Programming Languages and Techniques (CIS120)

Lecture 34

Swing II: Inner Classes and Layout

Chapter 30

Announcements

- HW8: TwitterBot
  - Due: Tuesday, November 26th at 11:59pm
  - This is a *new* project (replacing SpellChecker), so ask for clarifications!

- HW9a: Game Proposal Due *NOW*
- HW9: Game – Due Monday, December 9th at 11:59pm

- Regrade requests for Midterm 2 due by *tonight* at 11:59pm

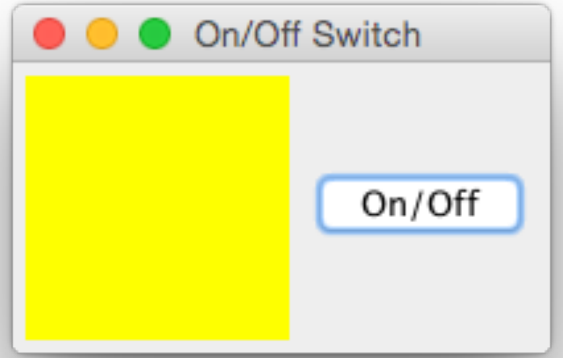
- Wednesday, November 27th – Bonus Lecture
  - Only 11:00 AM class
  - Material is not needed for HW or Exams
  - Should be fun!

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;
    }
    @Override
    public void paintComponent(Graphics gc) {
       // display the light bulb here
        if (isOn) {
            gc.setColor(Color.YELLOW);
        } else {
            gc.setColor(Color.BLACK);
        }
        gc.fillRect(0, 0, 100, 100);
    }

    @Override
    public Dimension getPreferredSize() {
         return new Dimension(100,100);
    }
}
```

Remember / update 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 Awkward 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));
```

Java

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

OCaml

Too much “boilerplate”!

- ButtonListener really only needs to do bulb.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 internals
- 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…

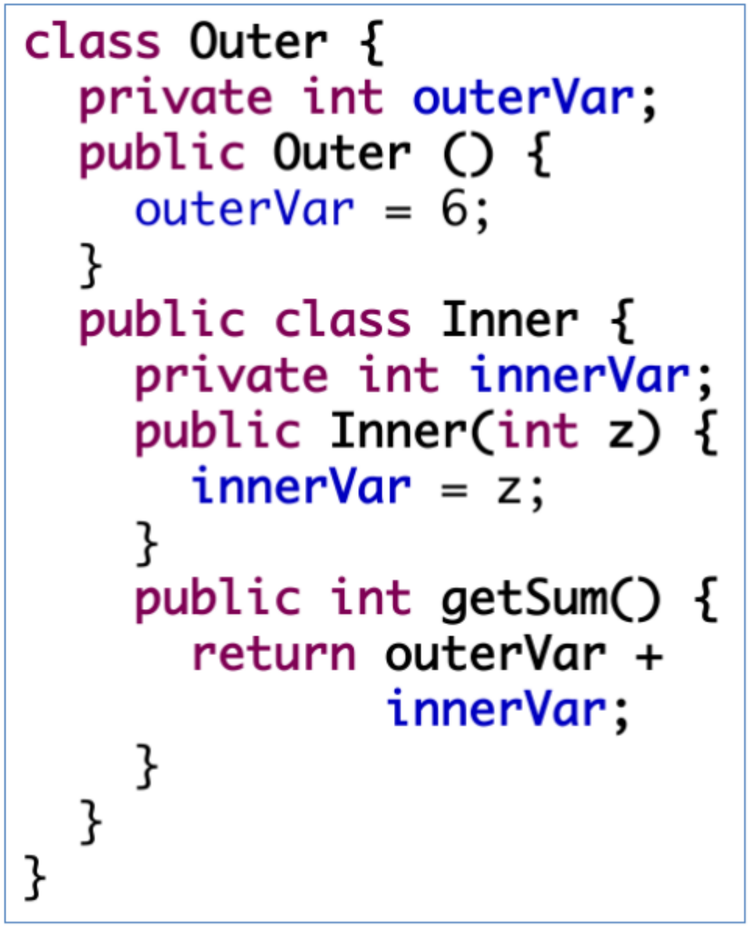
```
class Outer {
  private int outerVar;
  public Outer () {
    outerVar = 6;
  }
  public class Inner {
    private int innerVar;
    public Inner(int z) {
      innerVar = z;
    }
    public int getSum() {
      return outerVar + innerVar;
    }
  }
}
```

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

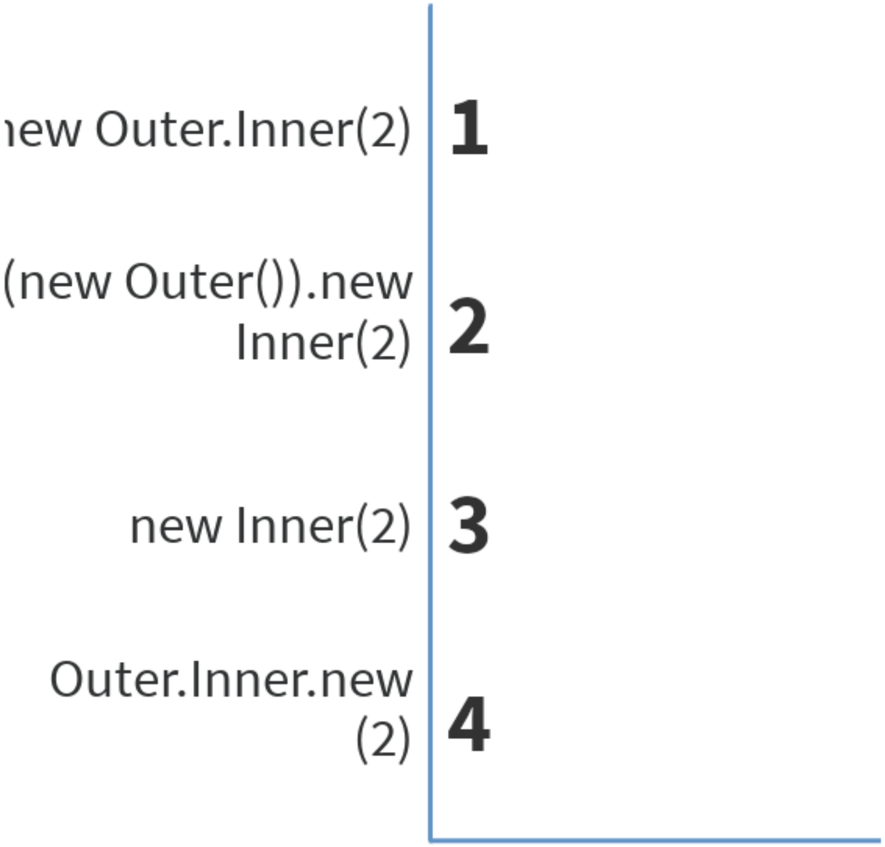
Inner classes can have their own fields and methods.

Reference from inner class to field bound in outer class

In Java, which makes sense for creating an object of type Outer.Inner?



```
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;
    }
  }
}
```



new Outer.Inner(2) 1

(new Outer()).new Inner(2) 2

new Inner(2) 3

Outer.Inner.new (2) 4

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

- 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

Anonymous Inner Classes

- New *expression* form: define a class and create an object from it all at once

New keyword

```
new InterfaceOrClassName() {
    public void method1(int x) {
        // code for method1
    }
    public void method2(char y) {
        // code for method2
    }
}
```

Normal class definition, no constructors allowed

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.

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 "functional interfaces” 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 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

After the lectures so far, how confident are you in your ability to work with Swing?

I’m hopelessly lost

OK, but I will probably need guidance

I can probably figure it out myself with some experimentation.

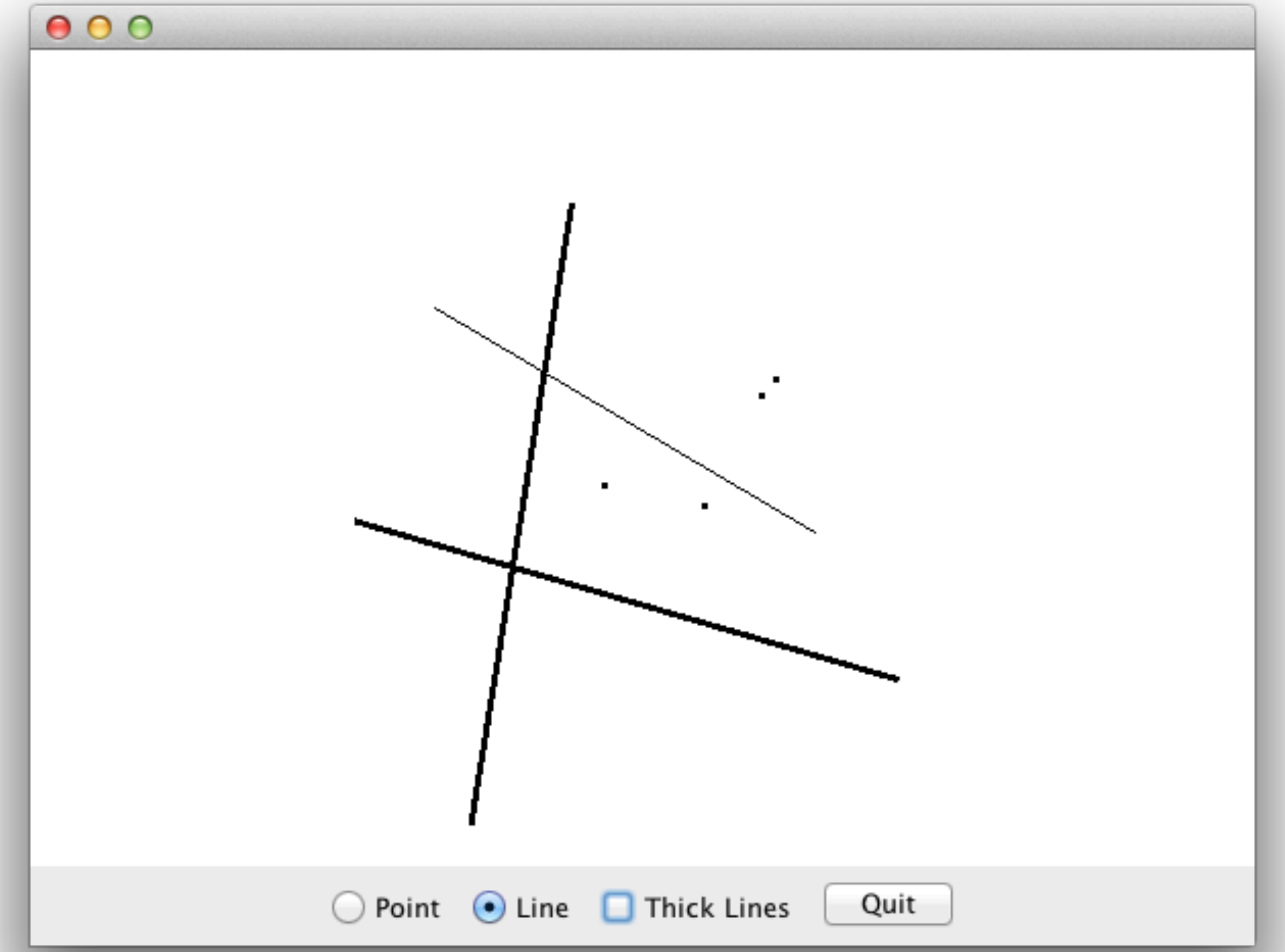
No problem, seems pretty straightforward

Total Results

Paint Revisited

Using Anonymous Inner Classes

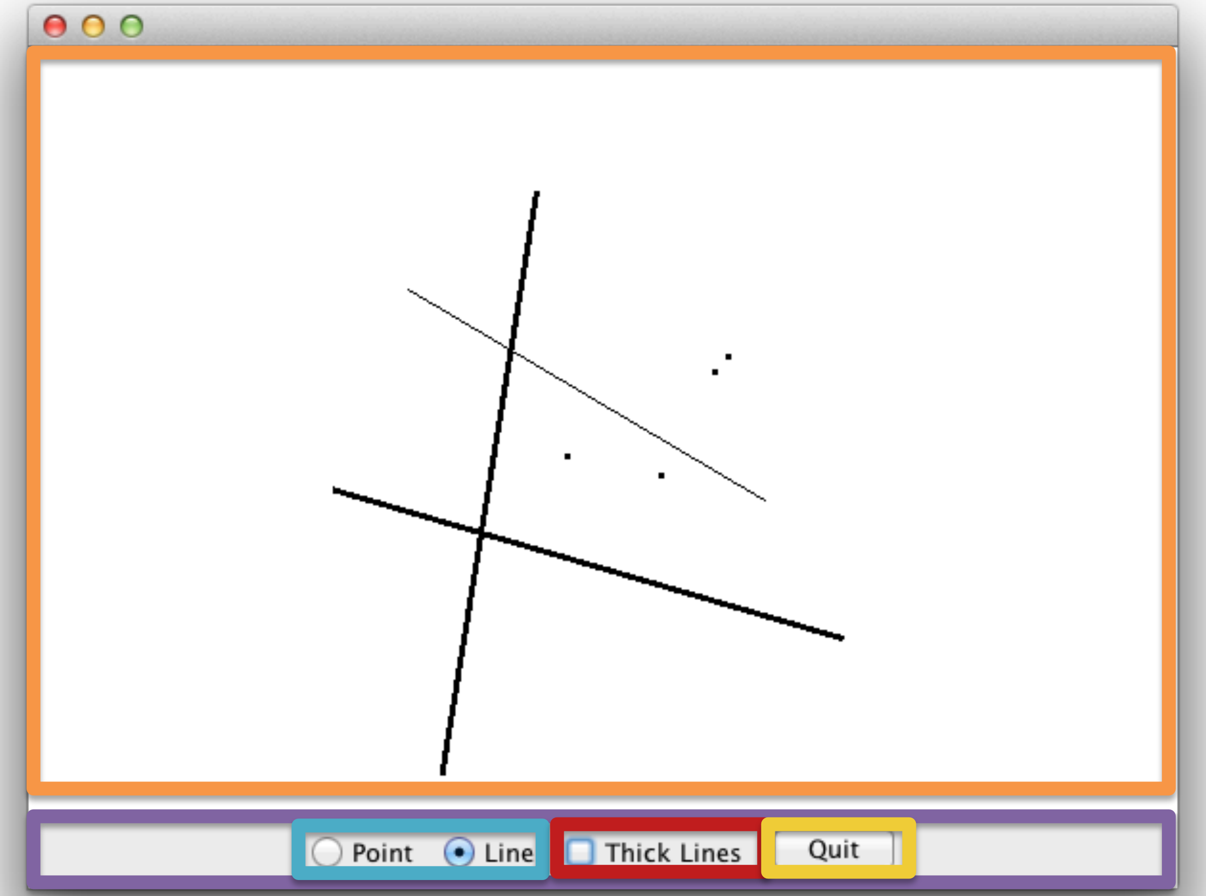
Refactoring for OO Design



○ Point ● Line □ Thick Lines Quit

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

Canvas
subclass of
JPanel
(canvas)



○ Point ● Line □ Thick Lines Quit

JPanel
(toolbar)

JRadioButton
(point, line)

JCheckbox
(thick)

JButton
(quit)

Paint Revisited (thoroughly discussed in Chap 31)

Using Anonymous Inner Classes

Refactoring for OO Design

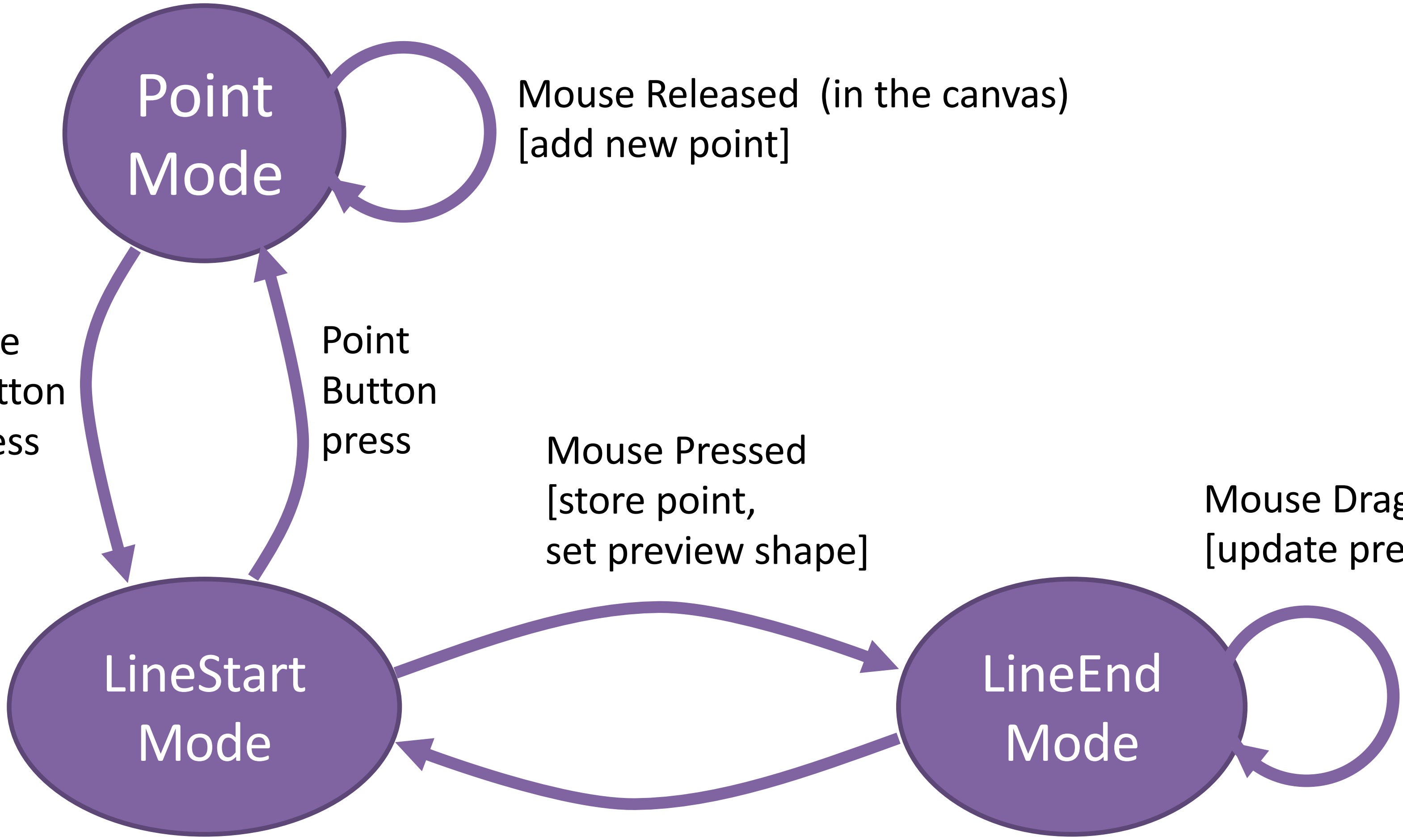
(See PaintA.java … PaintE.java)

Adapters

MouseAdapter

KeyAdapter

Mouse Interaction in Paint



Point Mode

Mouse Released (in the canvas)
[add new point]

Line
Button
press

Point
Button
press

LineStart Mode

Mouse Pressed
[store point,
set preview shape]

LineEnd Mode

Mouse Dragged
[update preview]

Mouse Released
[add new line,
set preview to null]

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…

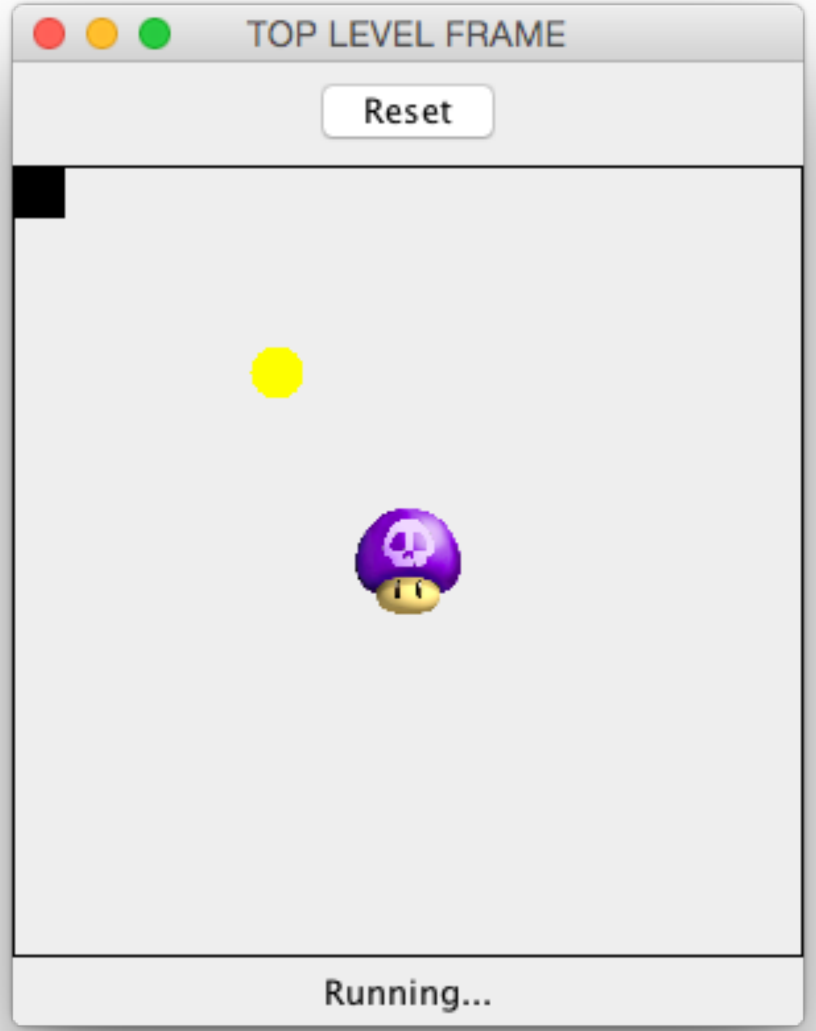
Adapter classes:

- 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

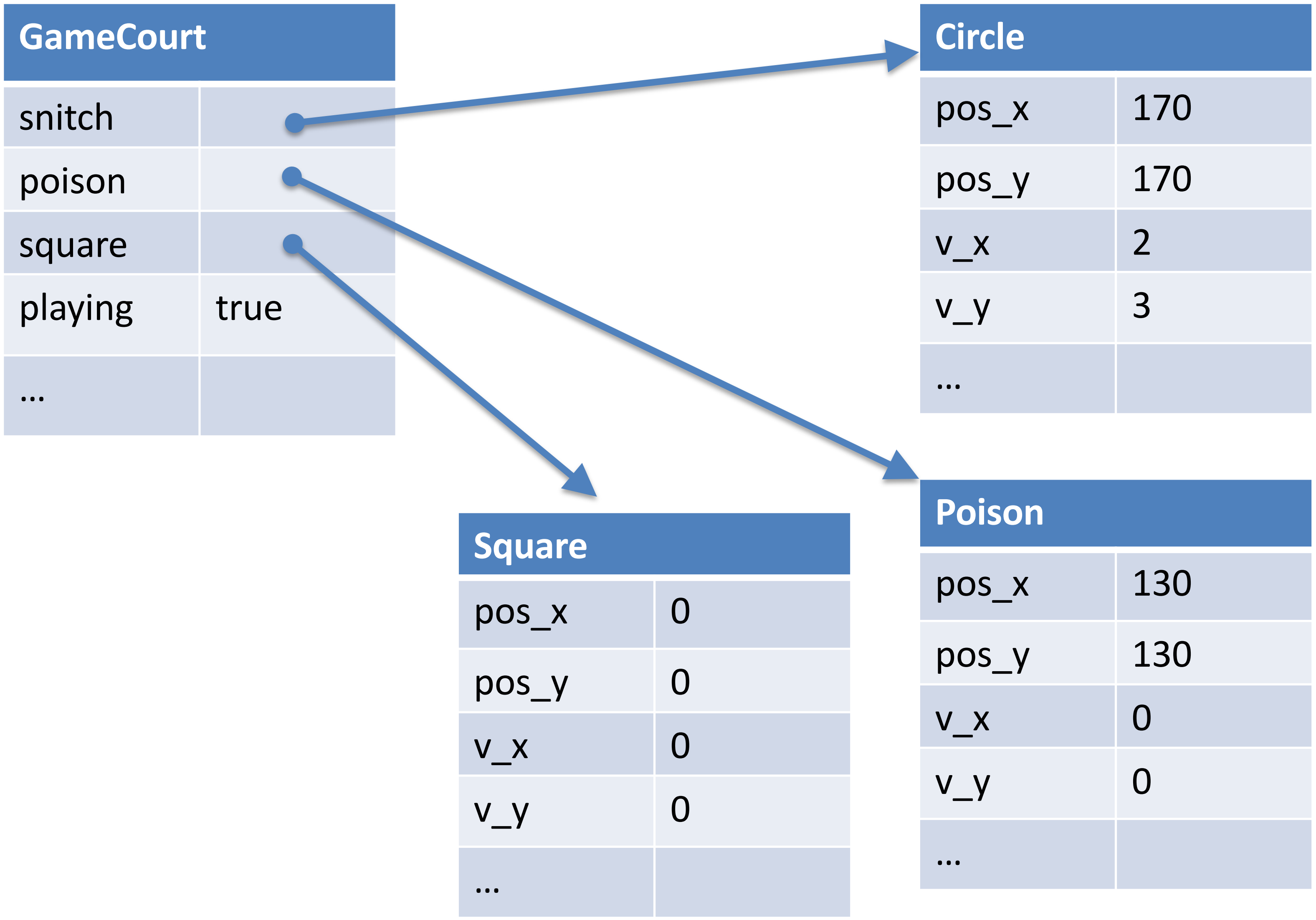
```
private class Mouse extends MouseAdapter {
   public void mousePressed(MouseEvent e) { … }
   public void mouseReleased(MouseEvent e) { … }
   public void mouseDragged(MouseEvent e) { … }
}
```

Mushroom of Doom

How do we put Swing components together to make a complete game?


TOP LEVEL FRAME
Reset
Running...

Game State

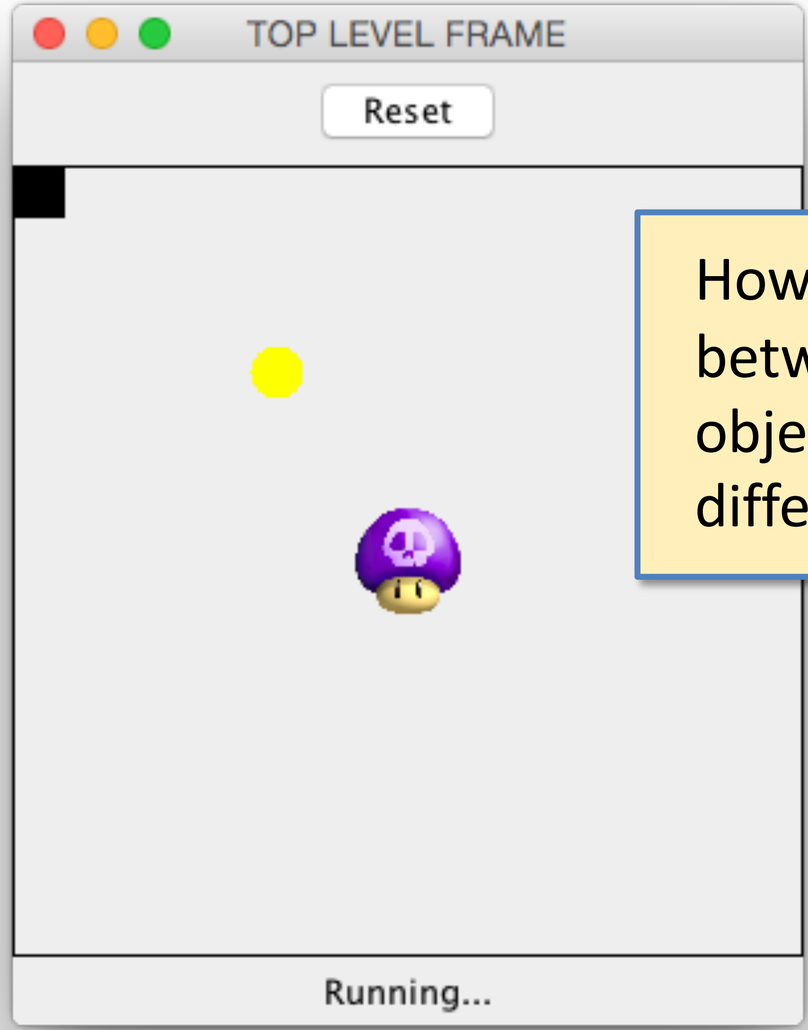


GameCourt	
snitch	●
poison	●
square	●
playing	true
…	

Circle	
pos_x	170
pos_y	170
v_x	2
v_y	3
…	

Square	
pos_x	0
pos_y	0
v_x	0
v_y	0
…	

Poison	
pos_x	130
pos_y	130
v_x	0
v_y	0
…	



How can we share code between the game objects, but show them differently?

Updating the Game State: timer

```
void tick() {
  if (playing) {
    square.move();
    snitch.move();
    snitch.bounce(snitch.hitWall());      // bounce off walls...
    snitch.bounce(snitch.hitObj(poison)); // ...and the mushroom

    if (square.intersects(poison)) {
      playing = false;
      status.setText("You lose!");
    } else if (square.intersects(snitch)) {
      playing = false;
      status.setText("You win!");
    }
    repaint();
  }
}
```

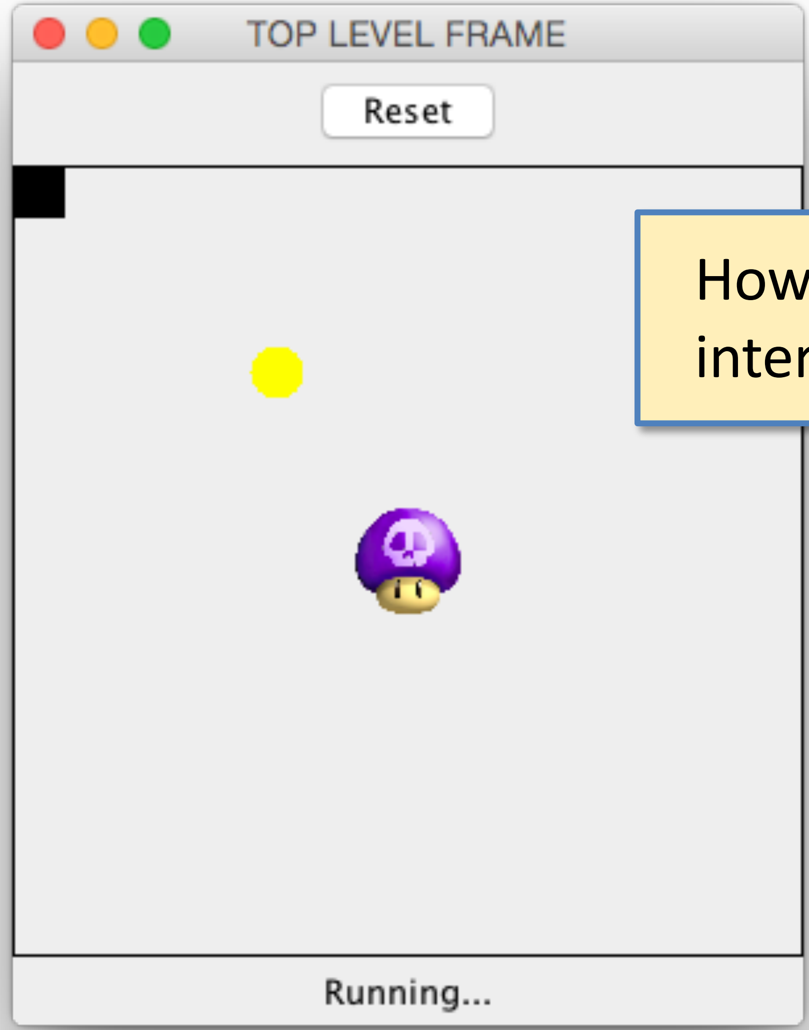
Updating the Game State: keyboard

```
setFocusable(true);
addKeyListener(new KeyAdapter() {
  public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_LEFT)
        square.v_x = -SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_RIGHT)
        square.v_x = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_DOWN)
        square.v_y = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_UP)
        square.v_y = -SQUARE_VELOCITY;
    }

    public void keyReleased(KeyEvent e) {
        square.v_x = 0;
        square.v_y = 0;
    }
});
```

Make square's velocity nonzero when a key is pressed

Make square's velocity zero when a key is released

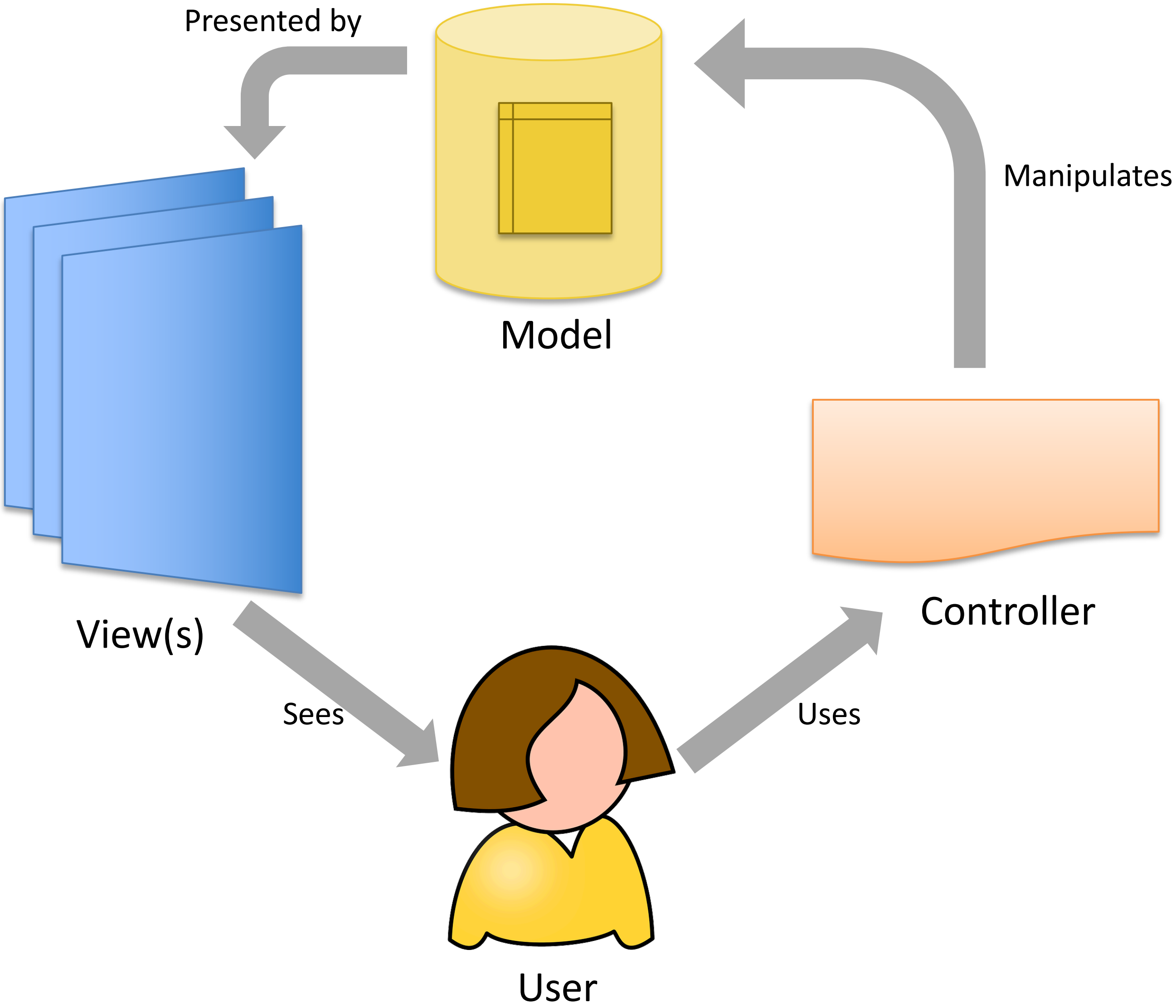


How does the user interact with the game?

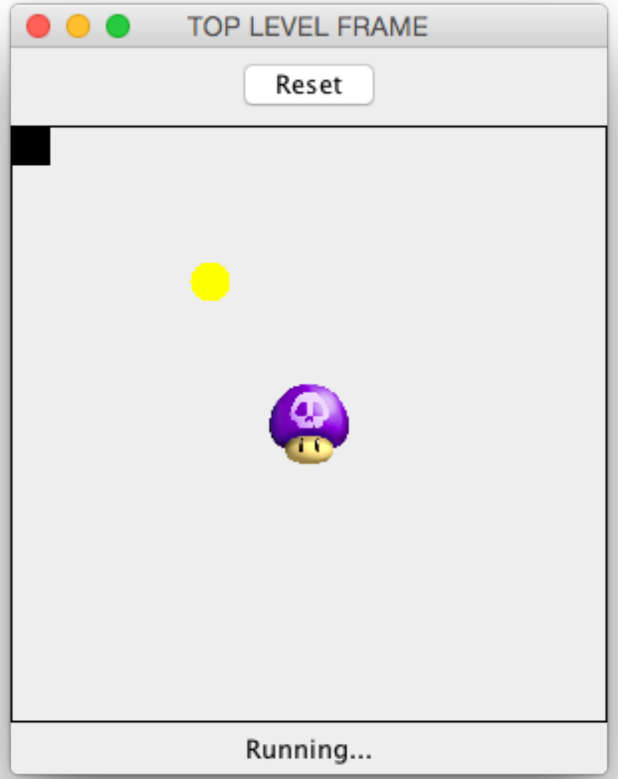
1. Clicking Reset button restarts the game
2. Holding arrow key makes square move
3. Releasing key makes square stop

Model View Controller Design Pattern

MVC Pattern


Presented by
Model
Manipulates
View(s)
Controller
Sees
Uses
User

Example 1: Mushroom of Doom



Running...

Example: MOD Program Structure

- GameCourt, GameObj + subclass local state
  - object location & velocity
  - status of the game (playing, win, loss)
  - how the objects interact with eachother (tick)

Model

- Draw methods
  - paintComponent in GameCourt
  - draw methods in GameObj subclasses
  - status label

View

- Game / GameCourt
  - Reset button (updates model)
  - Keyboard control (updates square velocity)

Controller

Example: Paint Program Structure

- Main frame for application (class Paint)
  - List of shapes to draw
  - The current color
  - The current line thickness

Model

- Drawing panel (class Canvas, inner class of Paint)

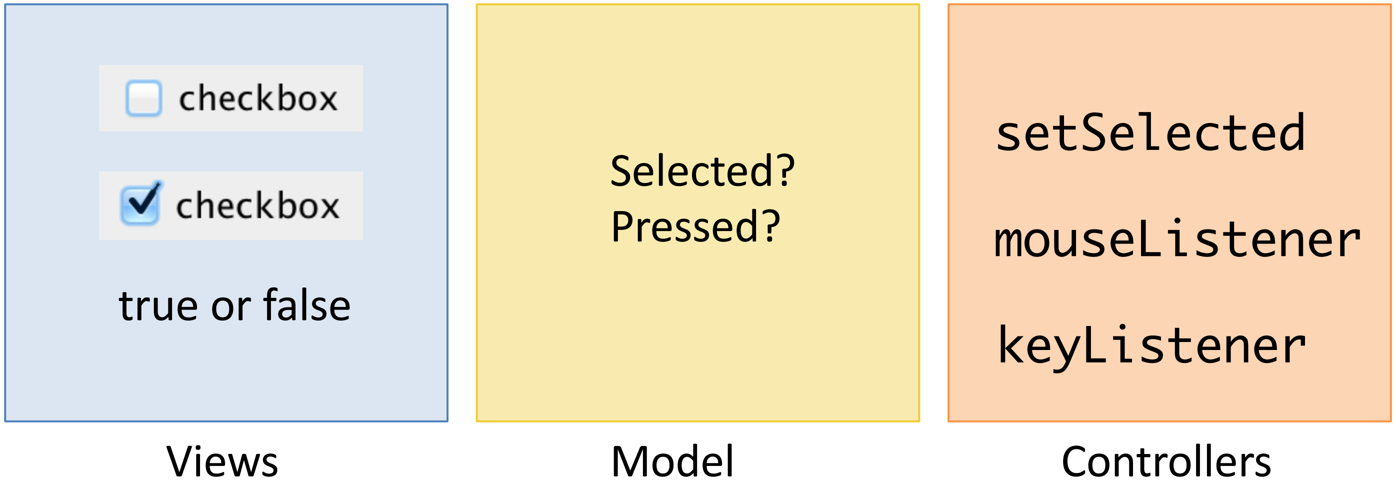
View

- Control panel (class JPanel)
  - Contains radio buttons for selecting shape to draw
  - Line thickness checkbox, undo and quit buttons

Controller

- Connections between Preview shape (if any…)
  - Preview Shape: View <-> Controller
  - MouseAdapter: Controller <-> Model

Example: CheckBox



☐ checkbox

☑ checkbox

true or false

Selected?
Pressed?

setSelected

mouseListener

keyListener

Views

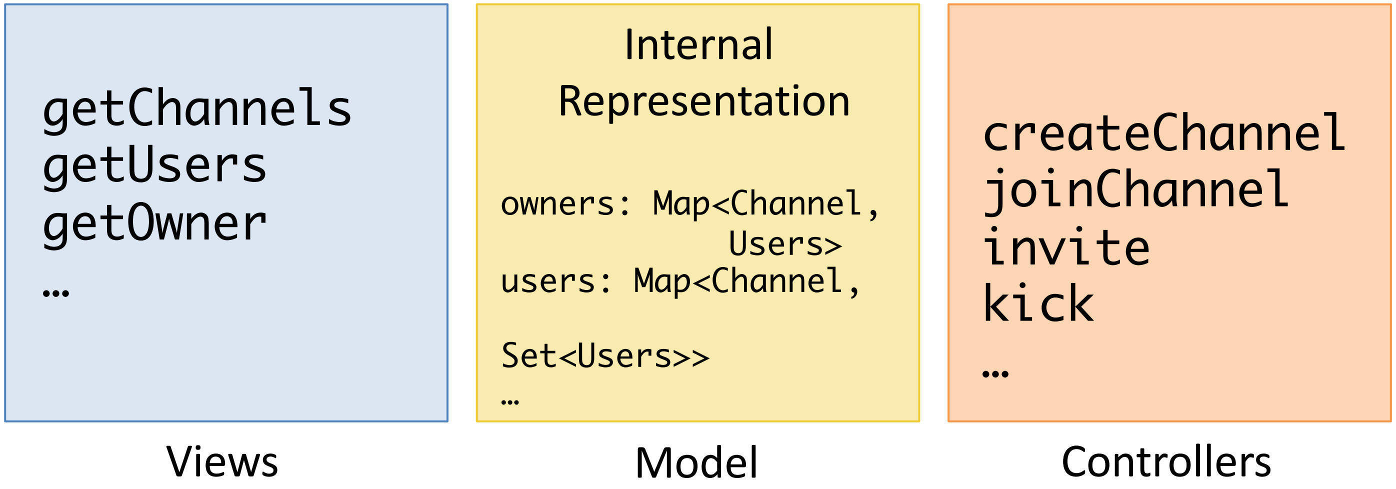
Model

Controllers

`Class JToggleButton.ToggleButtonModel`

`boolean isSelected()`	Checks if the button is selected.
`void setPressed(boolean b)`	Sets the pressed state of the button.
`void setSelected(boolean b)`	Sets the selected state of the button.

Example: Chat Server



```
getChannels
getUsers
getOwner
…
```

Internal Representation

```
owners: Map<Channel,
              Users>
users: Map<Channel,

Set<Users>>
…
```

```
createChannel
joinChannel
invite
kick
…
```

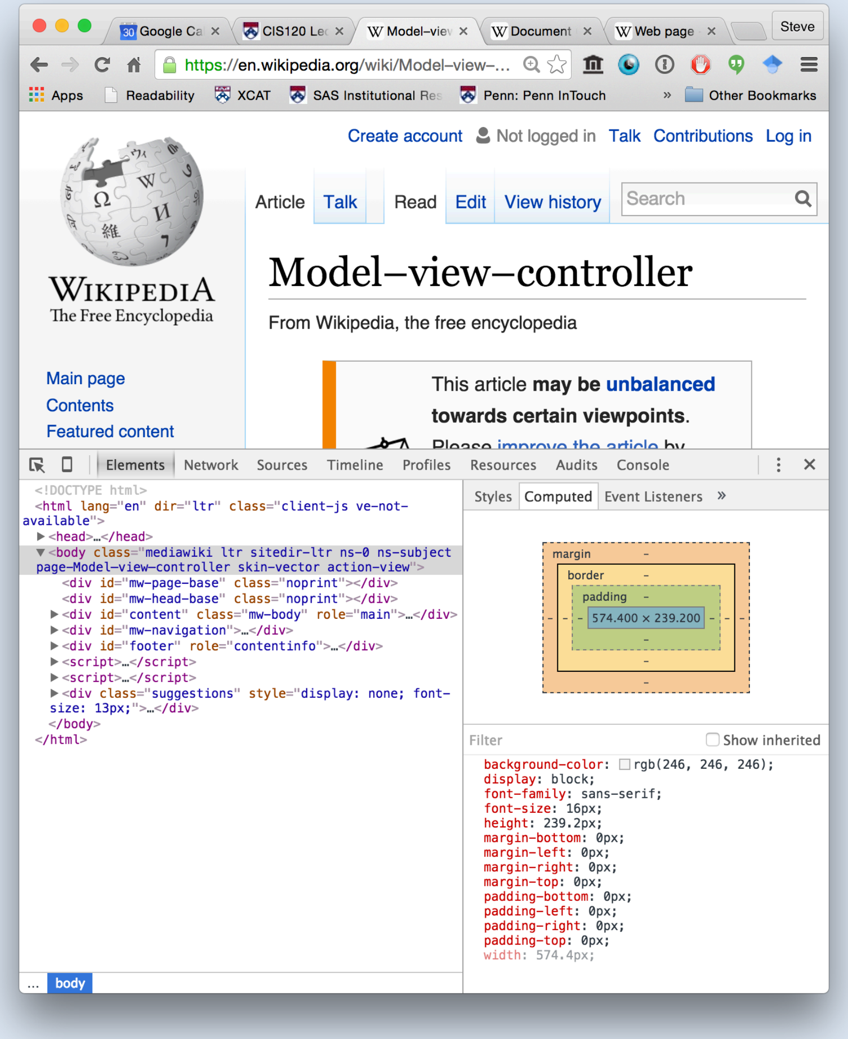
Views

Model

Controllers

`ServerModel`

Example: Web Pages



Model–view–controller

From Wikipedia, the free encyclopedia

This article **may be unbalanced towards certain viewpoints**. Please improve the article by

```
<!DOCTYPE html>
<html lang="en" dir="ltr" class="client-js ve-not-available">
 ▶<head>…</head>
 ▼<body class="mediawiki ltr sitedir-ltr ns-0 ns-subject page-Model-view-controller skin-vector action-view">
    <div id="mw-page-base" class="noprint"></div>
    <div id="mw-head-base" class="noprint"></div>
   ▶<div id="content" class="mw-body" role="main">…</div>
   ▶<div id="mw-navigation">…</div>
   ▶<div id="footer" role="contentinfo">…</div>
   ▶<script>…</script>
   ▶<script>…</script>
   ▶<div class="suggestions" style="display: none; font-size: 13px;">…</div>
  </body>
</html>
```

```
background-color: □rgb(246, 246, 246);
display: block;
font-family: sans-serif;
font-size: 16px;
height: 239.2px;
margin-bottom: 0px;
margin-left: 0px;
margin-right: 0px;
margin-top: 0px;
padding-bottom: 0px;
padding-left: 0px;
padding-right: 0px;
padding-top: 0px;
width: 574.4px;
```

Views

Internal Representation: DOM (Document Object Model)

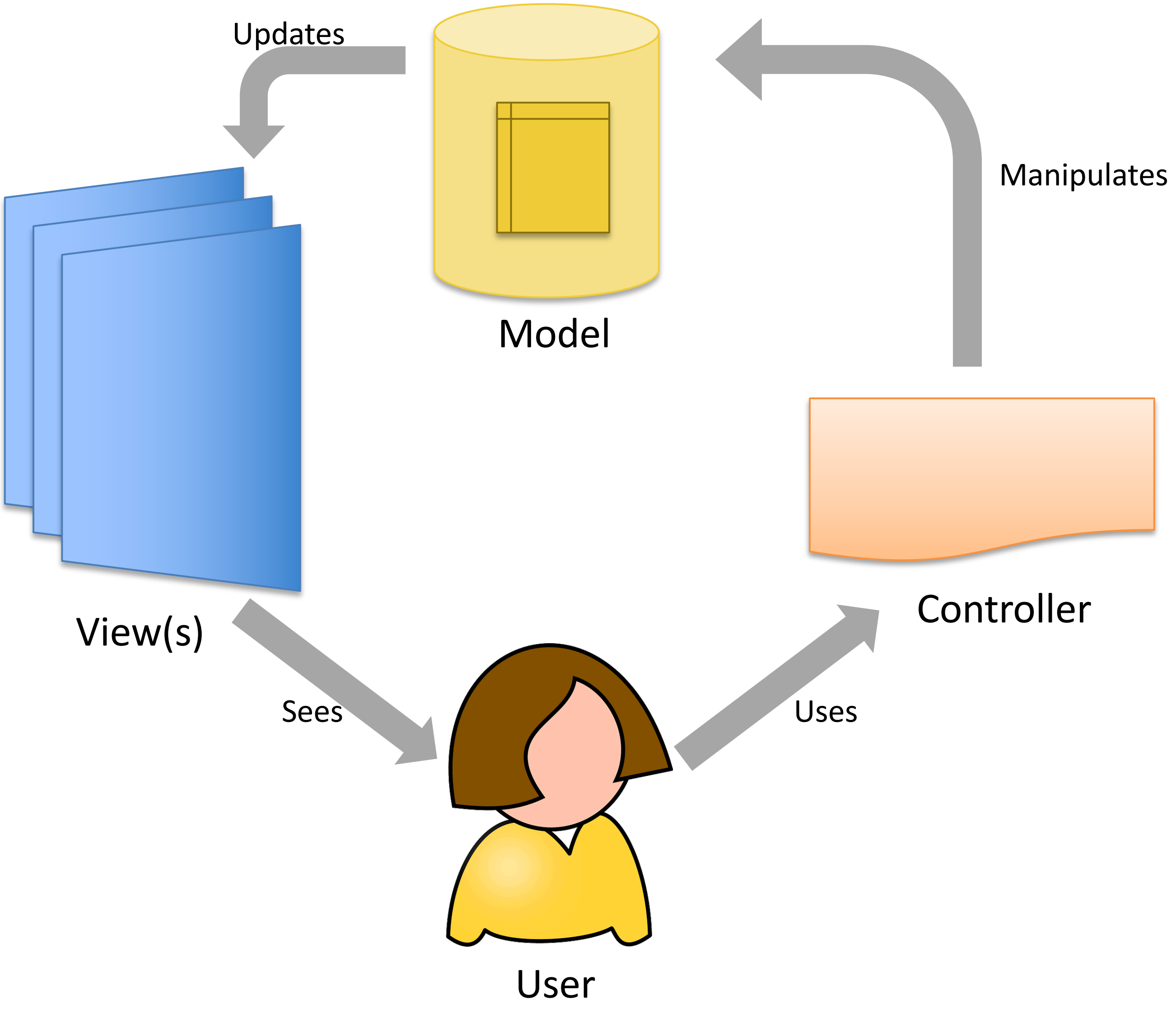
Model

JavaScript API

`document.addEventListener()`

Controllers

MVC Pattern


Updates
Model
Manipulates
View(s)
Controller
Sees
Uses
User

MVC Benefits?

- Decouples important "model state" from how that state is presented and manipulated
  - Suggests where to insert interfaces in the design
  - Makes the model testable independent of the GUI

- Multiple views
  - e.g. from two different angles, or for multiple different users

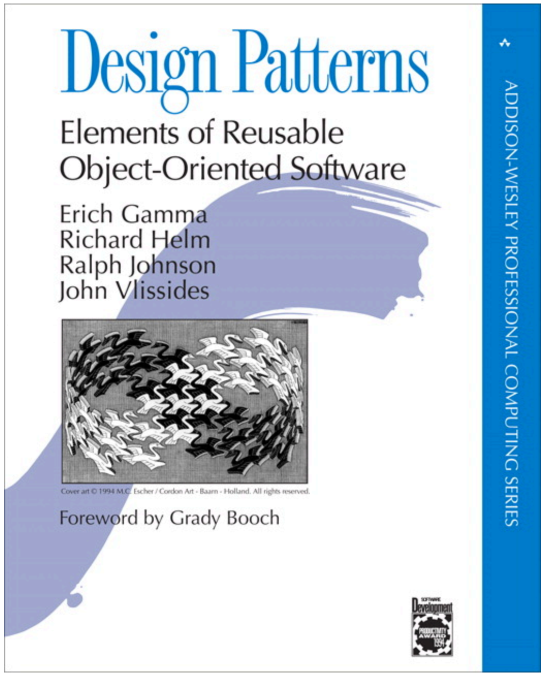
- Multiple controllers
  - e.g. mouse vs. keyboard interaction

MVC Variations

- Many variations on MVC pattern
- Hierarchical / Nested
  - As in the Swing libraries, in which JComponents often have a "model" and a "controller" part
- Coupling between Model / View or View / Controller
  - e.g. in MOD the Model and the View are coupled because the model carries most of the information about the view

Design Patterns

- Design Patterns
  - Influential OO design book published in 1994 (so a bit dated)
  - Identifies many common situations and "patterns" for implementing them in OO languages


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Some we have seen explicitly:
  - e.g. *Iterator* pattern
- Some we've used but not explicitly described:
  - e.g. The Broadcast class from the Chat HW uses the *Factory* pattern
- Some are workarounds for OO's lack of some features:
  - e.g. The *Visitor* pattern is like OCaml's fold + pattern matching