Objectives

- GUIs in Java
  - Event handling

Assignment 8 Questions?

Roulette Review

- A capstone project for this part of the course
- Bringing together:
  - Applying design principles
  - Testing
    - Non-deterministic
- Need to understand these well to move to bigger problems/code bases

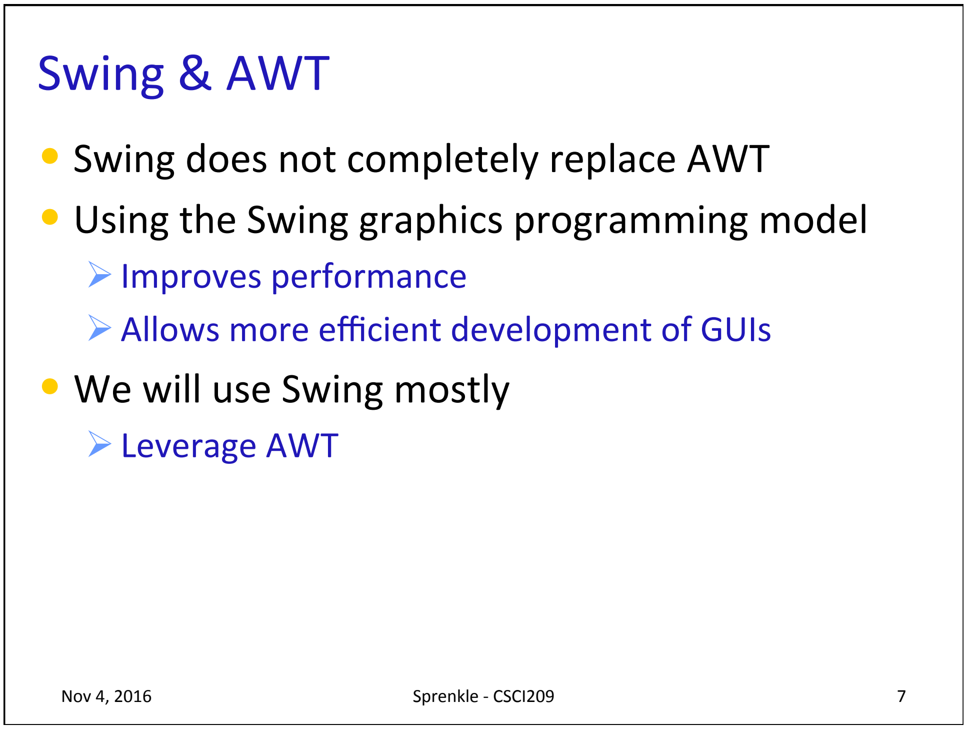
GUIS IN JAVA

GUIs Review

- What are the two packages used for GUI development in Java?
- What is a component vs what is a container?
- What are some of the classes involved in GUI programs?

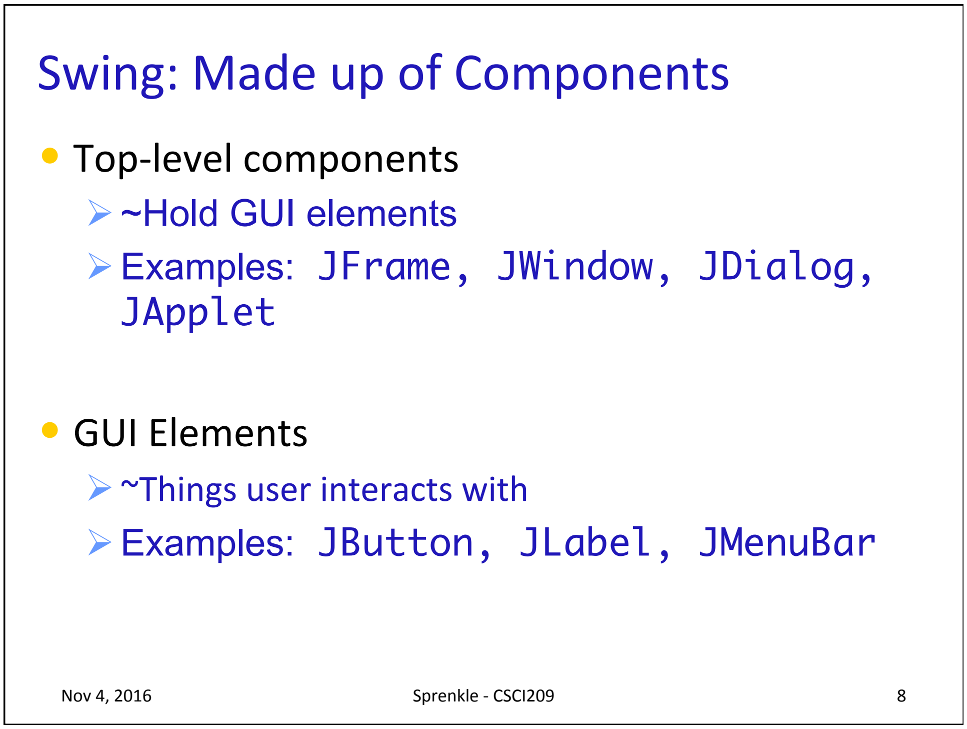
Java GUI Libraries: AWT & Swing

- AWT: Abstract Windowing Toolkit
  - Original GUI toolkit
  - Relies on operating system to render GUIs
    - Benefit: Match look and feel of platform
  - Classes in `java.awt.*`
- Swing: added to Java2
  - Classes in `javax.swing.*`
  - Extends AWT
  - Provides *Java* look and feel for applications
    - But can plug in other look & feels



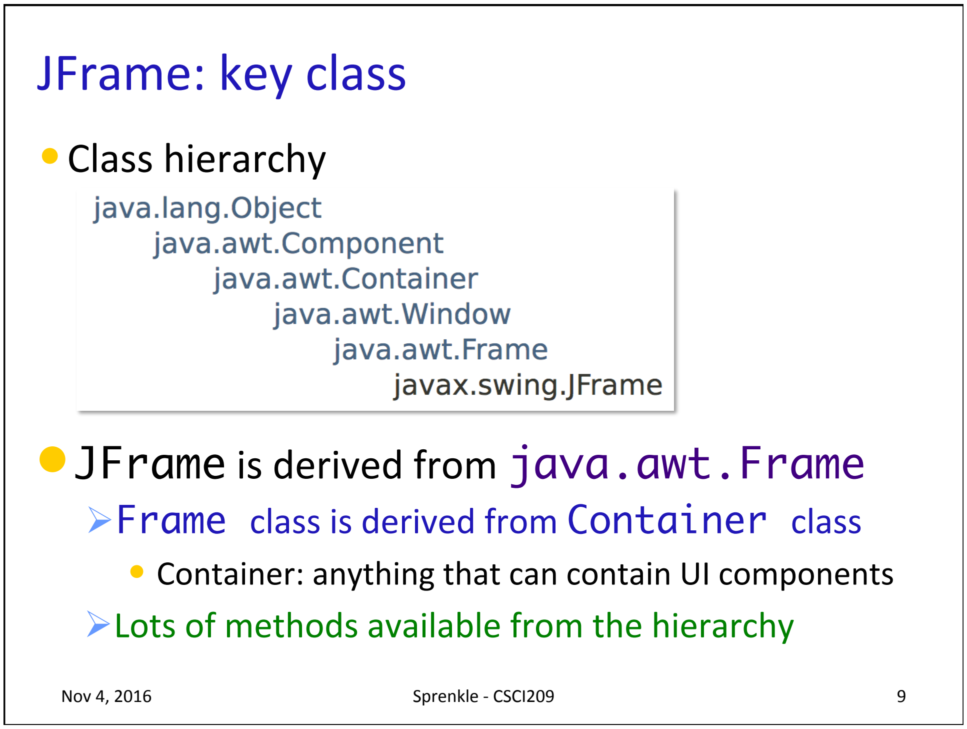
Swing & AWT

- Swing does not completely replace AWT
- Using the Swing graphics programming model
  - Improves performance
  - Allows more efficient development of GUIs
- We will use Swing mostly
  - Leverage AWT



Swing: Made up of Components

- Top-level components
  - ~Hold GUI elements
  - Examples: `JFrame, JWindow, JDialog, JApplet`
- GUI Elements
  - ~Things user interacts with
  - Examples: `JButton, JLabel, JMenuBar`

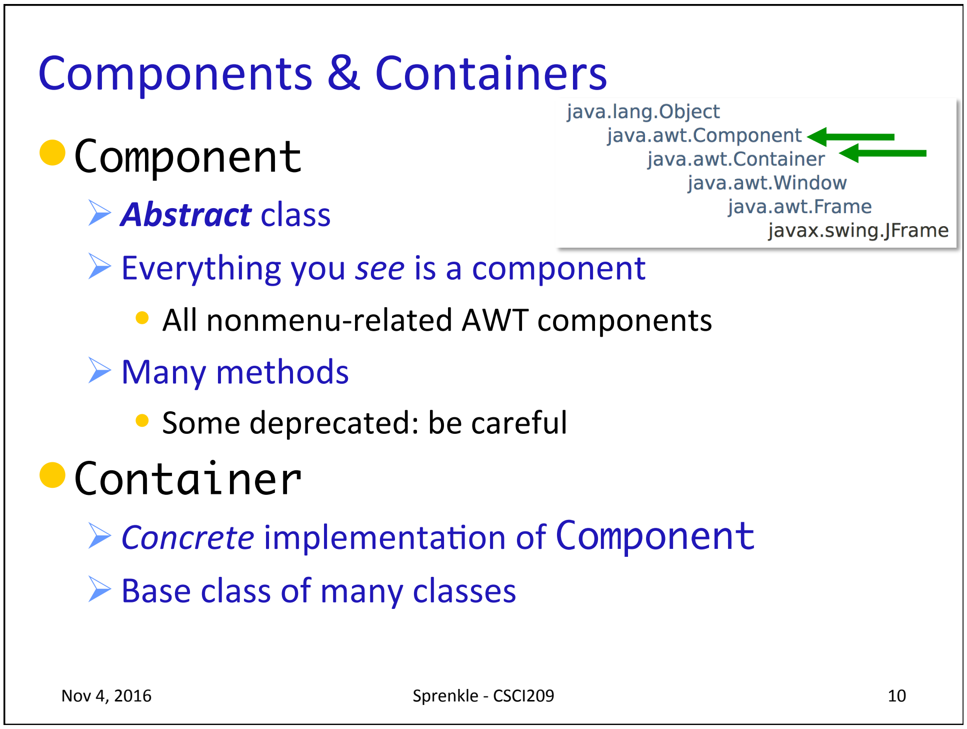


JFrame: key class

- Class hierarchy

```
java.lang.Object
     java.awt.Component
          java.awt.Container
               java.awt.Window
                    java.awt.Frame
                         javax.swing.JFrame
```

- `JFrame` is derived from `java.awt.Frame`
  - `Frame` class is derived from `Container` class
    - Container: anything that can contain UI components
  - Lots of methods available from the hierarchy

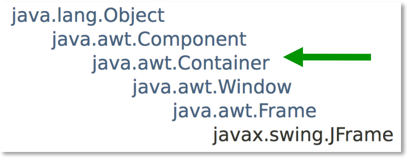


Components & Containers

```
java.lang.Object
     java.awt.Component ←
          java.awt.Container ←
               java.awt.Window
                    java.awt.Frame
                         javax.swing.JFrame
```

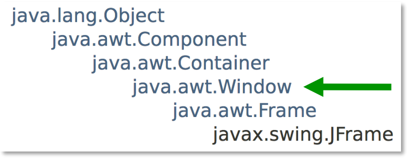
- Component
  - ***Abstract*** class
  - Everything you *see* is a component
    - All nonmenu-related AWT components
  - Many methods
    - Some deprecated: be careful
- `Container`
  - *Concrete* implementation of Component
  - Base class of many classes

Container Methods


java.lang.Object
java.awt.Component
java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame

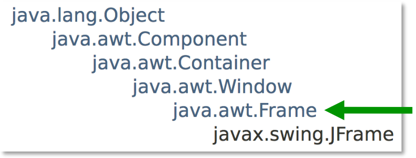
- `add(Component c)`
- `setSize`
  - Sets size of frame in *pixels*
- `setLocation`
  - Sets location of frame
    - Coordinates of top-left corner
- `setBounds`
  - Sets both size and location of frame
    - Provides information needed for `setSize` and `setLocation`

Window Methods


java.lang.Object
java.awt.Component
java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame

- Top-level window
- No borders
- No Menu Bar
- `dispose()`
  - Closes window and reclaims resources associated with it
- `toBack()`
  - Sends window to back, may lose focus/activation
- `toFront()`
  - Bring to front, make this the focused window

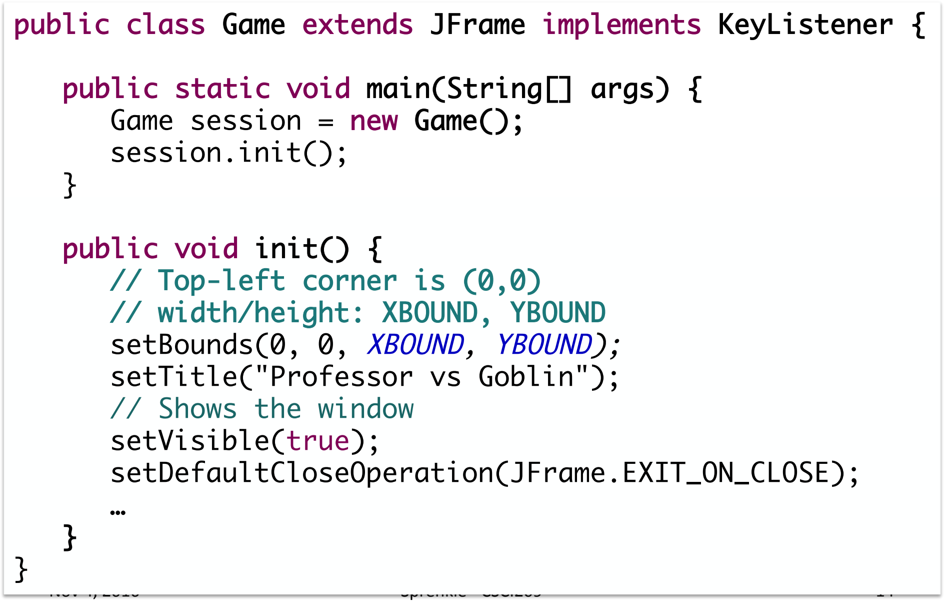
Frame's Methods



```
java.lang.Object
    java.awt.Component
        java.awt.Container
            java.awt.Window
                java.awt.Frame
                    javax.swing.JFrame
```

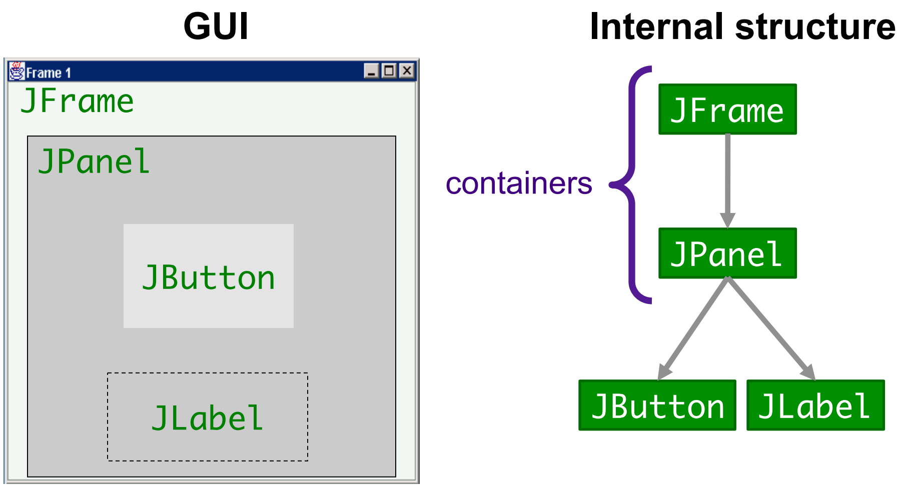
- Top-level window *with title and borders*
- `setTitle(String title)`
  - Sets title of frame (displayed in title bar)
- `setResizable(boolean resizable)`
  - Can the user resize the frame?

Frame: Key Class



```java
public class Game extends JFrame implements KeyListener {

    public static void main(String[] args) {
        Game session = new Game();
        session.init();
    }

    public void init() {
        // Top-left corner is (0,0)
        // width/height: XBOUND, YBOUND
        setBounds(0, 0, XBOUND, YBOUND);
        setTitle("Professor vs Goblin");
        // Shows the window
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        …
    }
}
```

Anatomy of an Application GUI



GUI

Internal structure

JFrame

JPanel

JButton

JLabel

containers

JFrame

JPanel

JButton

JLabel

Implementing a GUI Component

1. Create it
2. Configure it
3. Add children (if container)
4. Add to parent (if not JFrame)
5. Listen to it

order important

Implementing a GUI Component

1. Create it

```
JButton b = new JButton();
```

2. Configure it

```
b.setText("press me");
b.setForeground(Color.blue);
```

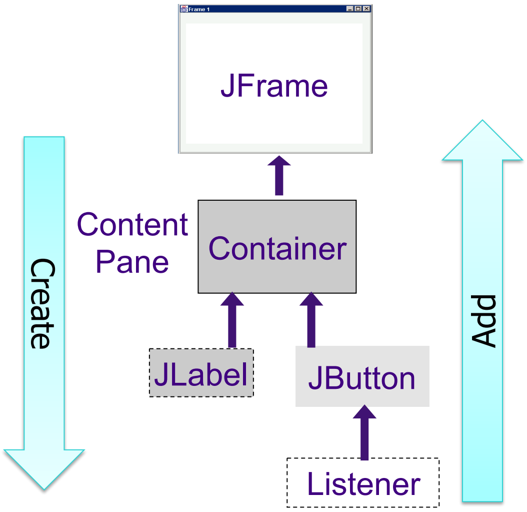
3. Add it to parent

```
panel.add(b);
```

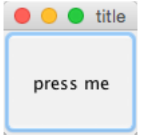
4. Listen to it
   - Events: Listeners

Building a GUI

1. Create (top down):
   - Frame
   - Container
   - Components
   - Listeners
2. Add (bottom up):
   - Listeners into components
   - Components into panel
   - Panel into frame



Example Code



```
// create the components
JFrame f = new JFrame("title");
f.setBounds(0, 0, 100, 100);
Container pane = f.getContentPane();
```

JFrame contains a ContentPane, a Container that holds UI components

```
JButton b = new JButton("press me");
// add button to panel
pane.add(b);
// show the frame
f.setVisible(true);
```

Practice: Combining Components

- Create a frame whose panel has three buttons on it
- What is our process?

ButtonPanel.java

Placement of Components

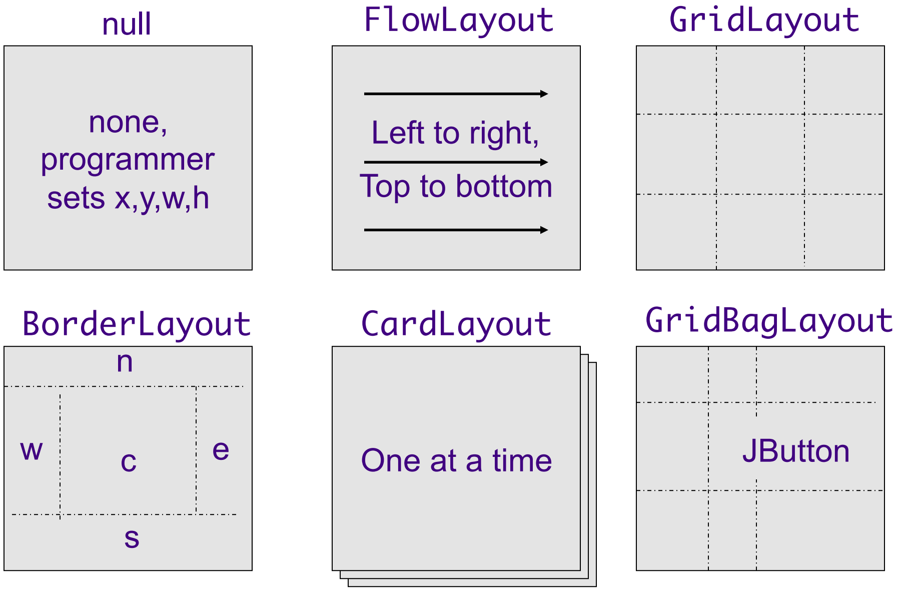
- How does the panel know where to place a button?
- How does the panel know where to place the next button?
- How does the panel know where to place *any* component that is added to it?

LAYOUT MANAGERS

Layout Managers

- Java uses ***layout managers*** to place components inside a container
- `LayoutManager` automatically handles placement of components
  - When a component is added to a container (through `add`), layout manager decides where to place the component

Layout Manager Heuristics



Default Layout Managers

- JFrame’s content pane: BorderLayout
- JPanel’s: FlowLayout
  - Commonly used container

Changing Layout Managers

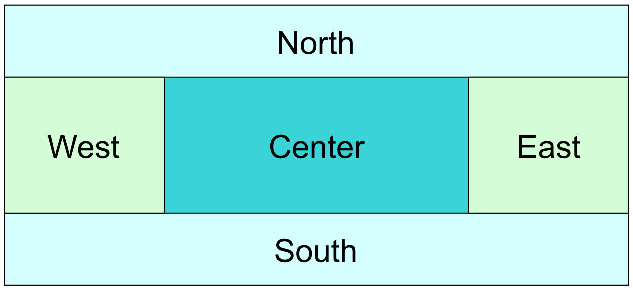
- Any container can use any layout manager
- Use setLayout to change layout manager *before adding components*

```
// sets layout to a new flow layout manager that
// aligns row components to the left and uses a 20 pixel
// horizontal separation and 20 pixel vertical separation
setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));

// sets layout to a new border layout manager that
// uses a 45 pixel horizontal separation between
// components (regions) and a 20 pixel vertical separation
setLayout(new BorderLayout(45, 20));
```

Border Layout Manager

- Default layout manager of the content pane for `JFrame`
- Lets you choose where you want to place each component

with respect to the container



- Edge components are laid out first
- Center occupies remaining space

Adding Components Using a Border Layout

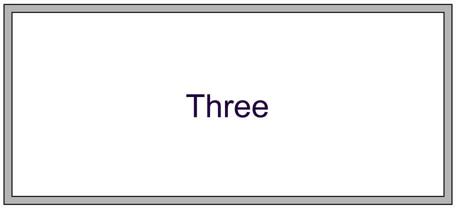
```
Container contentPane = getContentPane();
contentPane.add(button, BorderLayout.SOUTH);
```

- If no region specified, assumes center region

What happens if we add multiple components, e.g., three buttons, without specifying a region?

➢ Recall: border layout grows component to fit specified region

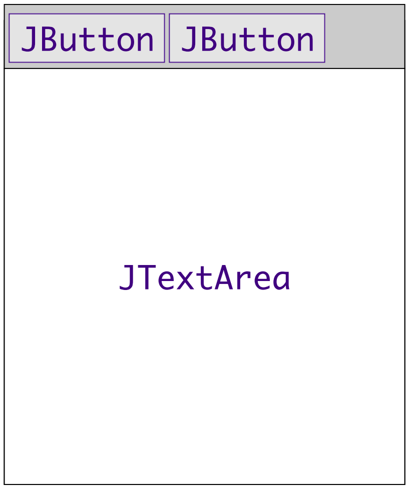
A Border Layout Limitation



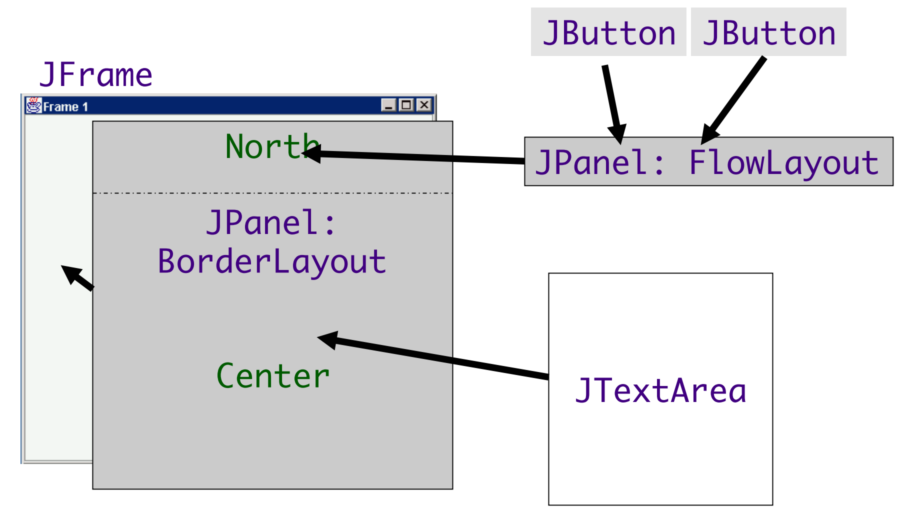
- Last button added grows to completely fill center region
- Explains our previous example!
  - First two buttons were discarded/overwritten by each subsequently added component

Combining Panels

- **Panels** act as (smaller) containers for UI elements
- Can be arranged inside a larger panel by a layout manager
- Use additional panels to customize look
  - Create a panel
  - Add some buttons to it
  - Add that panel to a region in content pan



Combining Panels



JButton JButton

JFrame

North

JPanel: FlowLayout

JPanel: BorderLayout

Center

JTextArea

Using Additional Panels

- Get fairly accurate and precise placement of components
- Use nested panels with

Layout	Use
BorderLayout	Content panes and enclosing panels
Flow Layouts	Panels containing buttons and other UI components

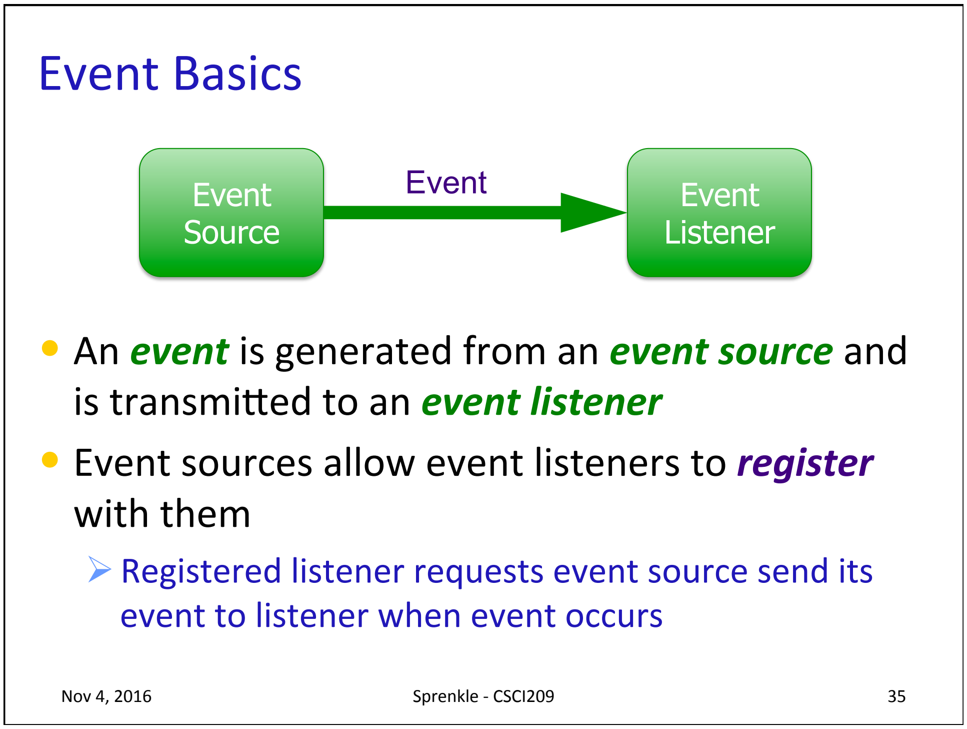
FlexibleLayout.java

HANDLING USER INTERACTIONS

Event-Driven Programming

- User actions (e.g., mouse clicks, key presses), sensor outputs, or messages from other applications determine flow of program
- Application architecture:

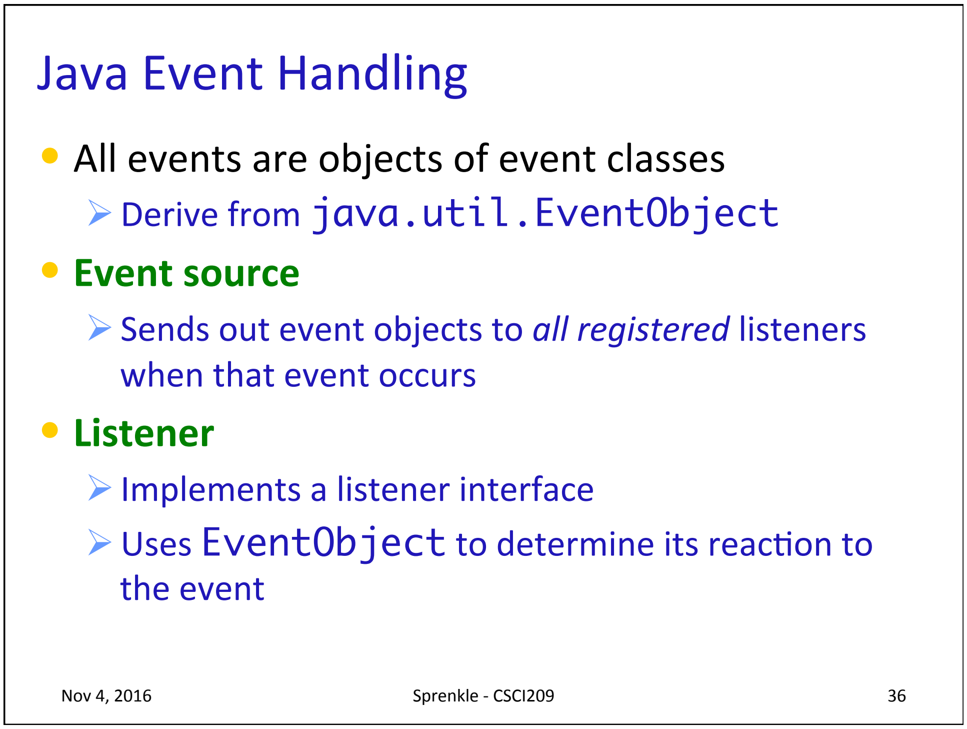
```
while ( true ) {
      event = waitForEvent();
      handleEvent(event);
}
```



Event Basics

Event Source → Event → Event Listener

- An ***event*** is generated from an ***event source*** and is transmitted to an ***event listener***
- Event sources allow event listeners to ***register*** with them
  - Registered listener requests event source send its event to listener when event occurs



Java Event Handling

- All events are objects of event classes
  - Derive from `java.util.EventObject`
- **Event source**
  - Sends out event objects to *all registered* listeners when that event occurs
- **Listener**
  - Implements a listener interface
  - Uses `EventObject` to determine its reaction to the event

Java Event Handling

- Register a listener with an event source:

```
eventSourceObject.addEventListener(
        eventListenerObject);
```

- Example:

```
ActionListener listener = . . .;
JButton button = new JButton("Click Me!");
button.addActionListener(listener);
```

  - ➢ Whenever an “action event” occurs on `button`, `listener` is notified
    - For buttons, an action event is a button click

Listener Objects

- A listener object must be an instance of a class that implements the appropriate interface
  - ➢ For buttons, that’s **`ActionListener`**
- Listener class must implement `actionPerformed(ActionEvent event)`

Listener Objects and Event Handling

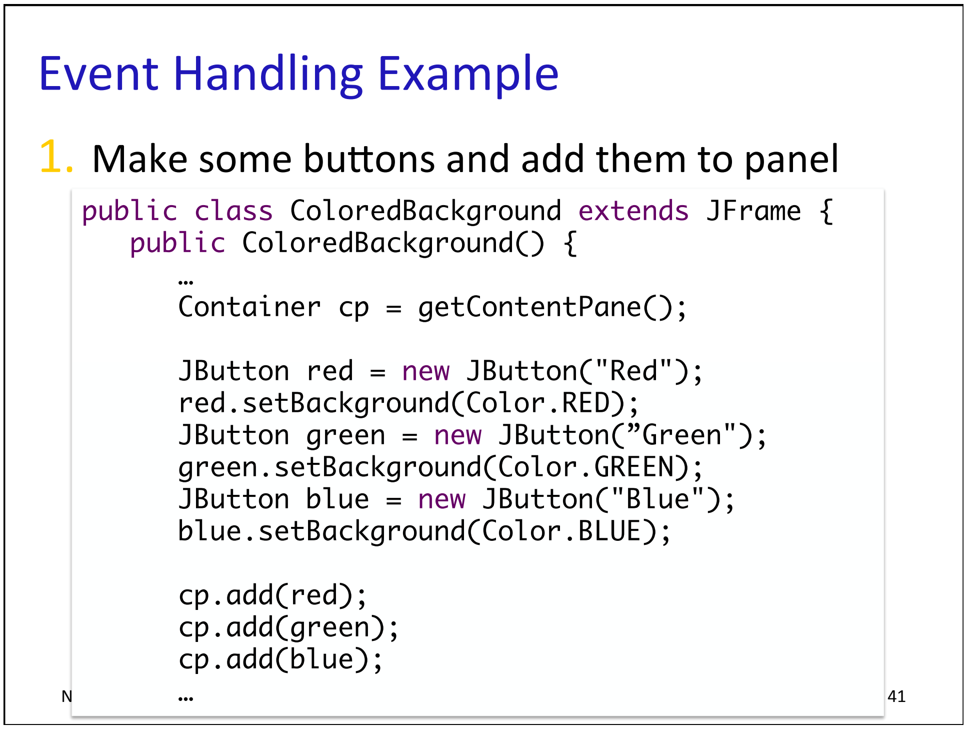
- When a user clicks a button, `JButton` object generates an `ActionEvent` object

Which makes `JButton` a *what*?

- `JButton` calls listener object's `actionPerformed` method, passing generated event object
- A single event source can have *multiple listeners* listening for its events
  - Source calls `actionPerformed` on each of its listeners

An Example of Event Handling

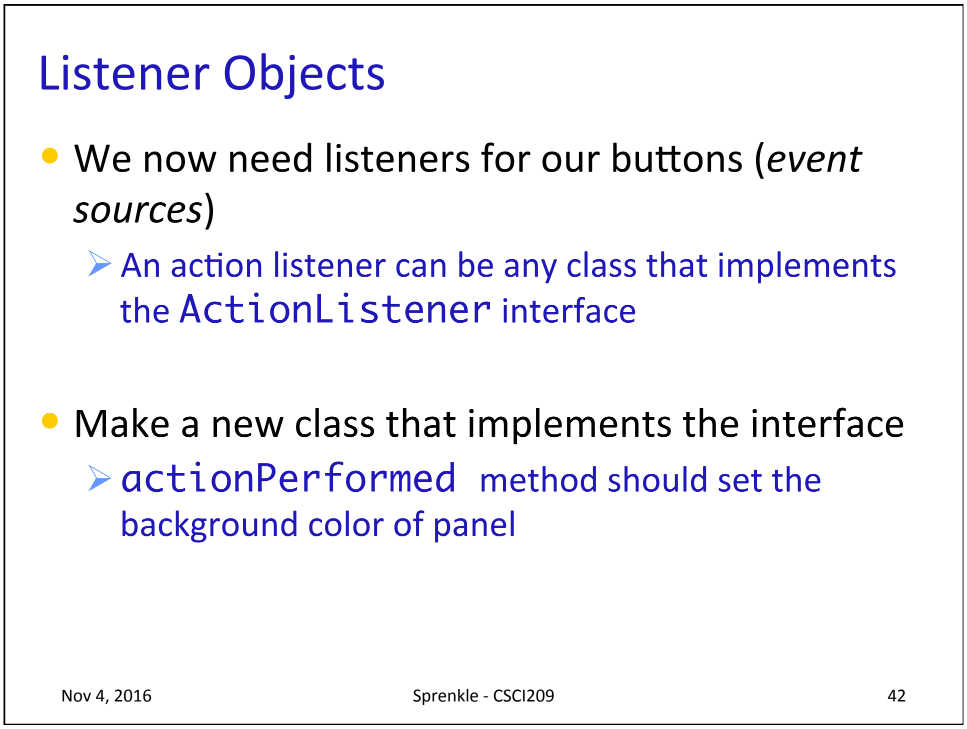
- Suppose we want to make a panel that has three buttons on it
  - Each button has a color associated with it
  - When user clicks a button, background color of panel changes to the corresponding color
- We need
  1. A panel with 3 buttons on it
  2. 3 listener objects, each registered to listen for a button's events



Event Handling Example

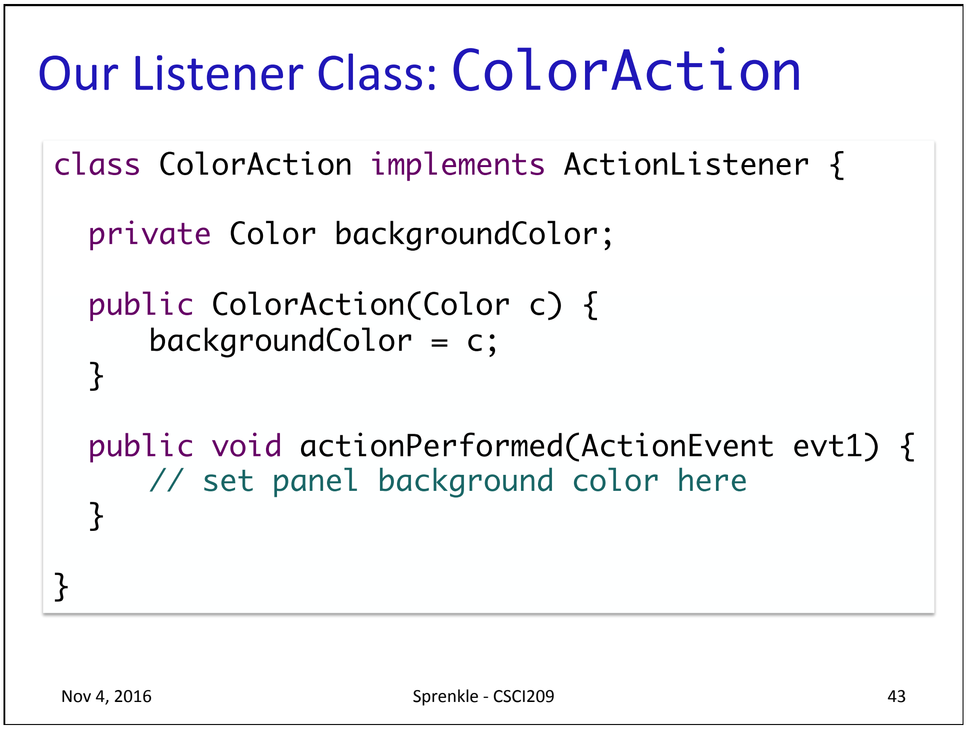
1. Make some buttons and add them to panel

```
public class ColoredBackground extends JFrame {
   public ColoredBackground() {
      …
      Container cp = getContentPane();

      JButton red = new JButton("Red");
      red.setBackground(Color.RED);
      JButton green = new JButton(”Green");
      green.setBackground(Color.GREEN);
      JButton blue = new JButton("Blue");
      blue.setBackground(Color.BLUE);

      cp.add(red);
      cp.add(green);
      cp.add(blue);
      …
```



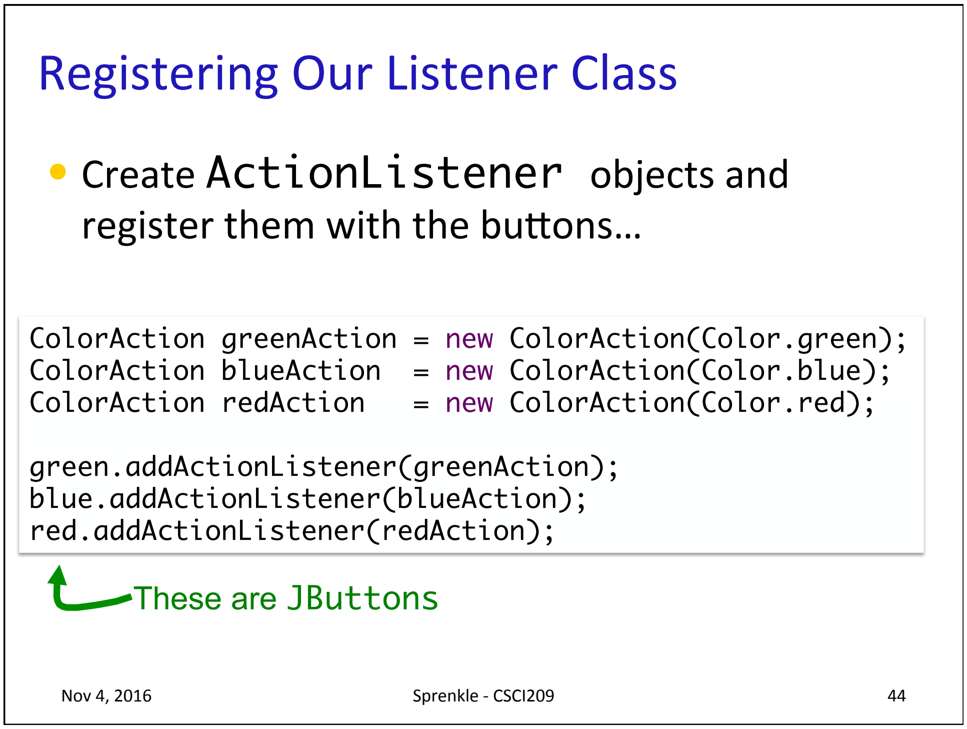
Listener Objects

- We now need listeners for our buttons (*event sources*)
  - An action listener can be any class that implements the `ActionListener` interface
- Make a new class that implements the interface
  - `actionPerformed` method should set the background color of panel



Our Listener Class: ColorAction

```
class ColorAction implements ActionListener {

  private Color backgroundColor;

  public ColorAction(Color c) {
      backgroundColor = c;
  }

  public void actionPerformed(ActionEvent evt1) {
      // set panel background color here
  }

}
```



Registering Our Listener Class

- Create ActionListener objects and register them with the buttons…

```
ColorAction greenAction = new ColorAction(Color.green);
ColorAction blueAction  = new ColorAction(Color.blue);
ColorAction redAction   = new ColorAction(Color.red);

green.addActionListener(greenAction);
blue.addActionListener(blueAction);
red.addActionListener(redAction);
```

These are JButtons

Registering Our Listener Class

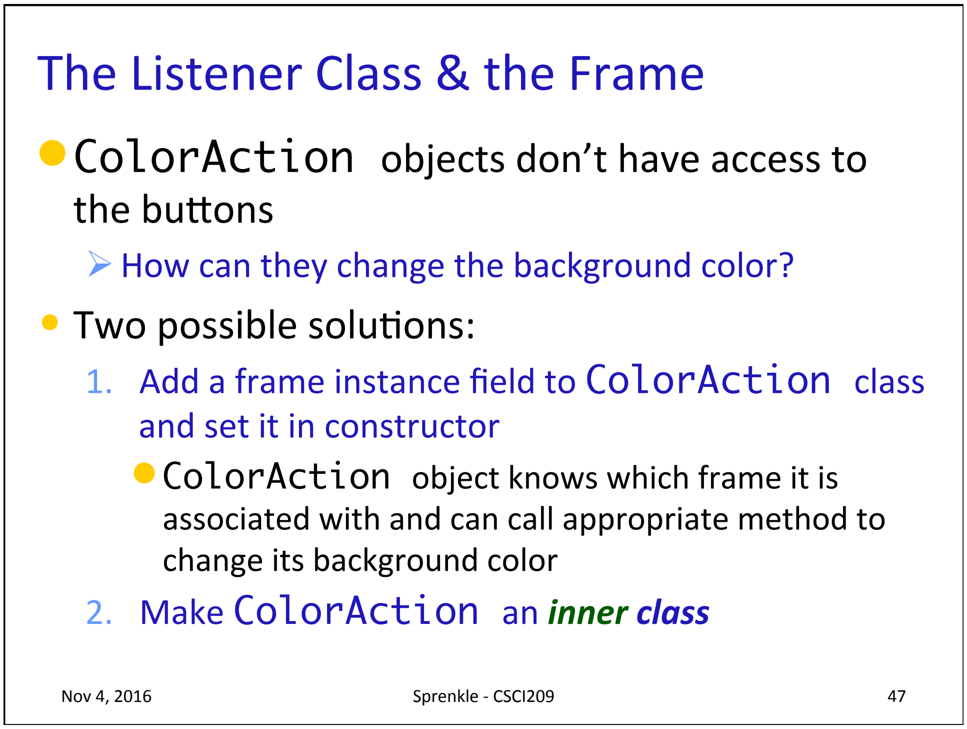
- When a user clicks the green button, the `JButton` generates an `ActionEvent`
  - Passes the `ActionEvent` object to `greenAction's` `actionPerformed` method
  - Method can then set frame's background color

```
class ColorAction implements ActionListener {
  private Color backgroundColor;
  public ColorAction(Color c) {
       backgroundColor = c;
  }
  public void actionPerformed(ActionEvent evt1) {
       // set panel background color here
       . . .
  }
}
```

How can we implement this?

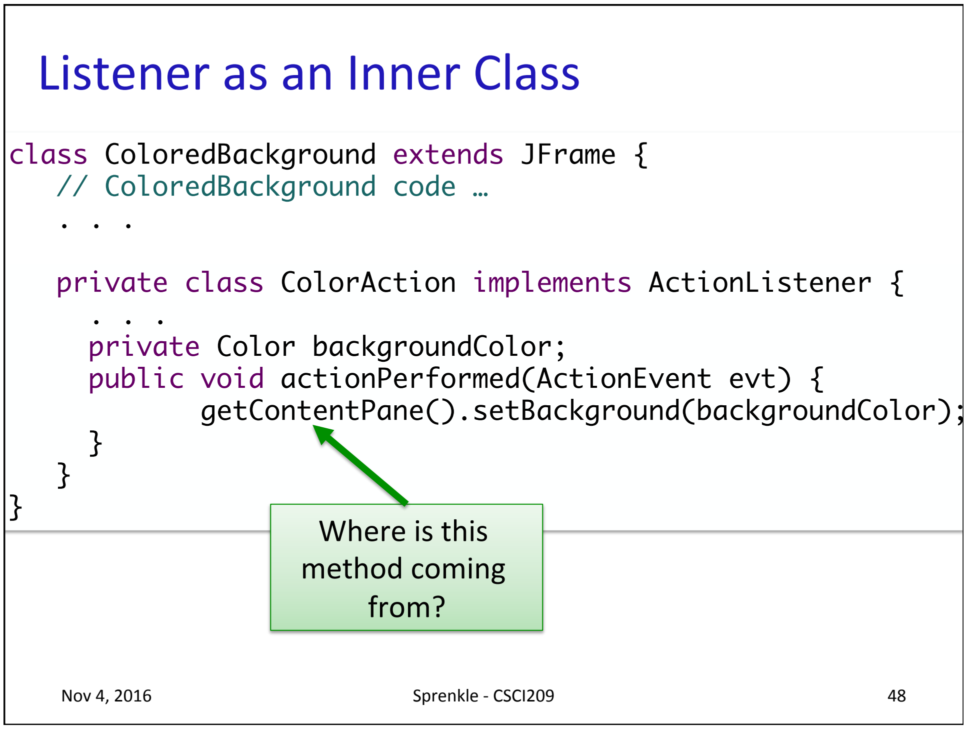
The Listener Class & the Frame

- `ColorAction` objects don't have access to the frame
  - How can they change the frame's background color?
- Possible solutions?



The Listener Class & the Frame

- ColorAction objects don't have access to the buttons
  - How can they change the background color?
- Two possible solutions:
  1. Add a frame instance field to ColorAction class and set it in constructor
     - ColorAction object knows which frame it is associated with and can call appropriate method to change its background color
  2. Make ColorAction an ***inner class***



Listener as an Inner Class

```
class ColoredBackground extends JFrame {
   // ColoredBackground code …
   . . .

   private class ColorAction implements ActionListener {
     . . .
     private Color backgroundColor;
     public void actionPerformed(ActionEvent evt) {
            getContentPane().setBackground(backgroundColor);
     }
   }
}
```

Where is this method coming from?

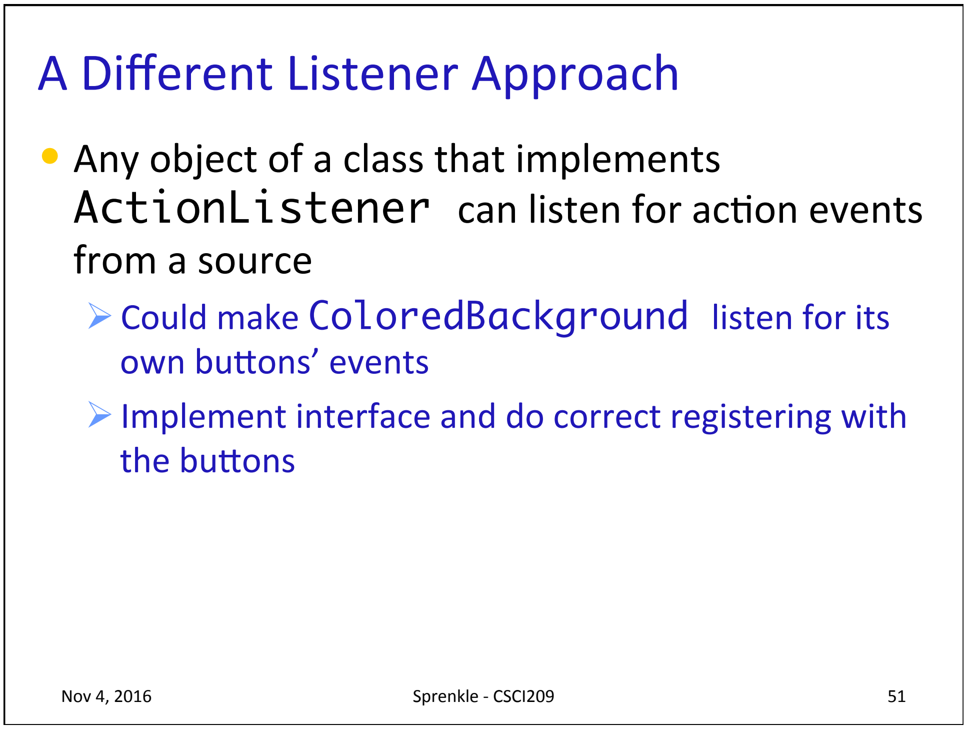
Close Up: actionPerformed()

```
public void actionPerformed(ActionEvent evt) {
        getContentPane().setBackground(backgroundColor);
}
```

- ColorAction does not have getContentPane() method
- Since ColorAction is an ***inner class*** of ColoredBackground, ColorAction can *directly access* ColoredBackground's instance fields and methods
  - Inner class calls outer class's method
    - Parameter: inner's private data (backgroundColor)

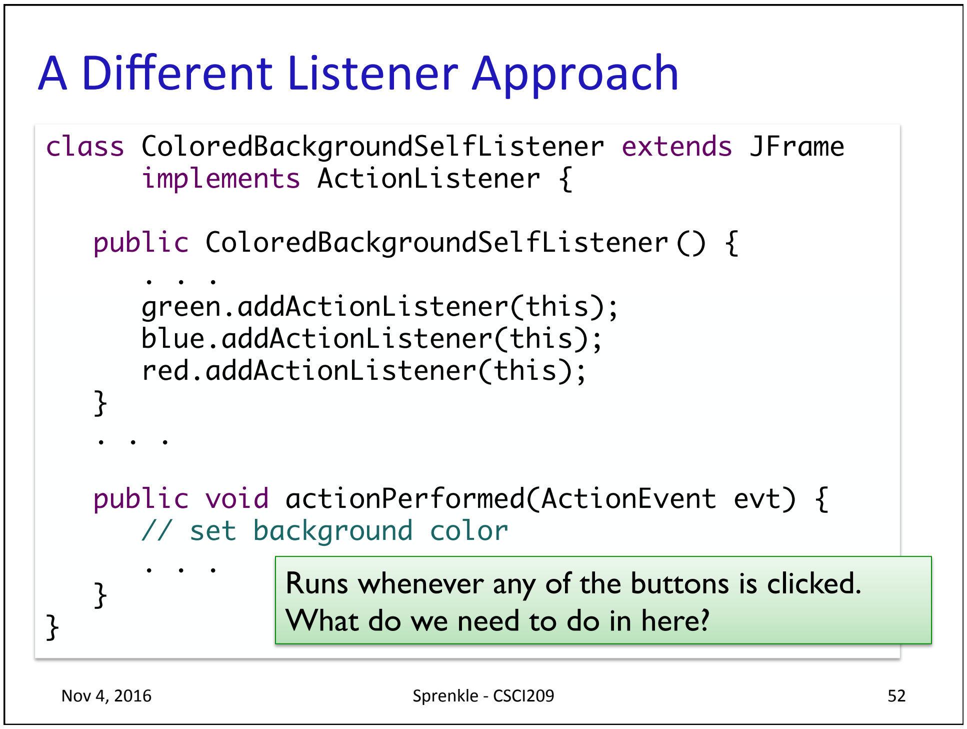
Event Listeners as Inner Classes

- A common and beneficial practice
- Event listener objects typically need to access/ modify other objects when their corresponding event occurs
- It is often possible to place the listener class inside the class whose state the listener should modify
- It's good OOP design
  - Doesn't violate encapsulation rules
  - Makes code easier



A Different Listener Approach

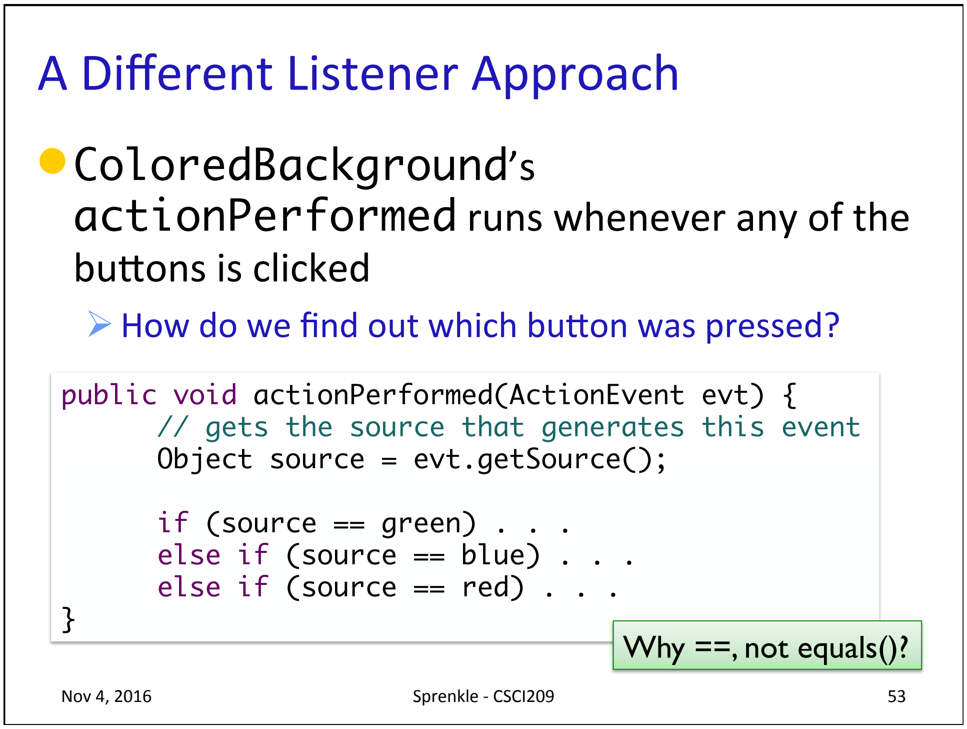
- Any object of a class that implements `ActionListener` can listen for action events from a source
  - Could make `ColoredBackground` listen for its own buttons' events
  - Implement interface and do correct registering with the buttons



A Different Listener Approach

```
class ColoredBackgroundSelfListener extends JFrame
      implements ActionListener {

   public ColoredBackgroundSelfListener() {
      . . .
      green.addActionListener(this);
      blue.addActionListener(this);
      red.addActionListener(this);
   }
   . . .

   public void actionPerformed(ActionEvent evt) {
      // set background color
      . . .
   }
}
```

Runs whenever any of the buttons is clicked.
What do we need to do in here?



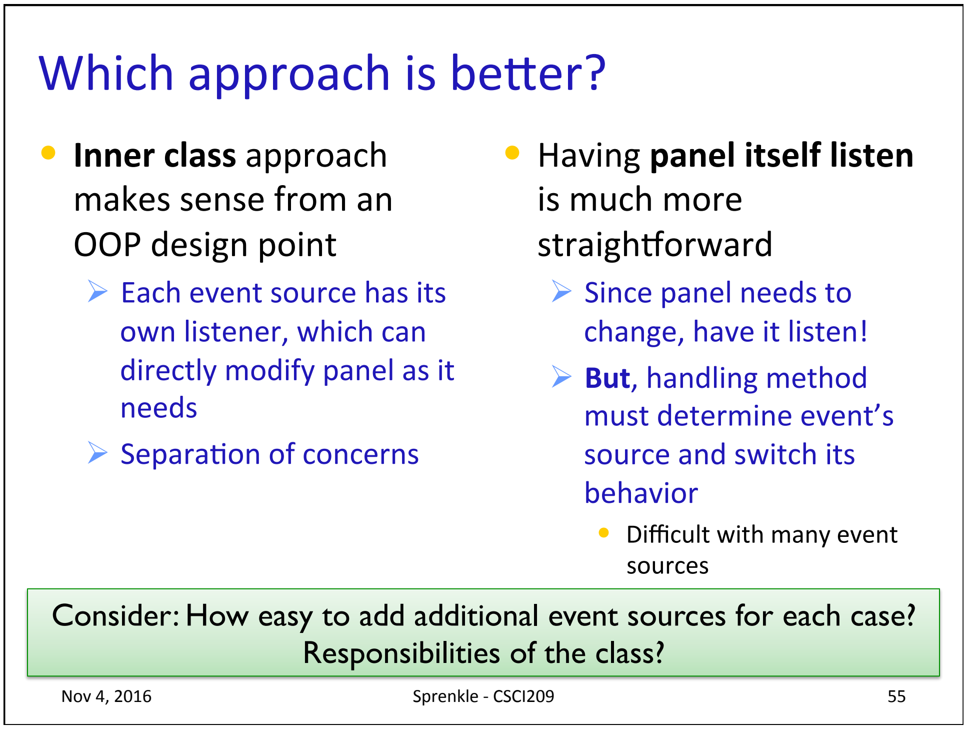
A Different Listener Approach

- ColoredBackground's actionPerformed runs whenever any of the buttons is clicked
  - How do we find out which button was pressed?

```
public void actionPerformed(ActionEvent evt) {
      // gets the source that generates this event
      Object source = evt.getSource();

      if (source == green) . . .
      else if (source == blue) . . .
      else if (source == red) . . .
}
```

Why ==, not equals()?

Which approach is better?

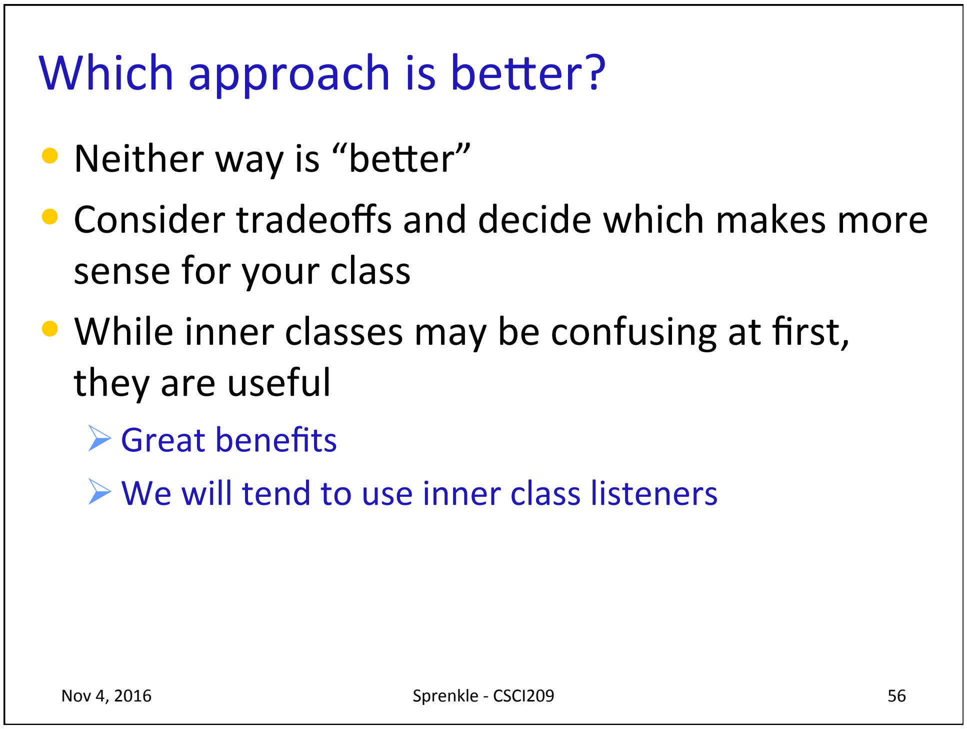


Which approach is better?

- **Inner class** approach makes sense from an OOP design point
  - Each event source has its own listener, which can directly modify panel as it needs
  - Separation of concerns

- Having **panel itself listen** is much more straightforward
  - Since panel needs to change, have it listen!
  - **But**, handling method must determine event's source and switch its behavior
    - Difficult with many event sources

Consider: How easy to add additional event sources for each case? Responsibilities of the class?



Which approach is better?

- Neither way is "better"
- Consider tradeoffs and decide which makes more sense for your class
- While inner classes may be confusing at first, they are useful
  - Great benefits
  - We will tend to use inner class listeners

Simplification of our Event Handlers

- For each button, we do four things:
    1. Construct the button with a label string
    2. Add the button to the panel
    3. Construct an action listener with the appropriate color
    4. Register that listener with the button

What does that call for?

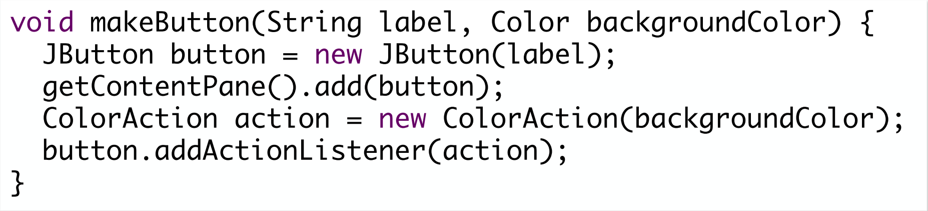
Simplification of our Event Handlers

```
void makeButton(String label, Color backgroundColor) {
  JButton button = new JButton(label);
  getContentPane().add(button);
  ColorAction action = new ColorAction(backgroundColor);
  button.addActionListener(action);
}
```

- Makes the ColoredBackground constructor much simpler…

```
public ColoredBackground()  {
      …
      makeButton("Green", Color.green);
      makeButton("Blue", Color.blue);
      makeButton("Red", Color.red);
}
```

Simplifying Further

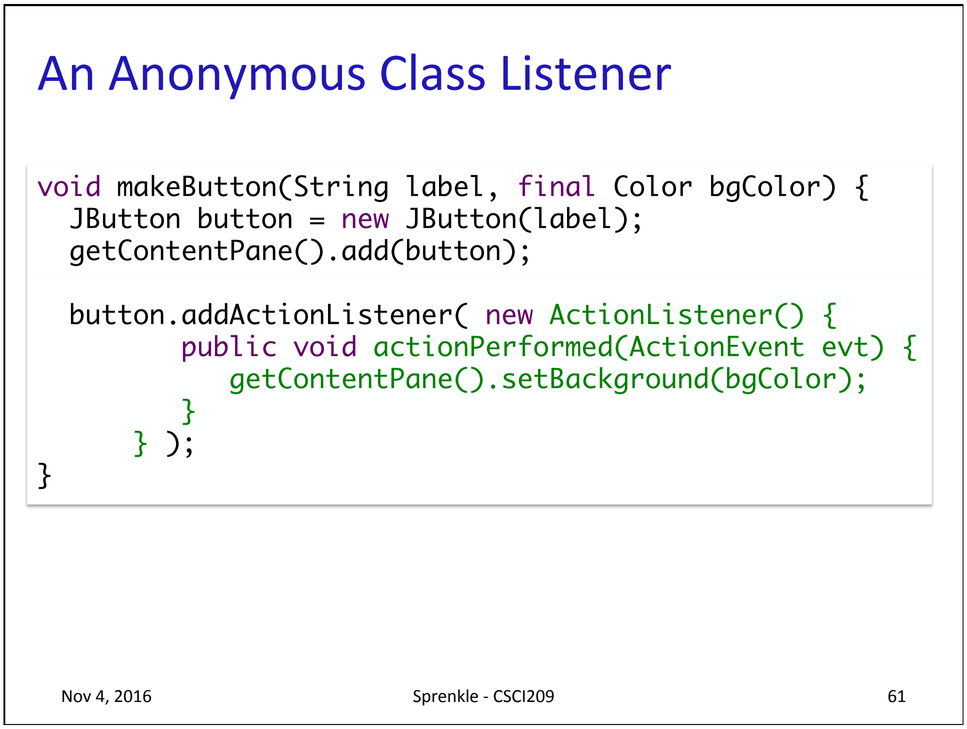


```
void makeButton(String label, Color backgroundColor) {
  JButton button = new JButton(label);
  getContentPane().add(button);
  ColorAction action = new ColorAction(backgroundColor);
  button.addActionListener(action);
}
```

- We *only* use the `ColorAction` class in `makeButton` method
  - We can further simplify the code…

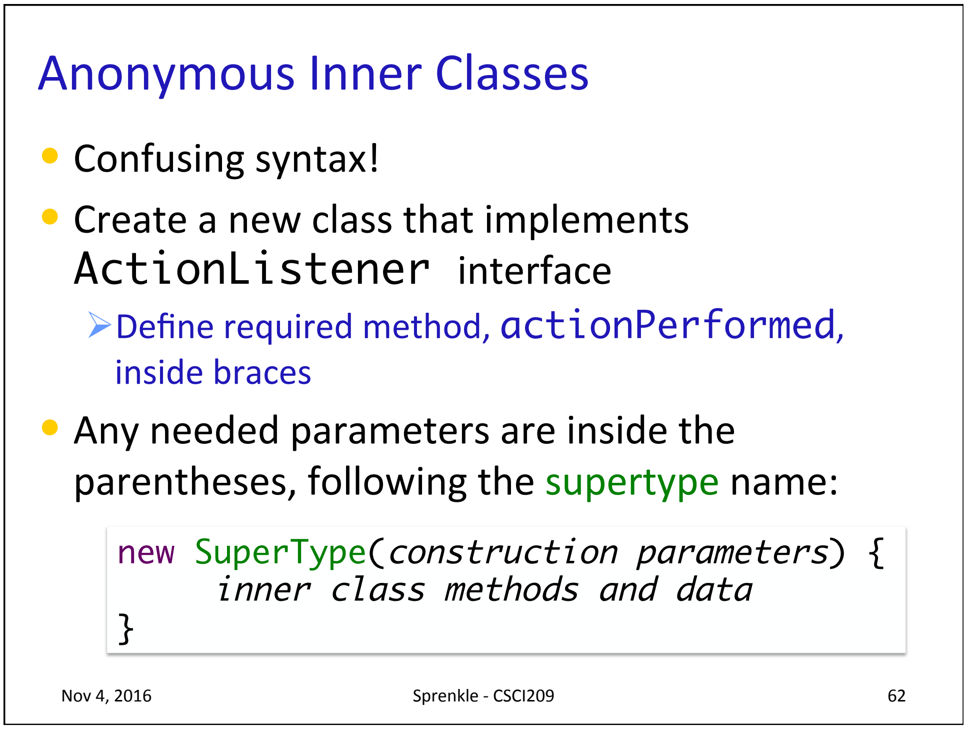
Simplifying Further

- Make the `ColorAction` class an ***anonymous inner class***
- Since only use class at one point, *define class on the fly*



An Anonymous Class Listener

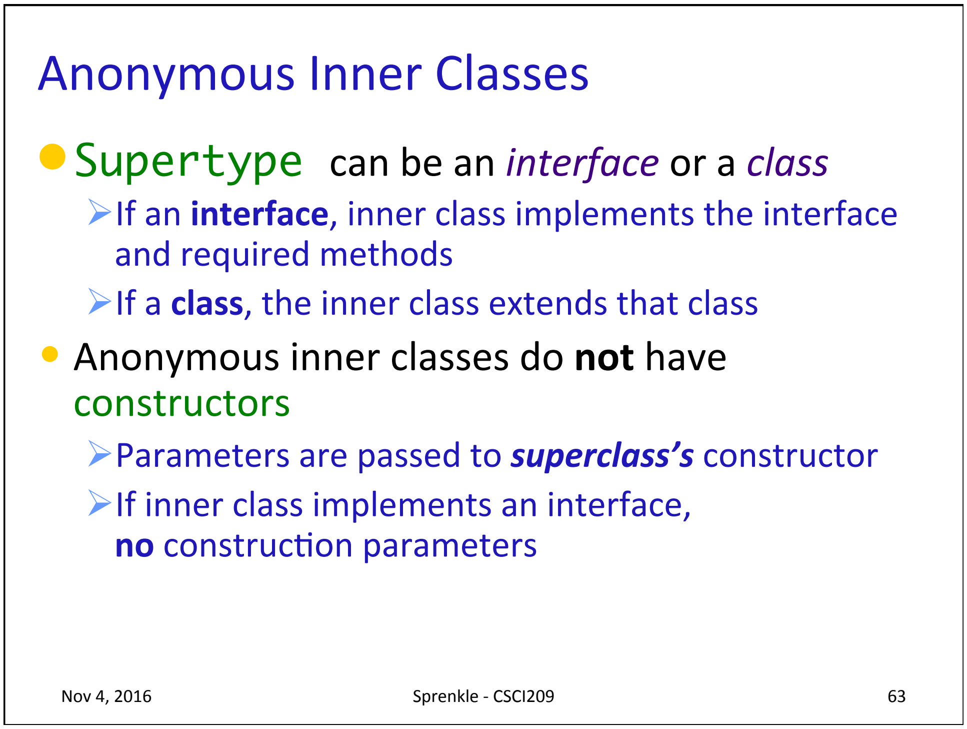
```
void makeButton(String label, final Color bgColor) {
  JButton button = new JButton(label);
  getContentPane().add(button);

  button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            getContentPane().setBackground(bgColor);
         }
      } );
}
```



Anonymous Inner Classes

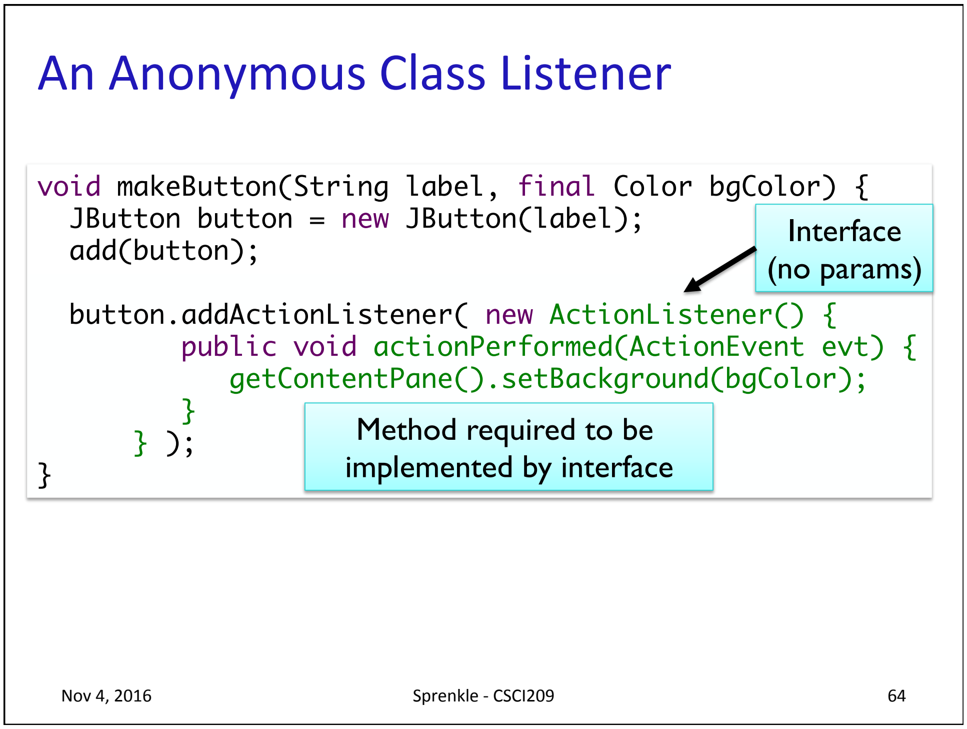
- Confusing syntax!
- Create a new class that implements `ActionListener` interface
  - Define required method, `actionPerformed`, inside braces
- Any needed parameters are inside the parentheses, following the supertype name:

```
new SuperType(construction parameters) {
      inner class methods and data
}
```



Anonymous Inner Classes

- Supertype can be an *interface* or a *class*
  - If an **interface**, inner class implements the interface and required methods
  - If a **class**, the inner class extends that class
- Anonymous inner classes do **not** have constructors
  - Parameters are passed to ***superclass's*** constructor
  - If inner class implements an interface, **no** construction parameters



An Anonymous Class Listener

```
void makeButton(String label, final Color bgColor) {
  JButton button = new JButton(label);
  add(button);

  button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            getContentPane().setBackground(bgColor);
         }
      } );
}
```

Interface (no params)

Method required to be implemented by interface

Anonymous Inner Classes

- Differentiate between
  - Construction of a new object of a class
  - Construction of an object of an anonymous inner class that extends that class…

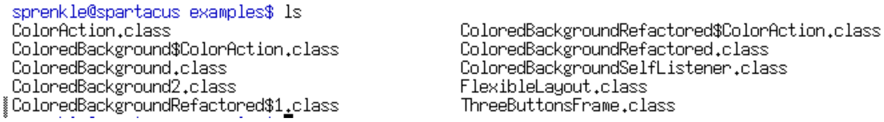
```
// this is a Person object
Person queen = new Person("Mary");

// this is an object of an anonymous
// inner class extending the Person class
Person count = new Person("Dracula") {. . .};
```

Finale!

- Show different versions of ColoredBackground GUI

Compiler's Names of Classes

- Contents of Eclipse project's `bin` directory examples:



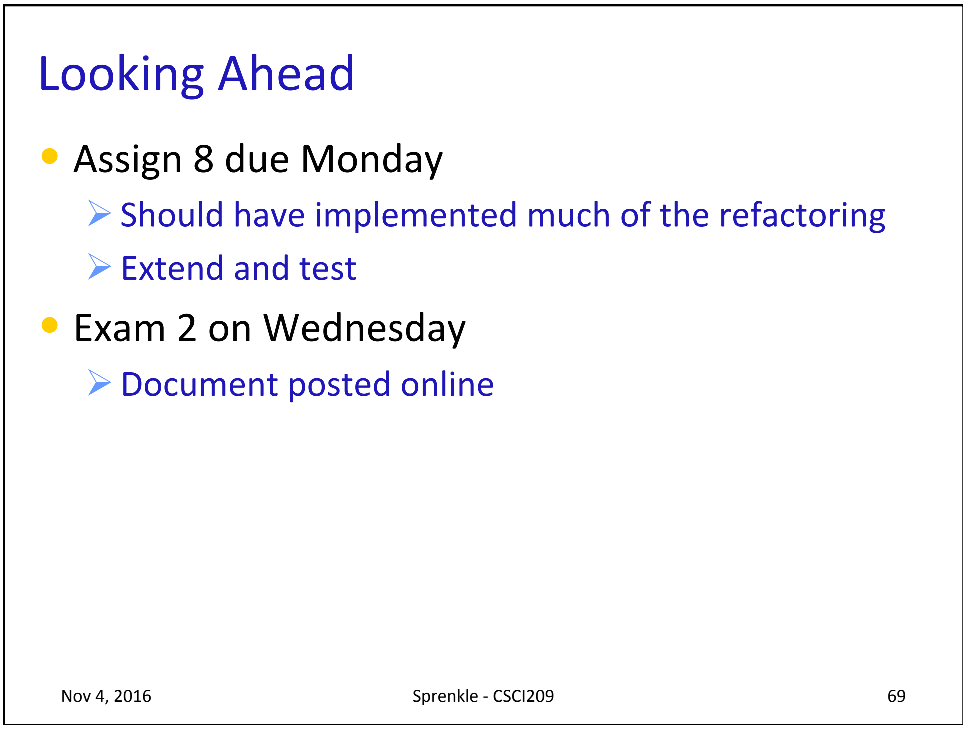
```
sprenkle@spartacus examples$ ls
ColorAction.class                              ColoredBackgroundRefactored$ColorAction.class
ColoredBackground$ColorAction.class            ColoredBackgroundRefactored.class
ColoredBackground.class                        ColoredBackgroundSelfListener.class
ColoredBackground2.class                       FlexibleLayout.class
ColoredBackgroundRefactored$1.class            ThreeButtonsFrame.class
```

Some unusual names. Why?

Read Others' GUI Code

- `CardLayoutDemo`
- `CardLayoutExample`

- Lots of example code and tutorials available online
  - Find something similar to what you want and adapt



Looking Ahead

- Assign 8 due Monday
  - Should have implemented much of the refactoring
  - Extend and test
- Exam 2 on Wednesday
  - Document posted online