Activate the buttons.

To do:

Add an event listener with a method that will be called when Cycle button is pressed.

Event listener must by a class that implements ActionListener interface.

(the only requirement for ActionListener is inclusion of an actionPerformed() method)

Sometimes you see programs where the GUI class itself does double duty as a listener:

Aka, it implements ActionListener, and has an actionPerformed() method.

This approach runs into problems when there are multiple buttons:

Each button ends up activating the same actionPerformed() method.

At the same time, it doesn’t make sense to implement a completely separate

Class to handle GUI events because this class only makes sense within the

context of the GUI.

Solution: Use private nested classes – aka, short utility classes that are defined within the BullseyeGUI class and can therefore access all of its members.

Here is the definition of a nested even listener class for the Cycle button:

/\*\* Even handler for Cycle button \*/

class CycleListener implements ActionListener{

/\*\*

\* Cycles the colors when the button is pushed.

\* @param e Holds information about the button-push event

\*/

public void actionPerformed(ActionEvent e){

bull.cycle(); // note the reference to the enclosing class’s //private field

}

}

put the above code inside BullseyeGUI.java

but at the end, after all the other methods (including main).

Before our button will function, it must be told about the listener we have created so that it can

activate the actionPerformed() method whenever the user clicks.

The following code goes inside createComponents() right after the button has been added to the pane:

cycleButton.addActionListener(new CycleListener());

Now go ahead and write the Invert button handler yourself, adding it to the code and registering it in the same manner.

/\*\* Event handler for Invert button \*/

class InvertListener implements ActionListener{

/\*\*

\* Inverts the colors when the button is pushed.

\* @ param b Holds information about the button-push event

\*/

public void actionPerformed(ActionEvent b){

bull.invert()

}

}

Good job! Pat yourself on the back for figuring out part 1 ☺

Let’s move on to the next thing, shall we?

It’s not hard to add simple animations to your GUI programs using events and the Java Swing Timer class.

The basic idea is that we will create a Timer instance, which causes an event to fire either once or repeatedly at some specified interval.

The Timer starts in an inactive state, and thereafter its behavior is controlled by calls to its start() and stop() methods.

Because the Timer will be used in several places within the GUI class, it should be a class field.

We will also establish a numeric constant that specifies how often the Timer should fire (in thousandths of a second).

/\*\* Used for animation of the bullseye colors \*/

private Timer timer;

/\*\* Time between updates is half a second \*/

private static final int TIMER\_INTERVAL = 500;

We will have to instantiate the Timer object and assign it to our new field.

It makes sense to do it in the BullseyeGUI constructor:

timer = new Timer(TIMER\_INTERVAL, new CycleListener());

Note that as shown above, creation of a Timer takes as arguments the firing interval in milliseconds and an appropriate ActionListener object.

Since we already have an ActionListener class that cycles the colors,

we can just reuse it.

We have now created a timer, but since it begins in the inactive state and we have no way

to turn it on, it isn’t much use yet.

Add two more buttons, Start and Stop to the layout.

The event handlers for these should call timer.start() and timer.stop()

You also need to insert code within createComponents() adding the new buttons

to the panel.

Action events are quite useful, applying as they do to buttons, checkboxes, radio

buttons, and menu items.

However, they are not the only kind of event that programs can generate.

Another very useful event type is the MouseEvent.

A basic mouse handler will implement the MouseListener interface,

which is somewhat more complicated than ActionListener.

MouseListener needs implementation of five different methods:

mouseClicked()

mousePressed()

mouseReleased()

mouseEntered()

mouseExited()

Not all of these need to do anything, but they must all be defined,

Possibly with empty bodies.

To make this easier, Java provides a predefined class called MouseAdapter that has all five

empty methods already in place.

You can create a class that inherits from MouseAdapter, and then simply override with new

definitions any methods you want active.

Anyways, Java also defines MouseMotionListener and MouseWheelListener interfaces for responding to drag events and scroll wheel events, respectively.

The MouseAdapter class also implements these interfaces and provides empty

event handlers for them.

Let’s override some of those empty handlers with ones tht provide information analogous to the five handlers already written.

If you look at the MouseMotionListener interface, you will see that it promises handlers called mouseMoved() and mouseDragged().

Go ahead and write these in your SampleMouseListener class, making them print the mouse location just like the other handlers.

Before you can see them in action, you have to register the listener using the appropriate

JComponent’s addMouseMotionListener() method.

Do this in createComponents(), right below the line that adds the mouse listener.

Even though the same class is handling both types of mouse events, it needs to be added as a listener twice. One for each kind.

Add a method to JBullseye that computes the index of the ring from (x,y) coordinates.

The outer ring is index 0.