Programming Languages and Techniques (CIS1200)

Lecture 34

Swing II: Inner Classes, Layout, MoD

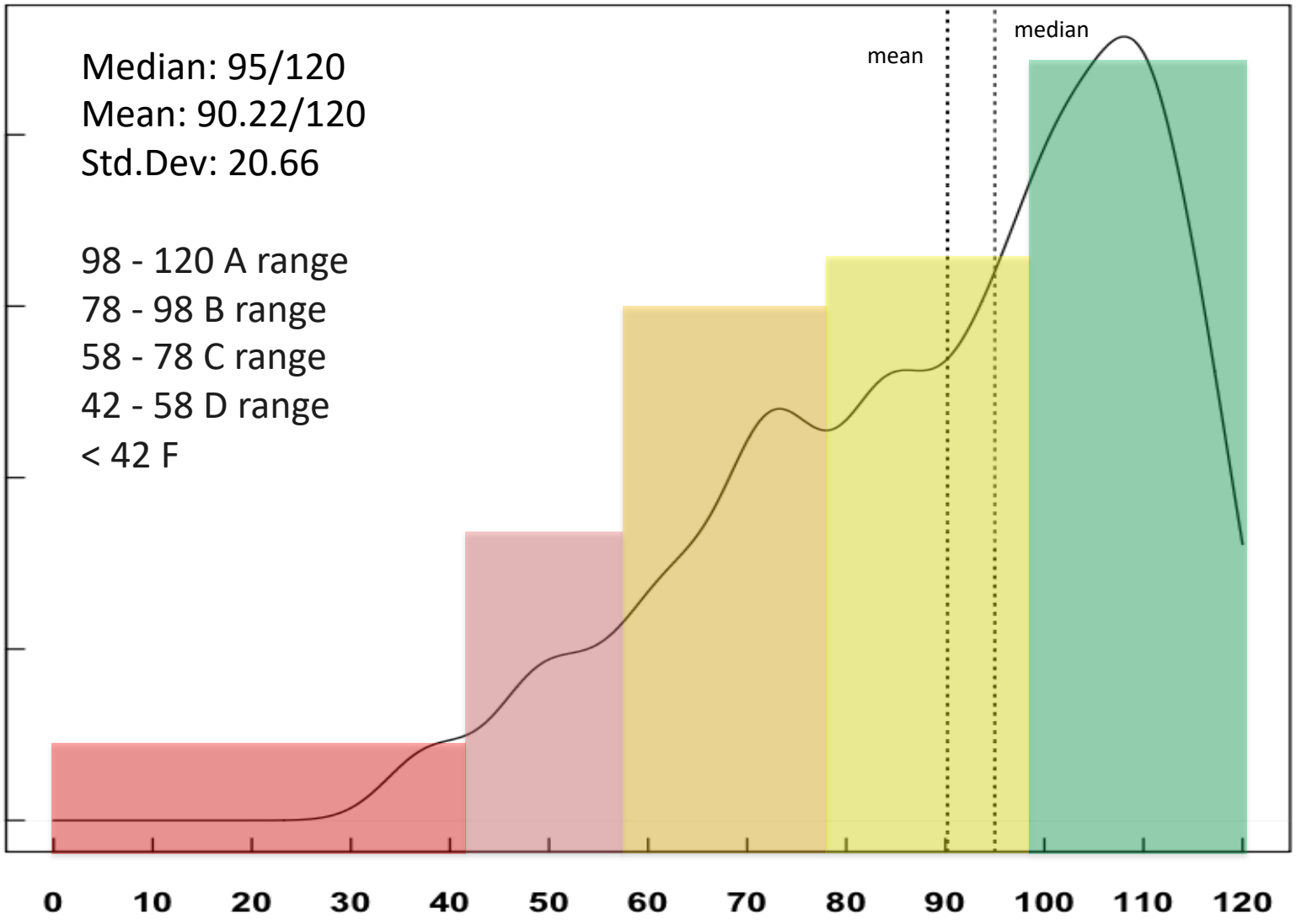
Chapters 29 and 30

Announcements (1)

- Midterm 2
  - Grades and solutions available
  - Regrade requests via Gradescope
    - Opens on Monday, November 25th
    - Due by Friday, December 6th
- HW08: TwitterBot*
  - Due on November 26th
  - Practice with I/O and Collections
- HW9: Game Project
  - TAs will give you feedback soon
  - Final Program Due: Monday, December 9th at 11:59pm
  - Grade based on demo with your TA during/after reading days
  - *NO LATE SUBMISSIONS PERMITTED*

* Maybe should be called "XBot" or "TheProjectFormerlyKnownAsTwitterBot"or ”AnywhereButTwitterBot” or…?

Midterm 2 Analysis



Announcements (2)

- Plans for the week of Thanksgiving
  - HW08 due on Tuesday at 11.59pm
  - No recitations that week
  - TA OH till Tuesday will be virtual
  - No OH from Wednesday to Sunday

  - Wednesday, November 27th – Bonus Lecture
    - Come to either lecture (10:15 or noon)
    - Material is not needed for HW or Exams
    - Should be fun!
    - (Will be recorded)

  - No lecture next Friday

Announcements (3)

- TA applications are still open
  - CIS 1100, 1200, 1600, 1210 (see https://tinyurl.com/2tn2t22f)
  - Other CIS and NETS classes (see https://www.cis.upenn.edu/ta-information/)
  - **Accepting applications until tonight**

Recap: Swing User Interaction

Start Simple: Light Switch

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 Lightbulb 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 right now! It is simply a notification to Swing that something needs repainting. (This is a difference from our OCaml GUI library.) But it is required.

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 instantiate a given Listener class in a GUI *exactly one time*

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

This is a job for…

Inner Classes

Inner Classes

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

- Replace 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;
    }
  }
}
```

The 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.

Inner class can refer to a to field bound in the outer class

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

0

new Outer.Inner(2)

0%

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

0%

new Inner(2)

0%

Outer.Inner.new (2)

0%

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 ();
}
```

I.e., `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

We can define a class and create an object from it *all at once* inside a method body

```
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 Classe

"Create an object by instantiating an anonymous class implementing the ActionListener interface, with a method actionPerformed…"

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

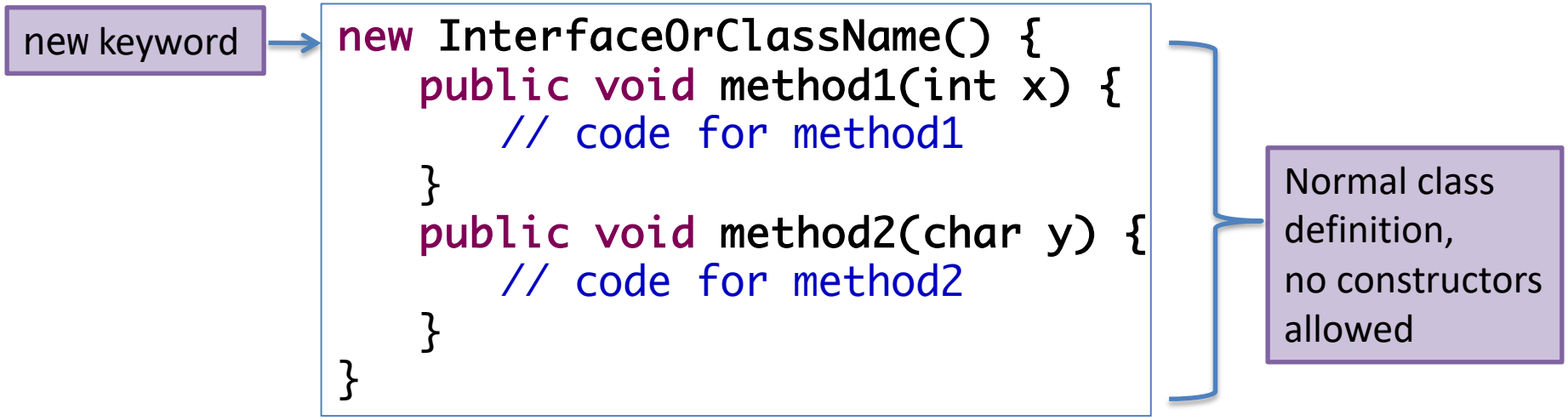
Puts button action with button definition

```
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



Static type of the expression is the interface / superclass named after the new

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
  - E.g., 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` (i.e., immutable)

But we can do even better…

“Lambdas” are Anonymous Inner Classes

- Often the implementation of an anonymous class is simple
  - e.g., an interface that contains only one method
- *Lambda* expressions*
  - treat functionality as method argument, or code as data
  - Java's version of first-class functions
- Pass functionality as an argument to another method,
  - e.g., what action should be taken when someone clicks a button.
- *Any* interface that has exactly one method can be implemented via a "lambda" (anonymous function).
  - Method’s "name" implicitly determined by the type at which the lambda is used
  - https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

*The term "lambda" comes from the *lambda calculus,* which was introduced by Alonzo Church in the 1930s. The lambda calculus forms the theoretical basis of all functional programming languages.

Lambda Expressions

- Java includes *lambda expressions,* which can implement classes that define only a single method

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

button.addActionListener((ActionEvent 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
```

Lambdas In A Nutshell

Lambda Notation

```
x -> x + x
```

"Ordinary" Java Notation

```
int method1(int x) {
  return x + x;
}
```

```
(x,y) -> x.m(y)
```

```
int method2(A x, B y) {
  return x.m(y);
}
```

```
(x,y) -> {
  System.out.println(x);
  System.out.println(y);
}
```

```
void method3(String x,
             String y) {
  System.out.println(x);
  System.out.println(y);
}
```

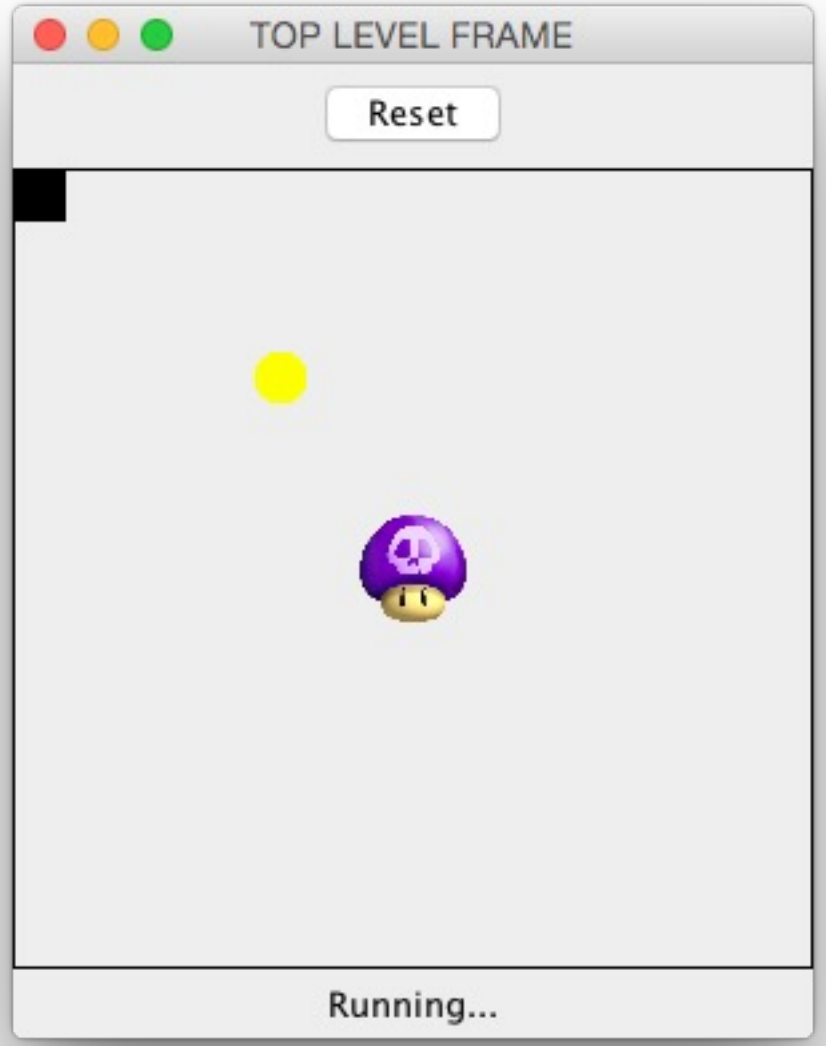
Method names and types are inferred from the context.

Swing Layout Demo

LayoutDemo.java

Mushroom of Doom

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


TOP LEVEL FRAME
Reset
Running...

JPanel
(control_panel)

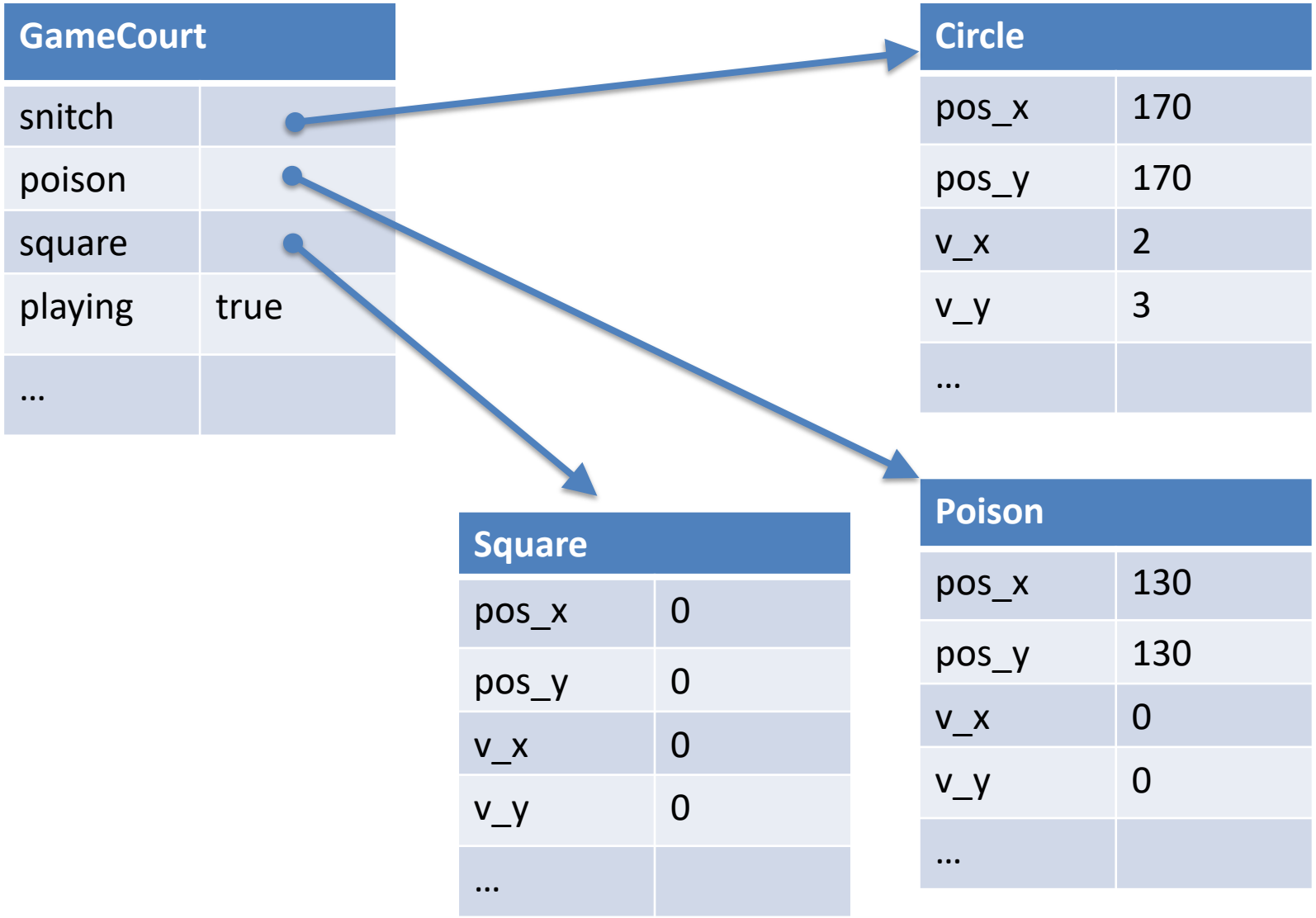
GameCourt,
subclass of
JPanel
(court)

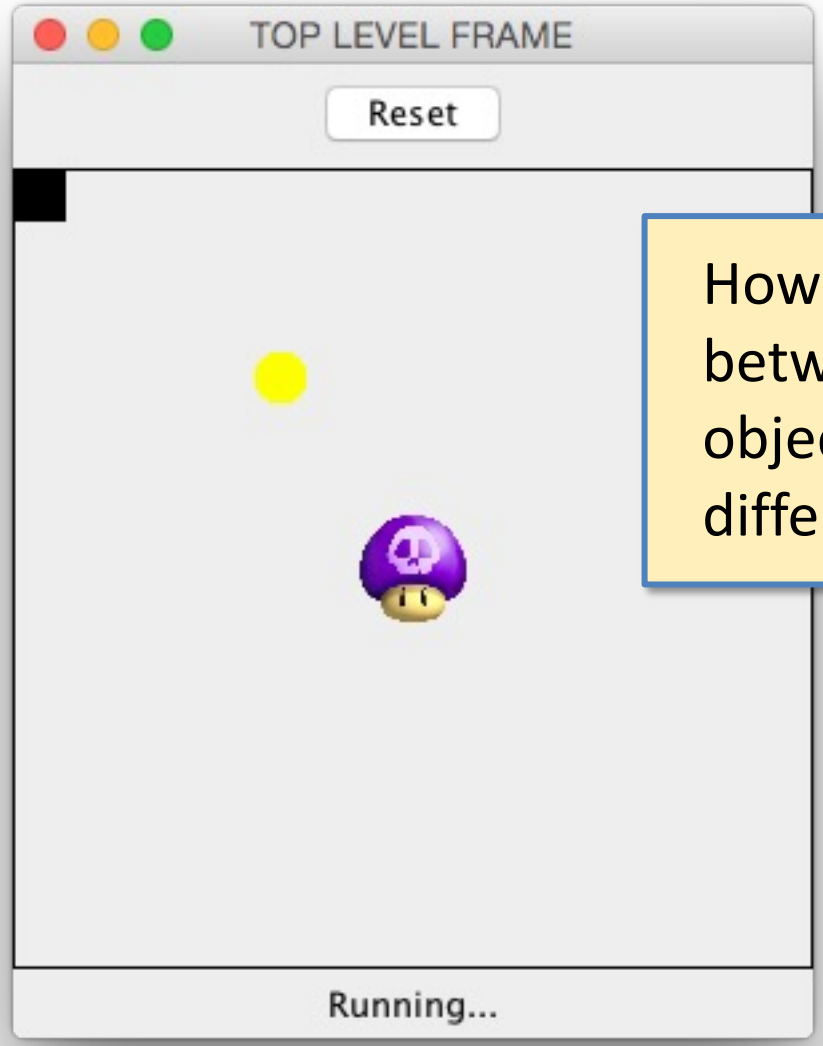
JButton
(reset)

JPanel
(status_panel)

JLabel (status)

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?

Abstract Classes

- An abstract class provides an *incomplete* implementation:
  - some methods are marked as abstract
  - those methods must be overridden to create instances

```
public abstract class AbstractClass {
    private int x = 0;
    public int m() {
        return frob(frob(x));
    }
    abstract int frob(int x);
}

class ConcreteClass extends AbstractClass {
    @Override
    int frob(int x) {
        return x * 120;
    }
}
```

Keyword "abstract" marks methods without implementations.

A subclass overrides the abstract method with an implementation.

36: It is possible to fill in __??__ with code so that, when run, the variable ac will contain an object of type AbstractClass.

0

True

0%

False

0%

```
public abstract class AbstractClass {
    private int x = 0;
    public int m() {
        return frob(frob(x));
    }
    abstract int frob(int x);
}

// somewhere in main:
Abstract Class ac = new AbstractClass __??__;
```

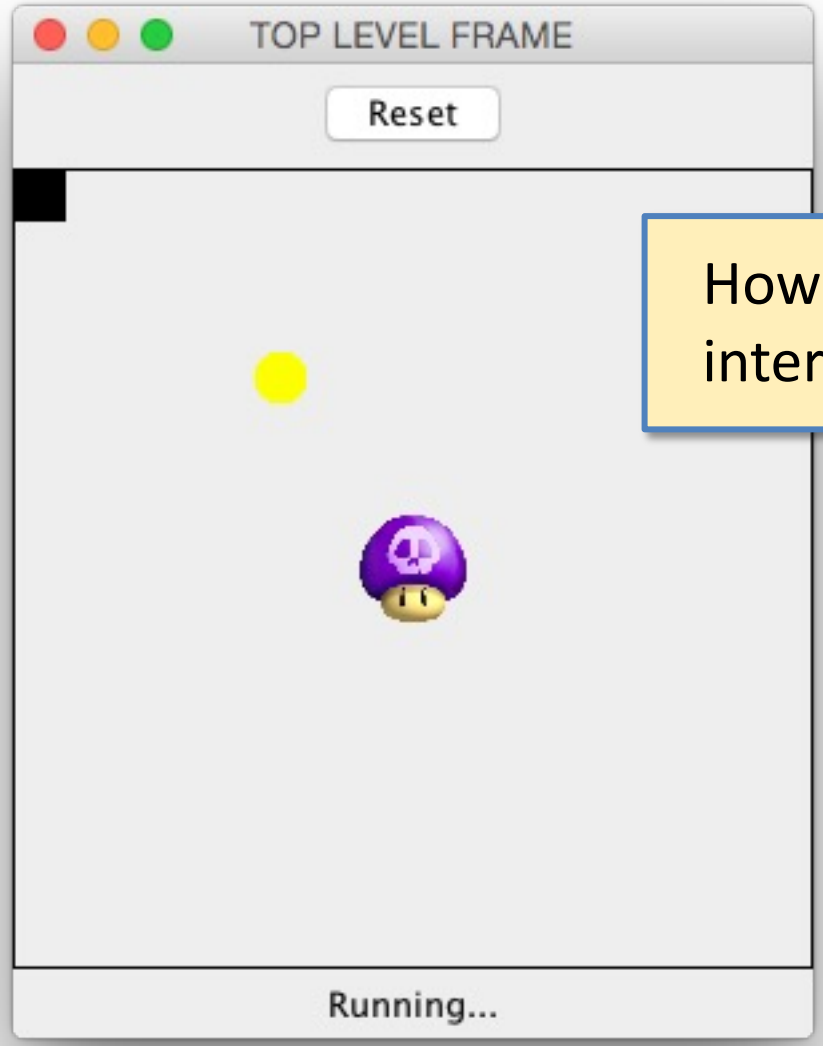
True or False: It is possible to fill in the hole marked __??__ so that, when run, the variable ac will contain a new object of type AbstractClass.

```
public abstract class AbstractClass {
    private int x = 0;
    public int m() {
        return frob(frob(x));
    }
    abstract int frob(int x);
}

// somewhere in main:
AbstractClass ac = new AbstractClass () {
    @Override
    int frob(int x) { return 0; }
};
```

Answer: True – use an anonymous inner class!

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();
  }
}
```



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