Programming Languages and Techniques (CIS1200)

Lecture 33

Swing I: Drawing and Event Handling

Chapter 29

Announcements (1)

- Midterm 2
  - Grades and solutions will be posted by Friday
- HW08: TwitterBot*
  - Due on November 26th
  - Practice with I/O and Collections

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

HW9: Game project

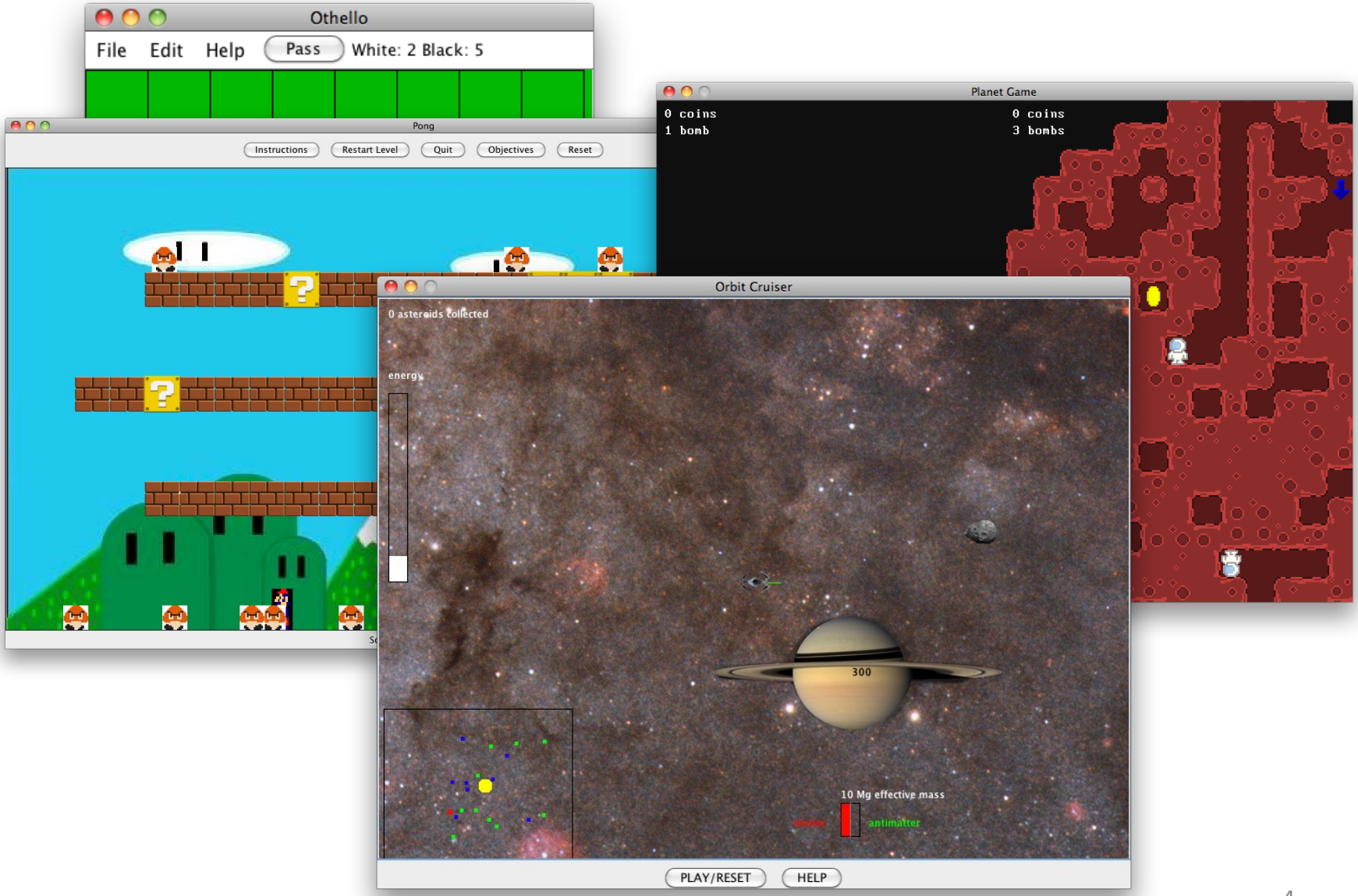
- Game Design Proposal Milestone Due: (8 points)
  Tomorrow at Midnight = 11:59PM!
  - (Should take about 1 hour)
  - Submit on GRADESCOPE
  - TAs will give you feedback soon

- Final Game Due: (92 points)
  Monday, December 9th at 11:59pm
  - Submit zipfile online, submission *only* checks if your code compiles
  - IntelliJ is **strongly recommended** for this project
  - You may distribute your game (after the deadline) if you do not use any of our code

- Grade based on demo with your TA during/after reading days
  - Grading rubric on the assignment website
  - Recommendation: don't be too ambitious.

- *NO LATE SUBMISSIONS CAN BE ACCEPTED*

HW9: Game Project



Announcements (3)

- 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
    - Material is not needed for HW or Exams
    - Should be fun!
    - (Will be recorded)
  - No lecture on Friday

Announcements (4)

- 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 Friday, November 22nd**
  - Intro CIS TA Panel
    - Recording should be available

Swing

Java's GUI library

33: Have you ever used the Swing library to build a Java app before?

0

Nope

0%

No, but I've used a different GUI library in Java

0%

Yes, but I didn't really understand how it worked

0%

Yes, I'm an expert

0%

Why study GUIs (again)?

- Most common example of *event-based programming*
- Heavy (and effective) use of OO inheritance
- Case study in library organization
  - and some advanced Java features
- Ideas applicable everywhere:
  - Web apps
  - Mobile apps
  - Desktop apps
- Fun?

Terminology overview

	GUI Library (OCaml)	Swing (Java)
Graphics Context	`Gctx.gctx`	`Graphics, Graphics2D`
Widget type	`Widget.widget`	`JComponent`

Basic Widgets	`button`
	`label`
	`checkbox`
		`JButton`
		`JLabel`
		`JCheckBox`

Container Widgets	`hpair, vpair`	`JPanel, Layouts`

Events	`event`	`ActionEvent`
		`MouseEvent`
		`KeyEvent`

Event Listener	`mouse_listener`
	`mouseclick_listener`
	(any function of type `event -> unit`)
		`ActionListener`
		`MouseListener`
		`KeyListener`

Swing practicalities

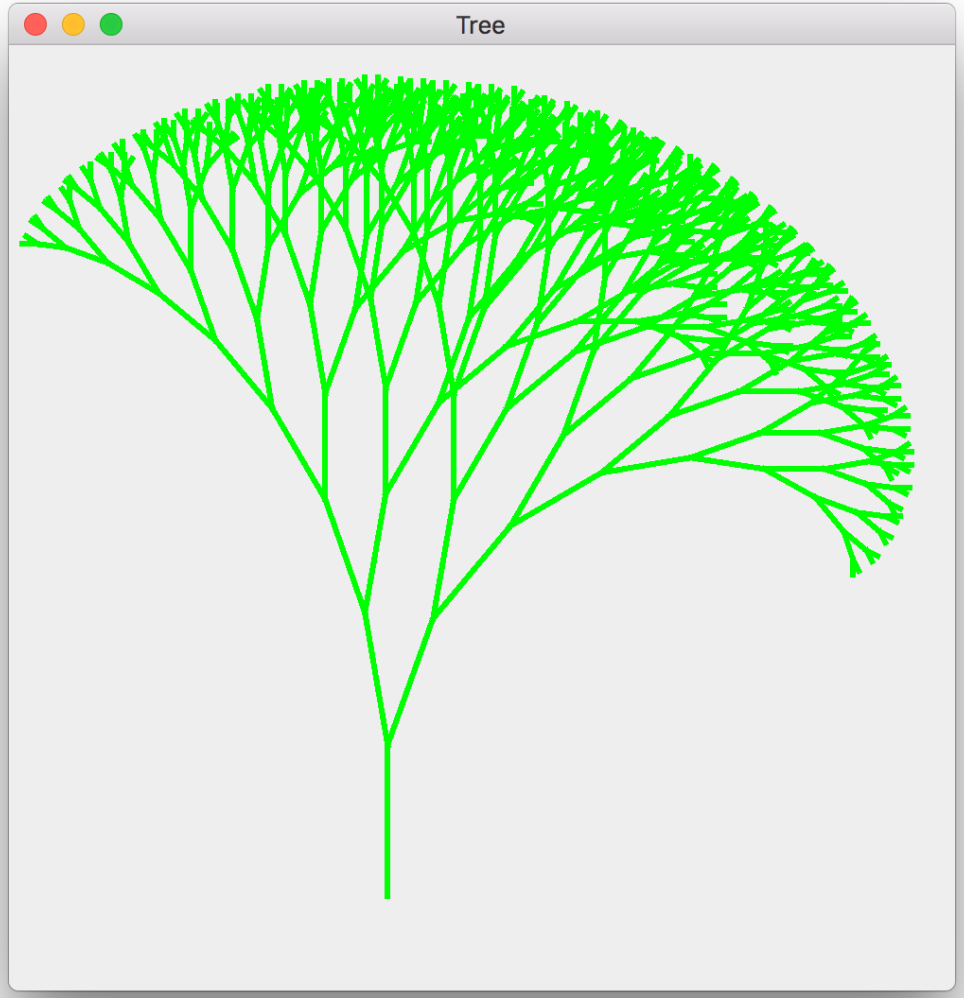
- Java library for GUI development
  - javax.swing.*
- Built on older library: AWT
  - java.awt.*
  - When there are two versions of something, use Swing's. (e.g., javax.swing.JButton instead of java.awt.Button)
    - The "JFoo" version is usually the one you want, not plain "Foo"
- Portable
  - Communicates with underlying OS's native window system
  - Same Java program looks appropriately different when run in the browser and on PC, Linux, Mac, etc.

Simple Drawing

DrawingCanvas.java

DrawingCanvasMain.java

Fractal Drawing Demo



How do we draw a picture?

- In the OCaml GUI HW, we created widgets whose **`repaint`** function used the graphics context to draw an image

```
let w_draw : widget =
{
   repaint = (fun (gc:gctx) ->
                fractal (with_color gc green)
                        200 450 270 80) ;

   size    = (fun () -> (200,200));

   handle  = (fun () -> ())
}
```

OCaml

- In Swing, the preferred idiom is to *extend* the JComponent class …

Fundamental Swing Class: JComponent

- Analog of *widget type* from OCaml GUI project
- Subclasses should *override* methods of JComponent
  - paintComponent (like repaint: displays the component)
  - getPreferredSize (like size: calculates the size of the component)
- Events are handled by *listeners*
  - no need for overriding here
- Rich functionality
  - minimum/maximum size
  - font
  - foreground/background color
  - borders
  - visibility
  - much more…

Step 1: Recursive function for drawing

```
private static void fractal(Graphics gc, int x, int y,
          double angle, double len) {

    if (len > 1) {
        double af = (angle * Math.PI) / 180.0;
        int nx = x + (int)(len * Math.cos(af));
        int ny = y + (int)(len * Math.sin(af));
        gc.drawLine(x, y, nx, ny);
        fractal(gc, nx, ny, angle + 20, len - 8);
        fractal(gc, nx, ny, angle - 10, len - 8);
    }
}
```

How do we turn this into a GUI component?

Step 2: Simple Drawing Component

```
public class DrawingCanvas extends JComponent {

    // paint the drawing panel on the screen
    public void paintComponent (Graphics gc) {
        super.paintComponent(gc);

        // set the pen color
        gc.setColor(Color.GREEN);
        ((Graphics2d)gc).setStroke(new BasicStroke(3));

        // draw a fractal tree
        fractal(gc, 200, 450, 270, 80);
    }

    // give the size of the drawing panel
    public Dimension getPreferredSize() {
        return new Dimension(200,200);
    }
}
```

How do we put this component on the screen?

Step 3: JFrame

- Represents a top-level window
  - Displayed directly by OS (looks different on Mac, PC, etc.)
- Contains JComponents
- Can be moved, resized, iconified, closed

```
public void run() {
    JFrame frame = new JFrame("Tree");

    // set the content of the window to be our drawing
    frame.getContentPane().add(new DrawingCanvas());

    // make sure the application exits when the frame closes
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // resize the frame based on the size of the panel
    frame.pack();

    // show the frame
    frame.setVisible(true);
}
```

(Plus a bit of boilerplate to call `run` from `main`…)

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

- 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);
   }
 });
```

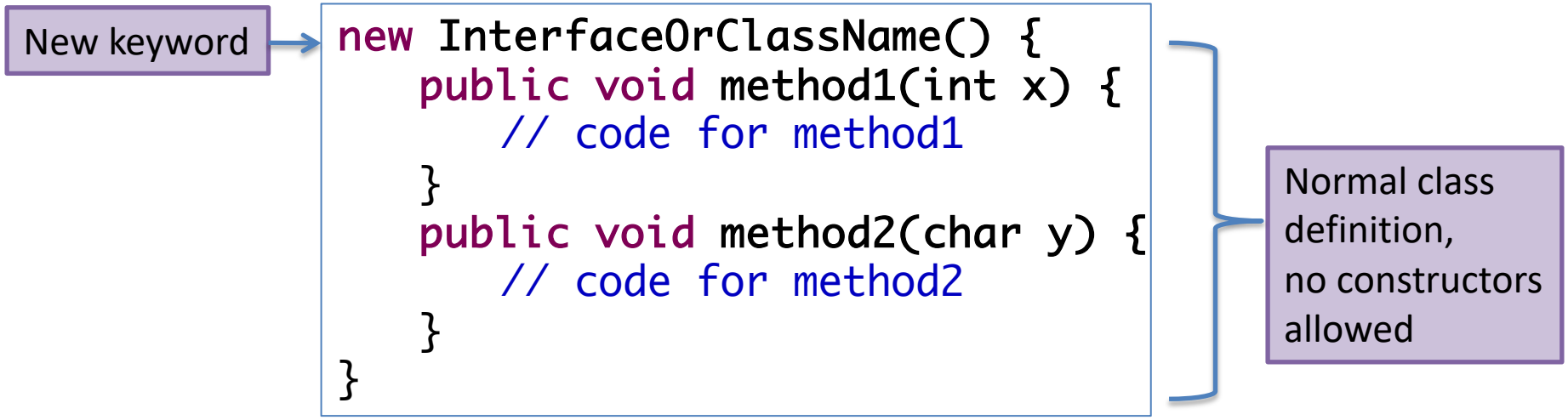
Puts button action with button definition

```
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



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`

Lambdas are Anonymous Inner Classes

- Often implementation of anonymous classes 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 "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
```

Java Lambda 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.