CS 2300 Lab MVC GUI – Part 4 Week 11

Color, Solid, and Task Panels

1. Do some preliminary reading on topics for today's lab on `ActionListeners`, `JRadioButton`s, `ButtonGroups` and the `JColorChooser` component.

2. Continue the project from the last lab. This should closely parallel the classroom development.

Next up, we will put in an interesting control panel.

3. Go to the `PnlColor` class. We need a field `JLabel jlColor` with the text `Set Color:` on it. Create this object with the following statement (above the c-tor):

```
private final JLabel jlColor = new JLabel("Color:");
```

Fix imports. (Do this whenever needed).

4. Also create a field `JButton jbColor` with the text `Change` on it, like this:

```
private final JButton jbColor = new JButton("Change");
```

We will learn how to make buttons do whatever we want with an **action listener**.

Also create a `jlSpacer` label as in the previous class, but put about 8 or 10 spaces in this one.

5. There is already a c-tor for the `PnlColor` class. We will add some stuff to it.

Now for the fun stuff!

6. A key part of GUI programming is using **ActionListener**s. Prior to Java 8, this required a lot of "boilerplate code" (code that is almost exactly the same each time). Java 8 has added **functional programming** concepts. The key functional programming idea is a **lambda expression**, which makes writing action listeners a lot easier.

7. We write an **action listener** to define what will happen when the `jbColor` button gets clicked. An action listener listens for an event (say, a mouse click) to happen, and then deals with the event. This is done with the `addActionListener` method. The "lambda expression" way of doing this, as shown next, makes it easy. Inside the c-tor, add an `ActionListener` to `jbColor` button, like this:

```
 jbColor.addActionListener(ae -> changeColor());
```

This basically says: If anything "fires" this control (i.e., someone clicks the button in this case), do whatever is in the `changeColor` method (to be written soon).

When you write the above line of code, you will get an error, because we have not yet created the `changeColor` method. We will get to that shortly. NetBeans will offer to translate this "new style" statement with an "old style" version. **NOT a good idea!** The new style is much better!

Next we will write the `changeColor` method to deal with changing color.

8.1. Below the c-tor, write a method with signature

```
private void changeColor(){}.
```

8.2. This method has only two lines. Before we write the first one (which is long and involved), we will use a temporary simple version to check that things are OK, and then upgrade to the full method. (Always do one little thing at a time!).

Write the two lines:

```
model.setColor(Color.GREEN);

model.getView().repaint();
```

This will cause the color named `GREEN` to be used when repainting.

The job of the `model.getView().repaint()` line is to repaint the view with the new data. We do this by telling the model to get the address of the view (the `getView` method) and call the View panel repaint method.

8.3. Now go back to the c-tor for the Color Panel class, and add these lines at the bottom:

```
add(jlColor);

add(jlSpacer);

add(jbColor);
```

These lines "add" the label, the spacer, and the button to the Color Panel, so that they show up.

9. Run the program to see that there is now a control that says "Color: Change", with the Change on a button.

Click on the "change" button and be sure the ellipse in the view turns green. This is just a temporary step to be sure things are coded right. Soon we will upgrade so that you can pick any color you want.

10. Next we use a cool built in control choosing colors in Java GUI applications. Go to the `changeColor` method.

10.1. Replace `Color.GREEN` with `JColorChooser.showDialog(a,b,c).` Here, `a,b,c`, are three arguments that will be replaced with the correct values in a minute.

10.2. The first argument, `a`, can be `this`, or a reference to some component on the screen.

10.3. The second argument, `b`, is the text that will be displayed to the user. Something like "`Choose new color`" will be good.

10.4. The third argument, `c` , is the color to be used in the demo parts of the `JColorChooser`. A good choice is the current color of the ball. To get this color, replace `c` with: `model.getColor();`.

11. It is a little tricky getting this right. The method should wind up looking like this:

```
private void changeColor() {
    model.setColor(JColorChooser
            .showDialog(
                    this,
                    "Choose new color",
                    model.getColor())
    );
    model.getView().repaint();
}
```

In situations like this, where several involved arguments are sent to a method, it is sometimes a good idea to write it out as above. This "one parameter per line" style makes it easy to read in case you have to do some serious thinking. On the other hand, it takes a lot of space. If you are sure what you are doing, or have a lot of similar lines, you can write it something like this:

```
model.setColor(JColorChooser.showDialog(this, "Choose new color",
     model.getColor()));
```

There is too much code for one line, so it will extend too far to the right. Thus, pick a good place to start a new line (usually after a comma). Always indent at least one tab if you are continuing a line. Try to break the line at a good place, such as after a comma, or before a dot. Note you CANNOT break a line inside a string, such as "`Choose new color`". If you have a really long string, and need to break it, do it like this: "`Choose new " + "color`", and go to a new line after the `+`.

The style of using a lot of lines (as above) is rather new (I have been seeing it now for a couple of years) and I think it is a good idea, and here to stay. It is all about making it easy for the programmer to get right. BTW, the programmer is YOU, so accept all the help you can get!

12. Here is what is happening. When the button gets clicked, up pops the built in `JColorChooser` component. This is as highly sophisticated component that will return whatever color the user selects. We will discuss this component in class.

13. Run the program. Fix if needed. Try the various ways of choosing a color. Say "Wow! This is cool!". The are a huge number of ways to select a color in the `JColorChooser`. We will discuss these in class.

14. The **solid panel** demonstrates another common way of inputting information into a running app.

Go to the `PnlSolid` class.

14.1 Create a A `JLabel jlSolid` with the text “Solid:” on it. This means Declare the field (above the c-tor) AND also instantiate the field (with the new operator). For example:

```
private final JLabel jlSolid = new JLabel("Solid:");
```

Create a `jlSpacer` label as before.

Also create the following objects (using no-argument c-tors for each)

14.2. A `JPanel jpButtons`.

14.3. A `JRadioButton jrbYes` with “Yes” on it.

14.4. A `JRadioButton jrbNo` with “No” on it.

14.5 A `ButtonGroup bgSolid`.

15. We already have a constructor for `PnlSolid` similar to that of the other control panels (with “super”). Everything else goes inside this c-tor, after the `super(model)` statement.

16. Put in the following code in the c-tor to initialize the radio buttons:

```
if (model.isSolid()) {

    jrbYes.setSelected(true);

} else {

    jrbNo.setSelected(true);

}
```

17. Add `ActionListeners` to the two radio buttons in the usual way. Each action listener will only need two lines of code in the `actionPerformed` section, so we can put the code in the AL’s rather than in a separate method.

17.1. Write the AL for `jrbYes` like this:

```
jrbYes.addActionListener(ae -> {

    model.setSolid(true);

    model.getView().repaint();

});
```

17.2. The similar code for `jrbNo` that sets `solid` to `false`.

18. Next add the two radio buttons to the button group. Note that this DOES NOT put the radio buttons where they can be seen (that will be next). The button group is a logical concept. Adding the buttons to the button group is how the button group knows what buttons are in the group.

19. Next we will put the buttons in a panel. Add the two radio buttons to the panel `jpButtons`.

20. So that we can actually see where this panel will turn up, add a thin (1 pixel) border to the panel like this:

```
jpButtons.setBorder(BorderFactory.createLineBorder(Color.CYAN));
```

Fix imports if needed

21. Last of all, add `jlSolid, jlSpacer`, and `jpButtons` to the `PnlSolid`.

22. Try the project out and see what the effect of clicking the buttons is. Ensure the project is working correctly.

23. The **Task Panel** will allow the user to select what kind of picture is drawn. This will be rather similar to the Color control above.

23.1. Go to the `PnlTask` class. We need a field `JLabel jlTask` with the text `Set Task:` on it. Create this object with the following statement (above the c-tor):

```
private final JLabel jlTask = new JLabel("Set Task:");
```

Fix imports, and do this whenever needed below.

Also create a `JLabel` `jlSpacer` label as in the other panel classes.

23.2. Add the following statement to declare a “combo box”:

```
private final JComboBox<Task> jcbTask;
```

This will cause an error that we will fix inside the c-tor.

24. Add a bunch of statements to the c-tor so that it looks like this:

```
    public PnlTask(Model model) {
        super(model);
        jcbTask = new JComboBox<>();                  // declare combo box
        for (Task task : Task.values()) {             // populate combo box
            jcbTask.addItem(task);
        }
        jcbTask.addActionListener(ae -> selectTask());  // action listener


        add(jlTask);                                   // usual stuff
        add(jlSpacer);
        add(jcbTask);
        System.out.println("In PnlTask c-tor");
    }
```

What is going on here will be discussed in class.

24. There will be an error message. Fix it by creating the method:

```
private void selectTask(){}
```

Put these statements in it:

```
model.setTask(Task.ELLIPSE);
model.getView().repaint();
```

Run the program just to be sure everything is working as before. This is to check that the refactoring has not broken anything, and the code in the `selectTask` method is basically good

25. Next we will enable actual selection of new tasks. Change the code `selectTask` to:

```
        Task task = jcbTask.getItemAt(jcbTask.getSelectedIndex());
        System.out.println("task: " + task);
        model.setTask(task);
        model.getView().repaint();
```

Here we get to obtain a selected task from the combo box control. We then write the task to output in case we need to see what happened if things don't work correctly.

Then we reset the task (what is to be drawn in the view), and repaint the picture.

26. Run the program. Click on the “down triangle” on the combo box to pull down the “pull down menu”. Recall that so far, we only have one of the four tasks implemented in the View class, so be sure to select (click on) ELLIPSE. This should redraw the ellipse, just like before.

27. Now run the program, and select something else, like RECTANGLE. Because we have not yet implemented the code to draw the rectangle, there is no code for this situation in the `switch` statement in the `View` class. Therefore this selection will throw an exception. Don’t worry, this is the expected behavior, we are just checking to see it that happens. We implement the other tasks presently.

What to turn in.

1. Open file explorer

2. Find your project (likely C:\Users\<user>\IdeaProjects\MVC22F)

3. Right click on your project folder (this should contain src\mvc22f with all your .java files)

4. Select Send to > Compressed (zipped) folder

5. Submit MVC22F.zip to Blackboard