Programming Languages and Techniques (CIS120)

Lecture 35

Swing II: Event Handlers, Inner Classes and Layout

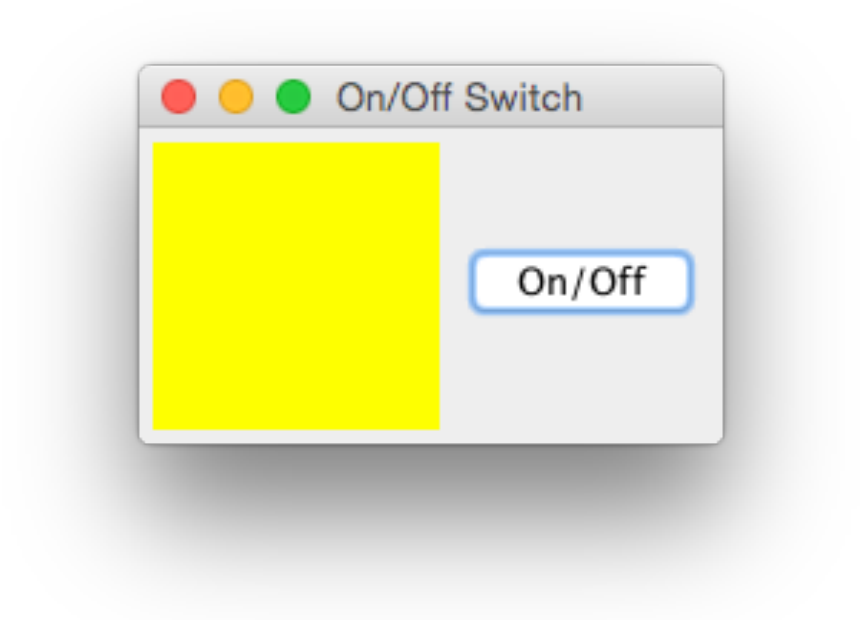
Chapter 30

Swing: User Interaction

Java's GUI Library

Start Simple: Lightswitch

Task: Program an application that displays a button. When the button is pressed, it toggles a “lightbulb” on and off.



Key idea: use a ButtonListener to toggle the state of the "lightbulb"

OnOffDemo

The Lightswitch GUI program in Swing.

Display the Lightbulb

```
class LightBulb extends JComponent {
    private boolean isOn = false;

    public void flip() {
        isOn = !isOn;
    }
    public void paintComponent(Graphics gc) {
        if (isOn) {
            gc.setColor(Color.YELLOW);
        } else {
            gc.setColor(Color.BLACK);
        }
        gc.fillRect(0, 0, 100, 100);
    }
    public Dimension getPreferredSize() {
        return new Dimension(100,100);
    }
}
```

Remember the private state of the lightbulb

Draw the Light bulb here using the graphics context

Set the size of the window

Main Class

```
public class OnOff implements Runnable {
  public void run() {
      JFrame frame = new JFrame("On/Off Switch");
      JPanel panel = new JPanel();
      frame.getContentPane().add(panel);
      LightBulb bulb = new LightBulb();
      panel.add(bulb);
      JButton button = new JButton("On/Off");
      panel.add(button);
      button.addActionListener(new ButtonListener(bulb));
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
  }
  public static void main(String[] args) {
       SwingUtilities.invokeLater(new OnOff());
  }
}
```

Open frame and make a panel

Create bulb and button

Start the (Swing) application

Making the Button DO something

```
class ButtonListener implements ActionListener {
    private LightBulb bulb;
    public ButtonListener (LightBulb b) {
        bulb = b;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        bulb.flip();
        bulb.repaint();
    }
}
```

Note that "repaint" does not necessarily do any repainting now! It is simply a notification to Swing that something needs repainting.

An Unflattering Comparison

Java

```
class ButtonListener implements ActionListener {
     private LightBulb bulb;
     public ButtonListener (LightBulb b) {
          bulb = b;
     }
     @Override
     public void actionPerformed(ActionEvent e) {
          bulb.flip();
          bulb.repaint();
     }
}

// somewhere in run …
LightBulb bulb = new LightBulb();
JButton button = new JButton("On/Off");
button.addActionListener(new ButtonListener(bulb));
```

OCaml

```
let bulb, bulb_flip = make_bulb ()
let onoff,_, bnc = button "ON/Off"
;; bnc.add_event_listener (mouseclick_listener bulb_flip)
```

Too much "boilerplate"!

- ButtonListener really only needs to do `flip()` and `repaint()`
- But we need all this extra boilerplate code to build the class
- Often we will only instantiate *one* instance of a given Listener class in a GUI

```
class ButtonListener implements ActionListener {
    private LightBulb bulb;
    public ButtonListener (LightBulb b) {
        bulb = b;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        bulb.flip();
        bulb.repaint();
    }
}
```

Inner Classes

Inner Classes

- Useful in situations where objects require “deep access” to each other’s internal state

- Replaces tangled workarounds like the “owner object” pattern
  - Solution with inner classes is easier to read
  - No need to allow public access to instance variables of outer class

- Also called “dynamic nested classes”

Basic Example

Key idea: Classes can be *members* of other classes…

```
class Outer {
  private int outerVar;
  public Outer () {
    outerVar = 6;
  }
  public class Inner {
    private int innerVar;
    public Inner(int z) {
      innerVar = z;
    }
    public int getSum() {
      return outerVar + innerVar;
    }
  }
}
```

Name of this class (i.e., the static type of objects that this class creates) is Outer.Inner

Inner classes can have their own fields and methods.

Reference from inner class to field bound in outer class

Constructing Inner Class Objects

```
class Outer {
  private int outerVar;
  public Outer () {
    outerVar = 6;
  }
  public class Inner {
    private int innerVar;
    public Inner(int z) {
      innerVar = z;
    }
    public int getSum() {
      return outerVar +
             innerVar;
    }
  }
}
```

Based on your understanding of the Java object model, which of the following make sense as ways to construct an object of an inner class type?

1. `Outer.Inner obj = new Outer.Inner(2);`
2. `Outer.Inner obj = (new Outer()).new Inner(2);`
3. `Outer.Inner obj = new Inner(2);`
4. `Outer.Inner obj = Outer.Inner.new(2);`

Answer: 2 – the inner class instances can refer to non-static fields of the outer class (even in the constructor), so the invocation of "new" must be relative to an existing instance of the Outer class.

Object Creation

- Inner classes can refer to the instance variables and methods of the outer class
- Inner class instances usually created by the methods/constructors of the outer class

```
public Outer () {
    Inner b = new Inner ();
}
```

Actually `this.new`

- Inner class instances *cannot* be created independently of a containing class instance.

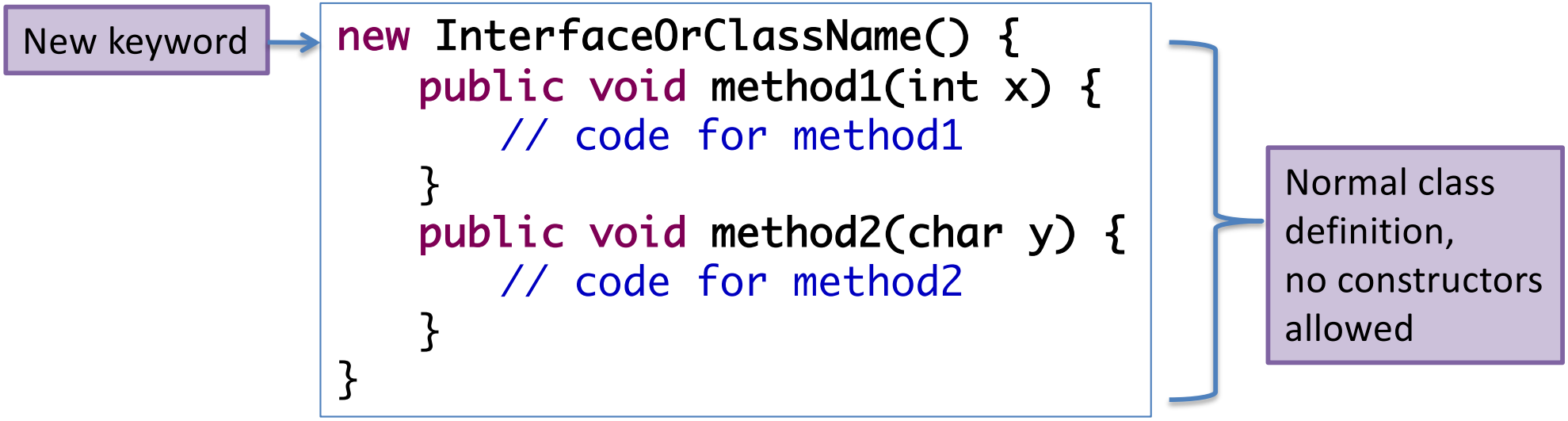
```
Outer.Inner b = new Outer.Inner()
```

```
Outer a = new Outer();
Outer.Inner b = a.new Inner();
```

```
Outer.Inner b = (new Outer()).new Inner();
```

Anonymous Inner Classes

- New *expression* form: define a class and create an object from it all at once, inside a method of another class



Static type of the expression is the Interface/superclass used to create it

Dynamic class of the created object is anonymous! Can't refer to it.

Anonymous Inner Classes

- Define a class *and create an object* from it all at once, inside a method

```
final LightBulb bulb = new LightBulb();
JButton button = new JButton("On/Off");

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        bulb.flip();
        bulb.repaint();
    }
});
```

Anonymous Inner Classes

```
quit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
```

Puts button action with button definition

```
line.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        shapes.add(new Line(…));
        canvas.repaint();
    }
});
```

Can access fields and methods of outer class, as well as **final** local variables

Like first-class functions

- Anonymous inner classes are a Java equivalent of OCaml's first-class functions
- Both create "delayed computations" that can be stored in a data structure and run later
  - Code stored by the event / action listener
  - Code only runs when the button is pressed
  - Could run once, many times, or not at all
- Both sorts of computation can refer to variables in the current scope
  - OCaml: Any available variable
  - Java: only variables marked `final`

Lambda Expressions

- Java 8 introduced *lambda expressions* which are simplified syntax for anonymous classes with just one method

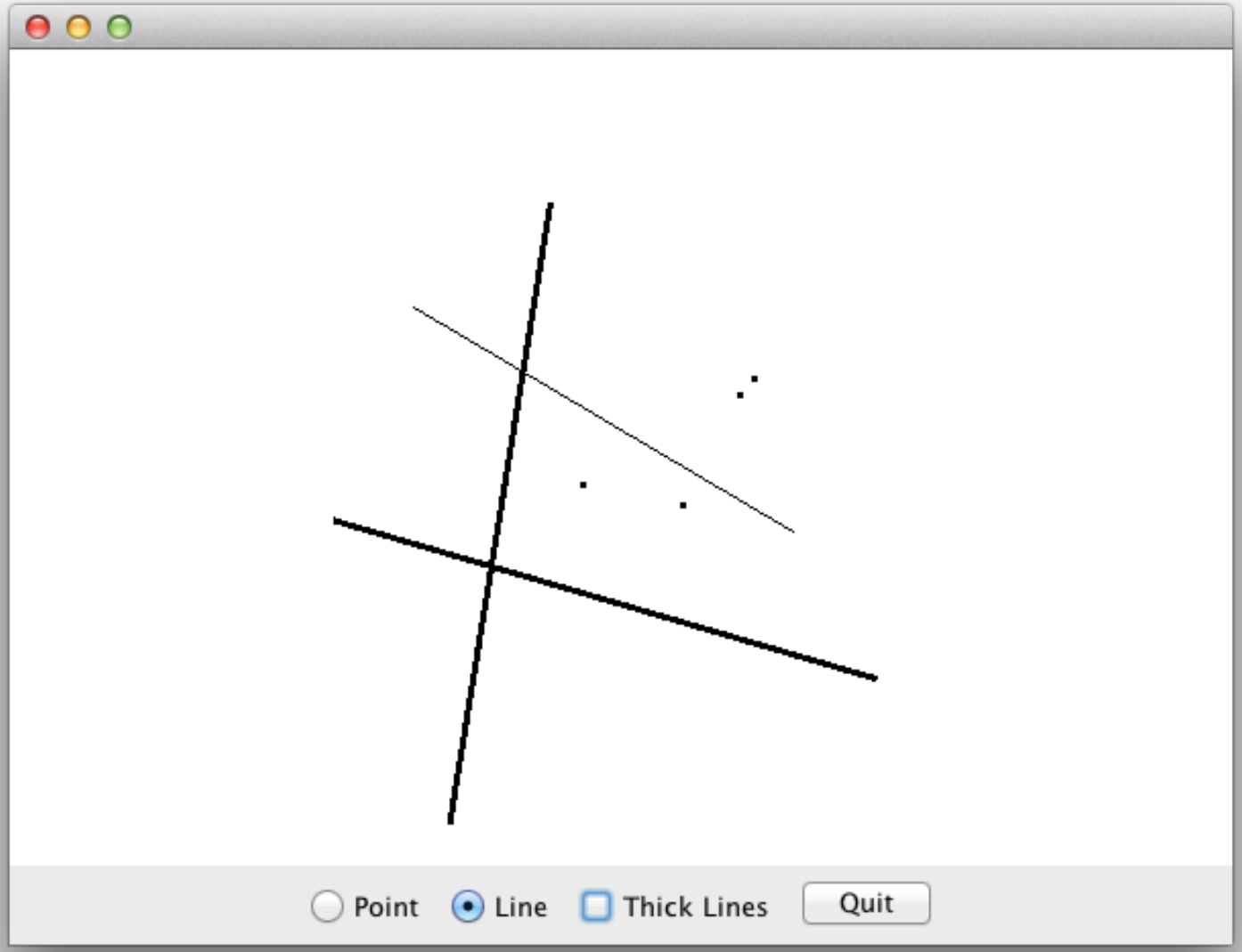
```
final LightBulb bulb = new LightBulb();
JButton button = new JButton("On/Off");

button.addActionListener(e -> {
        bulb.flip();
        bulb.repaint();
    });
```

- Any interface with exactly one method is called a *functional interface*
- Syntax:

```
x -> { body }              // type of x inferred
(T x) -> { body }          // arg x has type T
(T x, W y) -> { body }     // multiple arguments
```

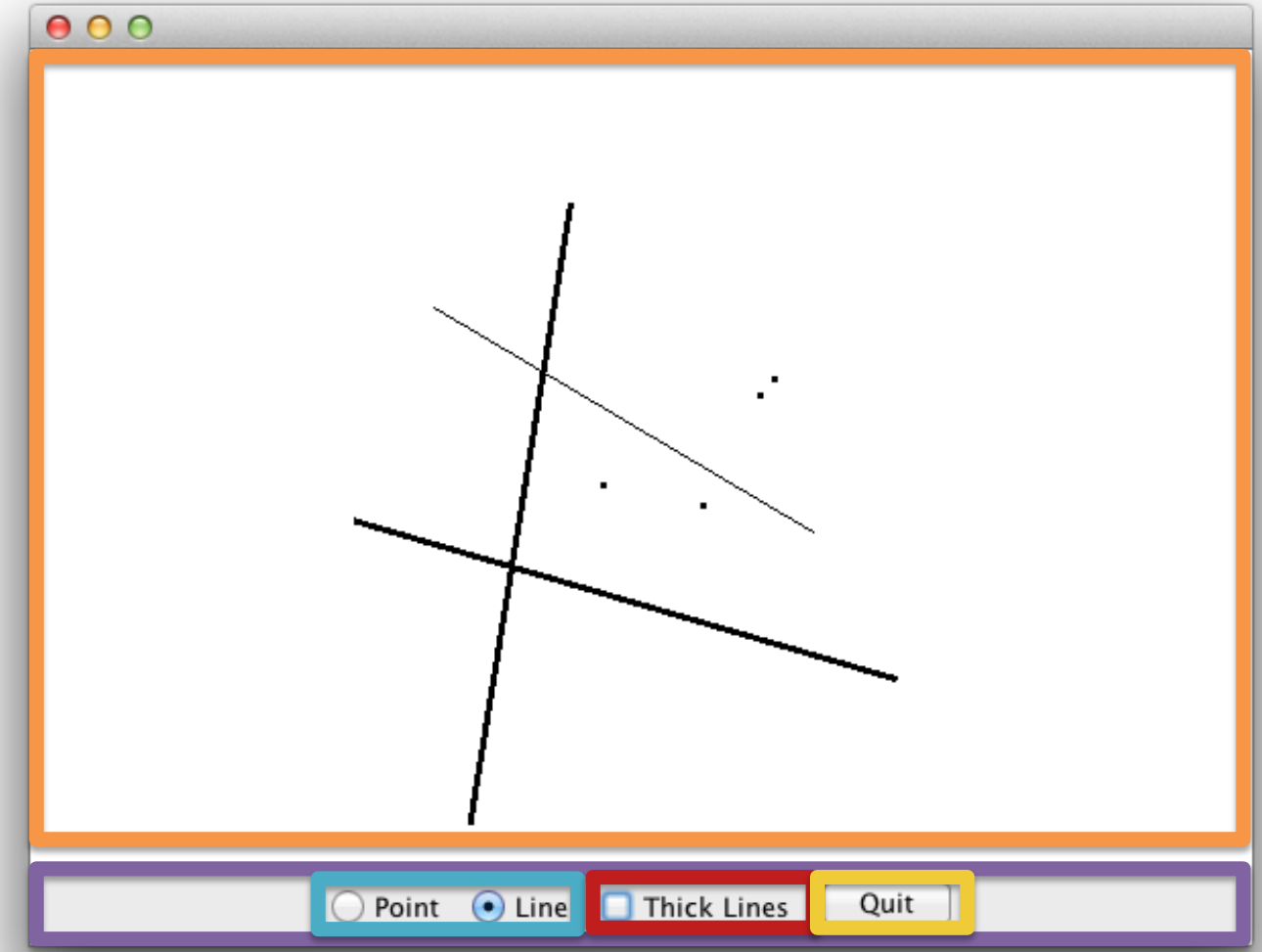
Swing Layout Demo

LayoutDemo.java



What layout would you use for this app? What components would you use?

Canvas
subclass of
JPanel
(canvas)



JPanel
(toolbar)

JRadioButton
(point, line)

JCheckbox
(thick)

JButton
(quit)

Paint Revisited

Using Anonymous Inner Classes

Refactoring for OO Design

Paint Revisited
(thoroughly discussed in Chap 31)

Using Anonymous Inner Classes

Refactoring for OO Design

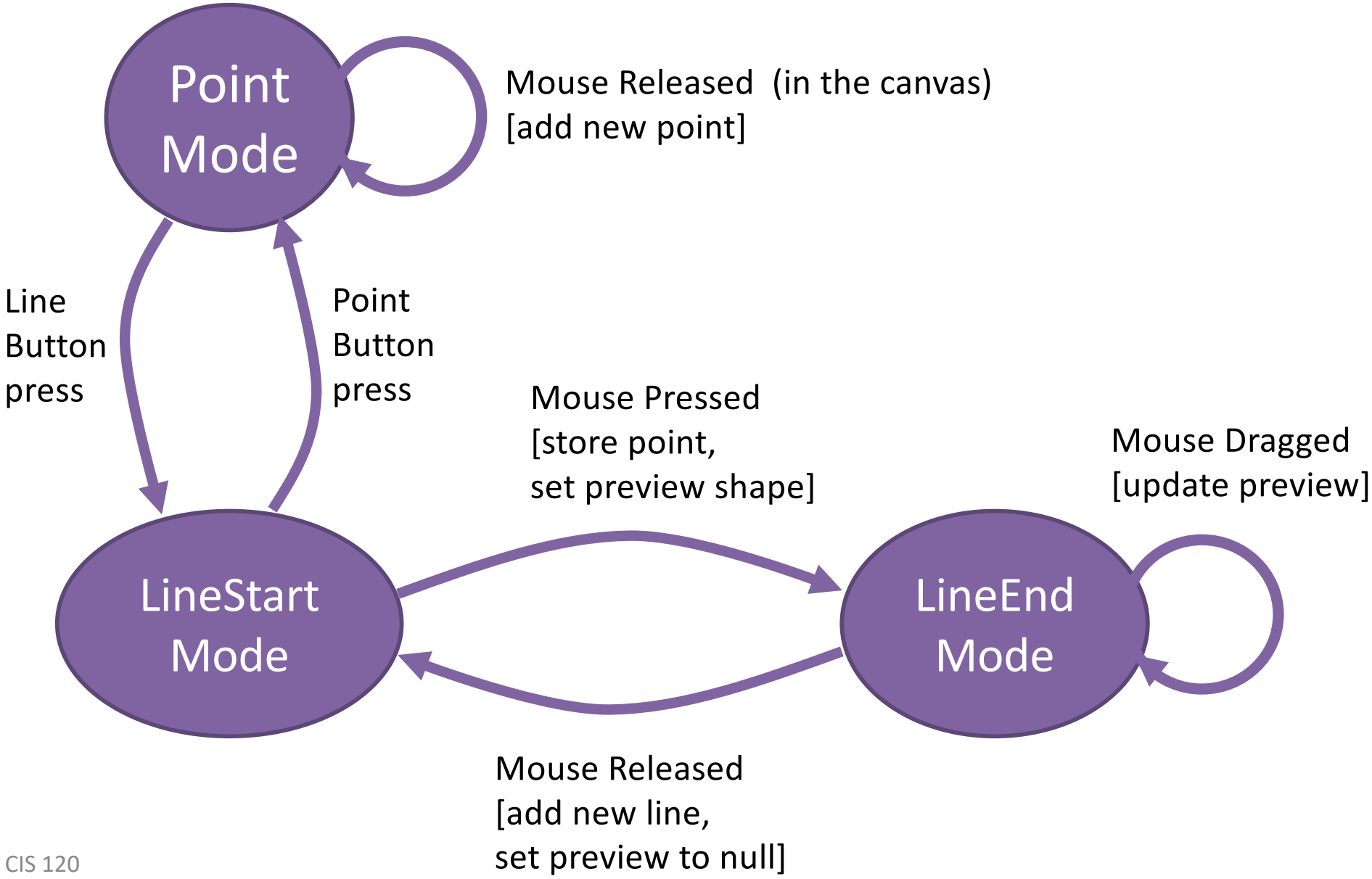
(See PaintA.java … PaintE.java)

Adapters

MouseAdapter

KeyAdapter

Mouse Interaction in Paint



Two interfaces for mouse listeners

```
interface MouseListener extends EventListener {
   public void mouseClicked(MouseEvent e);
   public void mouseEntered(MouseEvent e);
   public void mouseExited(MouseEvent e);
   public void mousePressed(MouseEvent e);
   public void mouseReleased(MouseEvent e);
}
```

```
interface MouseMotionListener extends EventListener {
   public void mouseDragged(MouseEvent e);

   public void mouseMoved(MouseEvent e);
}
```

Lots of boilerplate

- There are seven methods in the two interfaces.
- We only want to do something interesting for three of them.
- Need "trivial" implementations of the other four to implement the interface…

```
public void mouseMoved(MouseEvent e)   { return; }
public void mouseClicked(MouseEvent e) { return; }
public void mouseEntered(MouseEvent e) { return; }
public void mouseExited(MouseEvent e)  { return; }
```

- Solution: MouseAdapter class…

Adapter classes:

- Swing provides a collection of abstract event adapter classes
- These adapter classes implement listener interfaces with empty, do-nothing methods
- To implement a listener class, we extend an adapter class and override just the methods we need

```
private class Mouse extends MouseAdapter {
    public void mousePressed(MouseEvent e) { … }
    public void mouseReleased(MouseEvent e) { … }
    public void mouseDragged(MouseEvent e) { … }
}
```