that ***defines*** all tulips in the **Tulip.java** file. You should see this in **Tulip.java**:

```
public class Tulip extends Object
{

}
```

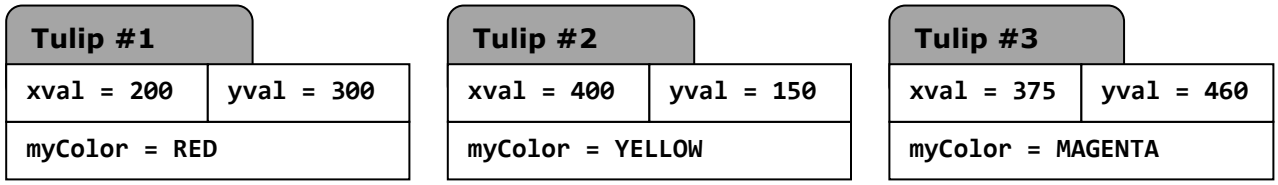
Each class that you write automatically extends (also: *inherits from*) the **Object** class. This means that you get certain methods for "free" because those methods already exist in the **Object** class.

The first step is to decide what data each tulip needs to store and retain over time. The choice you make about this step may have implications for how you write the **Tulip** methods.

We need to store three things: the *x* and *y* values of the tulip, and its color. Do this now by declaring three ***instance variables*** inside the **Tulip** class:

```
private int xval;
private int yval;
private Color myColor;
```

Each **Tulip** object created *in the future* gets its own copies of the ***instance variables*** listed above. In general, each tulip's variables will have different values. For example:



The instance variables are declared **private** to prevent other classes from accessing them and changing them in an unauthorized way.

3. In this step you will write a ***constructor*** for the **Tulip** class. The purpose of a constructor is to give the instance variables (see step #2) their initial values.

In the future, a ***client*** might be writing code in a class called **Picture.java**, and they may want to have a red tulip located at (50, 70). In their **Picture.java** they would write code such as this:

```
Tulip t1 = new Tulip(50, 70, Color.red);
```

They *have to* do it this way because you (today) wrote the constructor in **Tulip.java** like this:

```
public Tulip( int x, int y, Color c )
{
  xval = x;

  //etc...

}
```

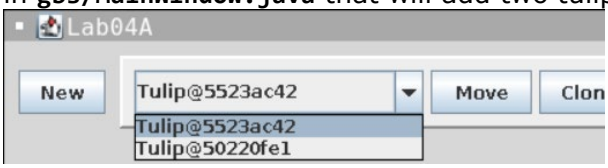
The constructor's explicit parameter variables (**x**, **y**, **c**) temporarily store the values specified by the client code (e.g.: 50, 70, and red). The **x**, **y**, and **c** are local variables (and short-lived), so you must transfer their values to the instance variables (**xval**, **yval**, **myColor**) within the constructor body.

The code in the constructor should initialize the ***instance variables*** to be equal to the corresponding explicit parameters of the constructor. The explicit values of the constructor are specified by the clients of the **Tulip** class (in the future).

Note that the constructor method has no return type (not **public void Tulip** or **public int Tulip**- just **public Tulip**) and that the method name **Tulip** exactly matches the name of the class. This is the way the compiler knows that you are writing a constructor (as opposed to just some other method).

When you run the app you should now see two tulips added to the first drop-down menu (I wrote code

in **gbs/MainWindow.java** that will add two tulips to the menu when the **Tulip** constructor is complete).

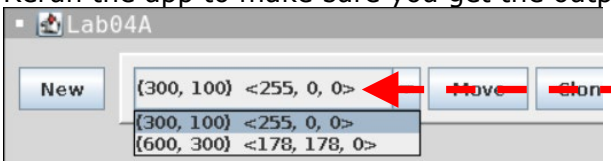


4. The drop-down menu code (written in 1998) contains a call to the tulip's **toString()** method to get a text version of each tulip. Even though you did not yet write a **toString()** method, it is one of the methods you get "for free" (inherit) from the **Object** class. The **toString()** method that the **Object** class provides is not very good: we will write our own version that returns a string in the format (*x*, *y*) <R, G, B>.

   Write the **toString()** method in the **Tulip** class. The **@Override** shown below is optional, but it is a hint to the compiler that you are attempting to "fix" the less than helpful **toString()** method you inherited "for free" from the **Object** class.

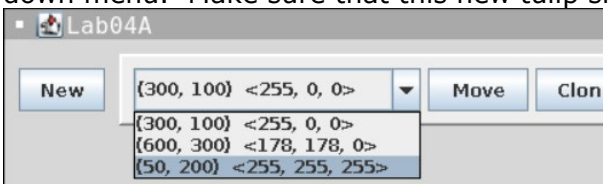
```
@Override
public String toString()
{
  String s = "(";
  //You finish this...
  return s;
}
```

   Rerun the app to make sure you get the output shown below.



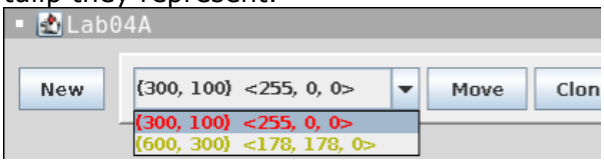
The final three numbers are the RGB values of the color of the tulip. You can get these values with **myColor.getRed()**, etc.

   Now press the **New** button and create a third tulip at (50, 200) with any color. The code that I wrote in **gbs/MainWindow.java** will instantiate a tulip with your specified values and then add it to the drop-down menu. Make sure that this new tulip shows up in the drop-down menu.



5. In this step you should write the **getX()**, **getY()**, and **getColor()** methods in the **Tulip** class. These methods simply return the corresponding instance variable.

   When you re-run the app, you should see that the drop-down menu items are now in the colors of the tulip they represent.

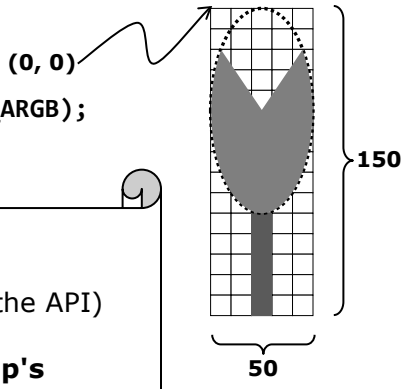


6. Write the **moveTo(int x, int y)** method in the **Tulip** class. This method simply changes the **xval** and **yval** instance variables to be equal to the explicit parameters.

   Run the app and press the **Move** button and enter new coordinates for one of the tulips. Check to make sure the selected tulip in the drop-down menu changes to reflect your new coordinates.

7. In this step you will write the **`getImage()`** method in the **`Tulip`** class. This method creates a simple picture of a tulip on an **`Image`** object and returns that image. The code that I started below first creates a 50 x 150 image, and then gets the "graphics" of the image so that you can draw on it. The method then returns that image.

```
public Image getImage()
{
  Image canvas = new BufferedImage(50, 150, BufferedImage.TYPE_INT_ARGB);
  Graphics g = canvas.getGraphics();
  g.setColor(Color.GREEN);
```



- Add code to here to draw the stem (use **`g.fillRect`**)
- change the color that **`g`** uses to that of the tulip
- Use **`g.fillArc`** for the flower (Google *java Graphics fillArc* for the API)

Note: all coordinates are relative to (0, 0), and not the tulip's (xval, yval)

```
  return canvas;
}
```

8. Suppose the code at right exists in the **`MainWindow.java`** file. In this code, **`b`** will be at (100, 200) with red petals. Note that the return value of the **`clone`** method is a **`Tulip`**.

```
Tulip a = new Tulip(50, 75, Color.RED);
Tulip b = a.clone(100, 200);
```

Write the **`clone(int x, int y)`** method in the **`Tulip`** class. Make sure that the **Clone** button works correctly: it will make a copy of the tulip that is selected in the first drop-down menu at a new location that you specify.

9. In this last step you will write the **`combine`** method in **`Tulip.java`**.

Suppose the code shown at right exists in the **`MainWindow.java`** file.

```
Tulip p = new Tulip(50, 75, Color.PINK);
Tulip c = new Tulip(90, 33, Color.CYAN);
Tulip z = p.combine(c);
```

In the third line of that code, **`z`** will have a location of $\left(\frac{50+90}{2}, \frac{75+33}{2}\right)$ and its color will be the arithmetic average of pink and cyan.

A) Add the **combine** method shown below to **Tulip.java**.

```
public Tulip combine(Tulip t)
{
  return null;  // You will replace this soon...
}
```

B) In this method you will need to access the instance variables of ***two*** Tulip objects. In the statement **p.combine(c)**, **p** is the *implicit* parameter and **c** is the *explicit* parameter.

In the *code* of the **clone** method you can access the implicit parameter variables as you have been (with **xval**, **yval**, and **myColor**). However, to access the explicit parameter variables you must use **t.xval**, **t.yval**, and **t.myColor**. In fact, some programmers use **this.xval** (instead of just **xval**) to emphasize that they are accessing the implicit parameter's instance variable.

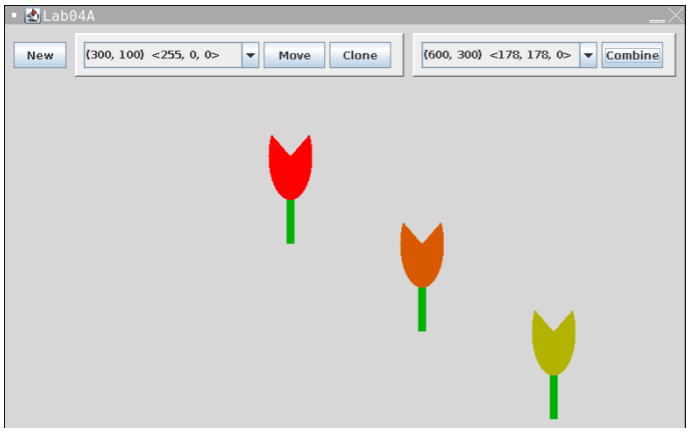
Add code like this **int x = (this.xVal + t.xVal)/2** to create the average x and average y values.

C) Calculate the average color with the arithmetic average of the reds, average the greens, and average the blues. Then create a new color like this: **Color mix = new Color(rAvg, gAvg, bAvg);**, where **rAvg** is the average of the two red components, etc.

D) Instantiate a tulip that has the explicit parameters that you calculated in steps (B) and (C).

E) Remove the **return null;** and replace it with code that will return the tulip you created in step (D).

F) Make sure that the **Combine** button works correctly. The **Combine** button combines the two tulips that you select at run time (from the two drop-down menus).



10. Once you have finished, click the ✓Submit button near the top right of your browser. This notifies me that you think your lab assignment is complete and ready for grading.