Lab 2 – Java GUIs

In this lab, you will create a basic drawing program using Java. This project will introduce you to some of the concepts and techniques for building GUIs in Java.

**Step 1 – Create your frame**

Most GUI systems make extensive use of Object Oriented Programming. There are classes and subclasses, and you use instance of these classes to construct and manipulate your GUI. The Java GUI classes are in packages called awt and swing. We will be using parts of both of these packages.

For starters, we need a Frame – an outer most container that all your other components will reside in. You can create your Frame as follows:

JFrame myFrame = new JFrame();

myFrame.setSize(600, 400);

myFrame.setTitle(“Phil Howard”);

myFrame.setVisible(true);

The functions above should be fairly obvious in what they do. There is another method that you should call prior to setVisible. This function tells Swing what to do when you close the window. For our application, we want the program to end when we close the window. We do that with the following call:

myFrame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

Run your program and experiment with different parameters to the methods to convince yourself that you know what the parameters do.

**Step 2: Create the drawing surface**

In general, JFrames are containers that hold JComponents. We would normally need a layout manager to control where the components go, but we are only going to use a single component, so we can skip the layout manager.

Create a new class that is a subclass of JComponent. You do that with the following (create a new class file):

public class DrawingArea extends JComponent

Since our DrawingArea extends JComponent, we can add our DrawingArea to our frame:

myFrame.add(new DrawingArea());

Go back to your DrawingArea class and add a paintComponent method. The Frame will call this method anytime it wants to draw the DrawingArea. The paintComponent method must have the following signature:

public void paintComponent(Graphics g)

The Graphics parameter specifies the surface that we get to draw on. It is easier to draw on a Graphics2D surface, so let’s create one from the Graphics object as follows:

Graphics2D g2 = (Graphics2D)g;

Now let’s draw a rectangle on our surface:

g2.setColor(java.awt.Color.GREEN);

Rectangle r = new Rectangle(10, 20, 30, 60);

g2.draw(r);

Now, if you run your program, you should see a green rectangle show up in the drawing area. Experiment with different parameters in the Rectangle constructor and with different colors to convince yourself that you know what the parameters do.

**Step 3 – add a menu**

We want to be able to draw different shapes in different colors. We will use a menu to select what we will draw. Create another class that extends JMenuBar. We will use the constructor of this class to create our menu. In Java, JMenuBars contain JMenus and JMenus contain JMenuItems. Let’s create our first menu and add it to the MenuBar:

JMenu menu;

menu = new JMenu(“Shape”);

add(menu);

Where is the add method defined? It is a member of the JMenuBar class. Since our class extends JMenuBar, the call to add is adding the menu to our menu bar.

We now need some items in the menu. You can add an item as follows:

JMenuItem menuItem;

menuItem = new JMenuItem(“Rectangle”);

menu.add(menuItem);

Add menu items for “Ellipse” and “Line” as well. Note: You can re-use the menuItem variable. You do not need a separate variable for each menuItem.

Now let’s add our menu to our Frame. Go back to the class that defines the frame and add the following before the call to setVisible():

myFrame.setJMenuBar(new DrawingMenu());

You may have chosen a different name for your menu class. If so, use the name of your class.

You should now be able to run your program and see the menu in the frame. Once you see your menu, add another menu for Color and add menu items for various colors (include at least three).

**Step 4: make the menus actually do something**

We have menus in our frame, and we can navigate through them, but they don’t actually do anything yet. To get them to do something, we need to attach an action listener to each menu item. An action listener is an object that “listens” for mouse (or keyboard) activity and responds to it.

You do not need to create new files for your action listeners. You can create the classes inside your menu class. You will need a class that implements the ActionListener interface. The interface requires a method with the following prototype:

public void actionPerformed(ActionEvent e)

For now, simply have your action listener print the parameter e.

Rather than creating a separate action listener class for each menu item, we could create an instance variable in our action listener and use the constructor to set the instance variable. Then, to attach the action listener, we could add the following immediately after the line the created the menu item (the following example is for the Rectangle menu item):

menuItem.addActionListener(new ShapeListener(“Rectangle”));

where ShapeListener is the name of the class you just defined. Now modify the actionPerformed method to also print out your instance variable so you know what menu item was selected. Add action listeners to each of your menu items and run your program. Each time you select a menu item, you should see output in the console window of Eclipse that indicates what menu item was selected.

Note: There are certainly other ways to accomplish this. If you find a different mechanism that your prefer, feel free to use it.

**Step 5: Drawing a shape**

To draw shapes based on mouse movement, we need to get the drawing area to listen to mouse events. You guessed it: another listener. There are a set of mouse events that we could listen for. In particular, we want to listen for mouse presses and releases. There is a class, MouseAdapter that has empty methods that listen for all mouse events. We can extend that class and override the methods for the events we are interested in. Just like our menu listeners were declared inside the menu class, our MouseAdapter can be declared inside our DrawingArea class. You will need to override the following methods:

public void mousePressed(MouseEvent e);

public void mouseReleased(MouseEvent e)

{

System.out.println(“Released at: “ +

e.getX() + “ “ + e.getY());

}

I gave you a code snippet giving a hint as to what you could do with the event variable.

You can add your mouse listener with the following line:

addMouseListener(new Mouse());

To finish this step, use the mouse events to draw a rectangle that goes from where you pressed the mouse to where you released the mouse. Here are several hints:

1. You will probably want an instance variable to store the rectangle.
2. You will want to do some error checking to handle degenerate cases (zero sized rectangles don’t show up very well on most screens).
3. You will want to make sure your rectangle has a positive size no matter what direction you move the mouse between the pressed and released locations.
4. After you’ve created your rectangle, call repaint() to make your rectangle actually show up.
5. At this point, you only need to show one rectangle at a time, and don’t worry about colors or other shapes yet.

Run this version of your program to make sure you can get a rectangle to show up and reappear every time you click, drag, and release the mouse.

**Step 6: Multiple shapes**

Instead of an instance variable that represents a single rectangle, create a list of rectangles. Every time you draw a new rectangle, add it to the list. Your paintComponent method should redraw everything in the list.

**Step 7: Connecting the menu to the drawing**

If you added a SetShape and SetColor method to your DrawingArea class, then the menu could call those methods with appropriate parameters to control what shape/color you draw. But the menu class needs to know what DrawingArea object to call those methods on. The easiest way to accomplish this is to add a DrawingArea member variable to your Menu class and pass a reference to the DrawingArea to the constructor of your Menu.

The next thing you have to do is create a Shape class that can be one of Rectangle, Ellipse, or Line. Your list of objects in your DrawingArea needs to be a list of Shapes. Finally, when you create a new shape in response to mouse activity, create the shape and color based on the last calls to SetShape and SetColor. The class to use for ellipses is Ellipse2D.Double. The class for lines is Line2D.Double. You might want to experiment with the g2.fill method instead of g2.draw.

**Wrapping up**

Your completed project should be able to display any number of rectangles, ellipses, and lines. You should be able to create shapes in at least three different colors. The shapes should “remember” the color they were drawn in.

Some things to check: If you minimize your window and re-expand it, your drawing should still be intact. If you cover your drawing window with another window and then uncover it, your drawing should still be intact.

**When you are satisfied with the behavior of your project, be sure to javadoc everything that is public in your classes.** Then create a JAR file that includes your source and submit that JAR file to Blackboard.