Objectives
- GUIs in Java
- Layout Managers
- Event Handling

Assignment 11 Notes
- Focus on Extensibility
- But, handle other code smells as well
- Any questions

GUI Review
- What are the two main packages for GUI development in Java?
- Is GUI development looking a little difficult?
  - Why?

Review: JFrame
- Top-level window
  - Has title, border
- Contains ContentPane
  - A Container object that holds components you add, placing them in the frame
  - The part of the frame that holds UI components

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

More GUI Components
- Choice
  - Drop-down list
- FileDialog
  - Opening and saving files
- List
  - Scrolling
  - Allowing multiple selections
- ScrollPane
  - Scrollbars
- TextField
  - Single line of text
- TextArea
  - Multiple lines of text
Menus

- MenuBar
  - Thing across top of frame
  - Frame: setMenuBar(MenuBar mb);

- Menu
  - The dropdown part
  - A sequence of MenuItems
  - Menu is a subclass of MenuItems, so can have submenus

Practice: Combining Components

- Create a panel with three buttons on it

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

- 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

Border Layout Manager

- Default layout manager of the content pane for JFrame
- Lets you choose where you want to place each component
  - Center
  - North
  - South with respect to the container
  - East
  - West
Border Layout Regions

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

Border Layout Rules

- Grows all components to fill available space
- If container is resized, edge components are redrawn and center region size recomputed
- To add a component to a container using a border layout
  - Ex: JFrame's content pane

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

Adding Components Using a Border Layout

- If no region of the layout is specified
  - Assumes center region
- Since border layout grows the component to fit specified region
  - What happens if we add multiple components, e.g., three buttons, without specifying a region?

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

A Border Layout Limitation

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

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));
```

The Flow Layout Manager

- Default layout manager for a panel
  - (not JFrame)
  - What I changed our JFrame to use
- Lines components up horizontally until no more room in container
  - Then starts a new row of components
- If user resizes component, layout manager automatically reflows components
The Flow Layout Manager

- Can choose how to arrange components in a row
  - Default: center each row
  - Other options: left or right align
- Change alignment using `setLayout`

```java
setLayout(new FlowLayout( FlowLayout.LEFT ));
```
- Panel set to use a flow layout manager, with row components aligned to the left
- Another constructor has `hgap` and `vgap` for gaps to put around components

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 address Border Layout problem
  - Create a panel
  - Add some buttons to it
  - Add that panel to a region in content pane

Using Additional Panels

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

<table>
<thead>
<tr>
<th>Layout</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>BorderLayout</td>
<td>Content panes and enclosing panels</td>
</tr>
<tr>
<td>Flow Layouts</td>
<td>Panels containing buttons and other UI components</td>
</tr>
</tbody>
</table>

FlexibleLayout.java

Grid Layout Manager

- Divides container into columns and rows of equal size, which collectively occupy the entire container region
- Rows and columns are aligned like a spreadsheet
  - When the container is resized, the “cells” grow and/or shrink
  - Cells always maintain identical sizes
**Grid Layout Manager Construction**

- Number of rows and columns in layout
  ```
  panel.setLayout(new GridLayout(5, 4)); // 5 rows, 4 cols
  ```
- Can specify a horizontal and vertical separation between rows and columns:
  ```
  panel.setLayout(new GridLayout(5, 4, 20, 20)); // 5 rows, 4 cols, 20 pixels between rows & between cols
  ```
  - Can also specify with border and flow layout managers

**Adding Components to a Grid Layout**

- Components added sequentially
  - 1st `add()` adds the component to 1st row, 1st column
  - 2nd `add()` adds the component to 1st row, 2nd column.
  - And so forth until 1st row is filled
  - Then 2nd row begins with the 1st column
  - Continues until the entire container is filled

**Grid Layout Rules**

- Components are resized to take up entire cell
- Restrictive but can be useful for some applications
- Example: Create a row of buttons of identical size
  1. Make a panel that has a grid layout with one row
  2. Add a button to each cell
  3. Set horiz/vert separation, so buttons are not touching

**Layout Manager Heuristics**

- null
  - none, programmer sets x,y,w,h
- FlowLayout
  - Left to right, Top to bottom
- BorderLayout
  - n, e, s, c
- CardLayout
  - One at a time
- GridBagLayout
  - JButton

**Event-Driven Programming**

- User actions (e.g., mouse clicks, key presses), sensor outputs, or messages from other applications determine flow of program
- Application architecture:
  ```java
  while (true) {
    event = waitForEvent();
    handleEvent(event);
  }
  ```
Event Basics

- 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:
  ```java
  eventSourceObject.addEventListener(eventListenerObject);
  ```
- Example:
  ```java
  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, one registered to listen for a button’s events.
Event Handling Example

1. Make some buttons and add them to panel

```java
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

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

Registering Our Listener Class

- Create ActionListener objects and register them with the buttons...

```java
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);
```

Registering Our Listener Class

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

```
Any implementation issues?
```

The Listener Class & the Frame

- ColorAction objects don't have access to frame
  - How can they change the background color?
- Possible solutions?
The Listener Class & the Frame

- ColorAction objects don't have access to frame
  - 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 of class

Close Up: actionPerfomed()

- public void actionPerfomed(ActionEvent evt) {
  - setBackground(backgroundColor);
  - repaint();
}

- ColorAction does not have setBackground() or repaint() method
- Since ColorAction is an inner class of ColoredBackground, ColorAction can directly access ColoredBackground's instance fields and methods
  1. Inner class calls outer class's method
     - Parameter: inner's private data (backgroundColor)
  2. Inner calls outer class's repaint() method
     - Redraw the frame

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 also good OOP design
  - Does not 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 ColoredBackground2 extends JFrame implements ActionListener {
  - public ColoredBackground2() {
    - ...
    - green.addActionListener(this);
    - blue.addActionListener(this);
    - red.addActionListener(this);
  - ...}
  - public void actionPerformed(ActionEvent e) {
    - // 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?

```java
public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    if (source == green) ... else if (source == blue) ... else if (source == red) ... }
```

Why `==`, not `equals()`?

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

  Consider: How easy to add additional event sources for each case?

- 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

Which approach is better?

- Neither way is "better"
  - If container has multiple UI components that generate events, the container listening for and handling them all gets really confusing and challenging
  - Inner classes make sense
    - Somewhat confusing at first
    - 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?

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

- Makes the ColoredBackground constructor much simpler...

```java
public ColoredBackground() {
    makeButton("Yellow", Color.yellow);
    makeButton("Blue", Color.blue);
    makeButton("Red", Color.red);
}
```
Simplifying Further

- We only use the ColorAction class in makeButton method
- How can we further simplify the code?

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

• 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);
    add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            setBackground(bgColor);
            repaint();
        }
    });
}

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
Anonymous Inner Classes

- Carefully differentiate between
  - Construction of a new object of a class
  - Construction of an object of an anonymous inner class that extends that class...

```java
// 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") { ...};
```

Midterm Prep

- Java
  - Collections Framework
  - Comparison with Python
  - Jar files
- Software Development
  - Models
  - Testing
  - Design Principles
  - Code smells
  - Refactoring
- GUI programming
  - Event handling, inner classes