Objectives

- GUIs in Java
- Layout Managers
- Event Handling

Assignment 11 Questions

GUI Review

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

Review: Swing & AWT

- Swing does not completely replace AWT
- Using the Swing graphics programming model
  - Improves performance
  - Allows more efficient development of GUIs
- We will write graphics code that uses Swing
- We may return to AWT later
  - what AWT offers that Swing does not

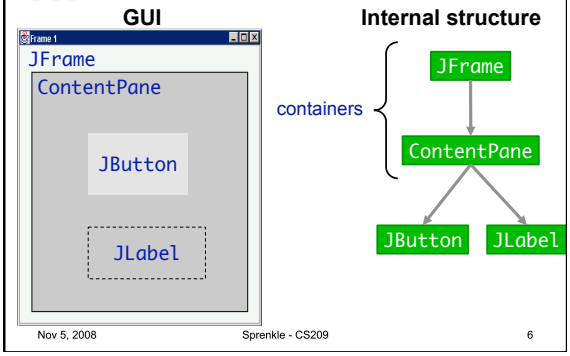
Review: Example Frame

```
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);
      // Shows the window
      setVisible(true);
      …
    }
}
```


Review: Anatomy of an Application GUI



Review: Using 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

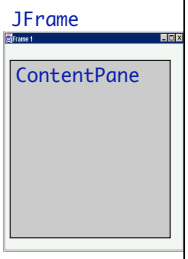
order important

Using a GUI Component

1. Create it
   - `JButton b = new JButton("press me");`
2. Configure it
   - `b.setText("press me");`
3. Add it
   - `panel.add(b);`
4. Listen to it
   - Events: Listeners

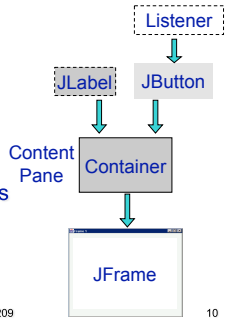
JFrame

- Contains `ContentPane`
  - A `Container` object within the `JFrame` holds components you add, placing them in the frame
  - The part of the frame that holds UI components



Building a GUI

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



Example Code

```
// create the components
JFrame f = new JFrame("title");
Container pane = f.getContentPane();
JButton b = new JButton("press me");

// add button to panel
pane.add(b);

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

JPanel

- Implements a panel
  - A panel has a surface on which you can draw
  - A panel is a `Container`
    - Can add components to a panel
  - Useful in designing layouts

To Draw on a Panel

- Define a new class that extends `JPanel`
- Override `paintComponent(Graphics g)` in derived class
  - `Graphics` object: collection of settings for drawing images and text, e.g., colors and fonts
  - All drawing in Java goes through a `Graphics` object

Drawing on a Panel

```
class MyPanel extends JPanel {

   public void paintComponent(Graphics g) {

     // code for drawing goes here

   }
}
```


`paintComponent`

- System calls `paintComponent()` *automatically* whenever container needs to be redrawn
  - Do *not* call this method yourself
  - It will be called when it needs to be
- If need to force repainting the screen, call `repaint()`
  - Calls `paintComponent()` for all needed components with appropriate `Graphics` objects

Drawing on a Panel: `Graphics`

- Measurements on a `Graphics` object is in pixels, as an offset from the top-left corner
  - (0,0) coordinates represent the top-left corner of the container on which you are drawing

Rendering Text

- Displaying text is a special type of drawing, called rendering text
- To render text on a panel, call `drawString()`

```
class HelloWorldPanel extends JPanel {
      public static final int MESSAGE_X = 75;
      public static final int MESSAGE_Y = 100;

      public void paintComponent(Graphics g) {
           super.paintComponent(g);

           g.drawString("Hello World.",
                 MESSAGE_X, MESSAGE_Y);
      }
}
```


Drawing on a Panel

- Notice we call superclass's (`JPanel`) `paintComponent()` method
- `JPanel` has its own idea on how to draw/ paint the panel
  - Fills in the background color
- To make sure background color gets filled, call superclass's `paintComponent()`
  - Every `JPanel` should color its background

Changing the Text Font

- Previous code drew text using default system font
- Can change the font
- Need to determine which fonts are installed on machine running the program

Determining Available Fonts

- `GraphicsEnvironment`
  - Represents the system's graphical environment
  - Call `getAvailableFontFamilyNames()`
    - Returns an array of Strings
    - Each String contains the name of a font installed on the system

Determining the Available Fonts

- To list all fonts installed on a particular system:

```
import java.awt.*;

public class ListFonts {

   public static void main(String[] args) {
     String[] fontNames = GraphicsEnvironment
       .getLocalGraphicsEnvironment()
       .getAvailableFontFamilyNames();
     for (int i=0; i < fontNames.length; i++)
       System.out.println(fontNames[i]);
   }
}
```


Determining the Available Fonts

- Your program can look through fonts to see if font(s) it wants is available on system
- Five fonts are always available, mapped to some font on machine
  - SansSerif
  - Serif
  - Monospaced
  - Dialog
  - DialogInput

Creating a `Font` Object

- `Font` object represents font on the system
- `Font` constructor takes 3 arguments:
  - a String with the font name
  - a constant (defined in the `Font` class) that describes the font style (plain, **bold**, *italic*, or ***bold italic***)
  - an integer for the point size

Creating a Font Object

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
Font helvi12    = new Font("Helvetica", Font.ITALIC, 12);
```

- You can change the font that the `Graphics` object uses by calling `setFont()`
- For example…

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g.setFont(sansbold14);
g.drawString("Hello there in SansSerif.", 75, 100);
```

More GUI components

- Label
  - Basically, just a string
- Buttons
  - Like a label but generates events
- Checkbox
  - Buttons with state about if checked
- CheckboxGroup
  - Radio buttons - only one can be selected at a time

More GUI Components

- Choice
  - Drop-down list
- FileDialog
  - Opening and saving files
- List
  - Scrollable
  - Allows multiple selections
- ScrollPane
  - scrollbars

More GUI Components

- 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?

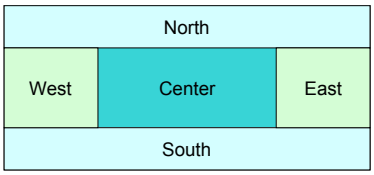
Layout Managers

- Java uses a concept of 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
  - *Flow layout* gives each component its preferred size
- 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

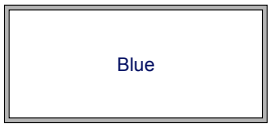
```
Container contentPane = getContentPane();
contentPane.add(yellowButton, BorderLayout.SOUTH);
```


Adding Components Using a Border Layout

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

- 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?

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 add 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

- You can choose how to arrange components in each row
  - Default: center each row
  - Other options: left or right align
- Change alignment using `setLayout()`

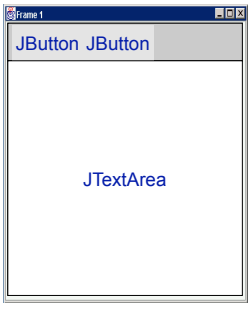
```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

  - Causes panel 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

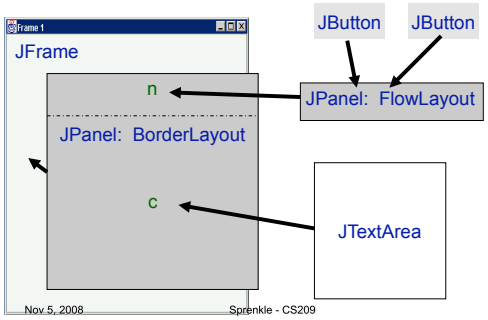
Using Panels w/ Border Layout

- 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 the south region of content pane

Combinations



Combinations



Using Additional Panels

- Get fairly accurate and precise placement of components
- Use nested panels with
  - Border layouts - for content panes and enclosing panels
  - Flow layouts - for panels containing the buttons and other UI components

FlexibleLayout.java

Grid Layout Manager

- Divides the 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 to be used in the layout

```
panel.setLayout(new GridLayout(5, 4)); // 5 rows, 4 cols
```

- As with border and flow layout managers, you 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
```

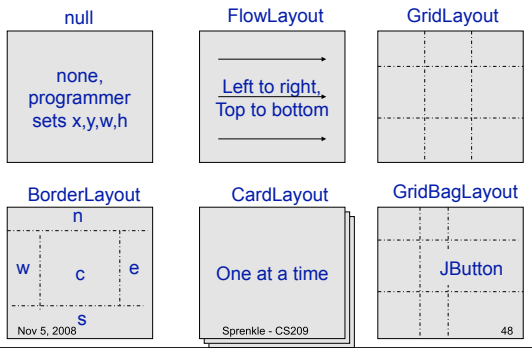

Adding Components to a Grid Layout

- Components are added to a grid layout sequentially
- First add() adds the component to the 1st row and 1st column
- Second add() adds the component to the 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

- When a component is added to a cell, it is resized to take up entire cell
- Quite restrictive but can be useful for some applications
- Example: Create a row of buttons of identical size
  - Make a panel that has a grid layout with one row
  - Add a button to each cell
  - Set horiz/vert separation, so buttons are not touching

Layout Manager Heuristics



HANDLING USER INTERACTIONS

Event-Driven Programming

- Flow of program is determined by user actions (e.g., mouse clicks, key presses), sensor outputs, or messages from other applications
- Application architecture:

```
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
  - Means: registered listener requests event source sends its event to listener when event occurs
- All events are objects of event classes, which derive from `java.util.EventObject`

Java / AWT Event Handling

- Listener object: implements a listener interface
- Event source: can register listener objects and send them event objects
  - Event source sends out event objects to *all* registered listeners when that event occurs
- Listener objects use the event object to determine their reaction to the event

Java / AWT Event Handling

- Register a listener with an event source:

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

- Example:

```
ActionListener listener1 = . . .;
JButton button1 = new JButton("Click Me!");
button1.addActionListener(listener1);
```

  - Whenever an "action event" occurs on button1, listener1 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 two things:
  - A panel with three buttons on it
  - Three listener objects, each registered to listen for events on one of the buttons

Event Handling Example

- Make some buttons and add them to panel

```
public class ColoredBackground extends JFrame {
   public ColoredBackground() {
      …
      JButton red = new JButton("Red");
      red.setForeground(Color.red);
      JButton yellow = new JButton("Yellow");
      yellow.setBackground(Color.yellow);
      JButton blue = new JButton("Blue");
      blue.setForeground(Color.blue);
      cp.add(red);
      cp.add(yellow);
      cp.add(blue);
…
```


Listener Objects

- Now that we have buttons (event sources), we need listeners
  - 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 {
  public ColorAction(Color c)
  { backgroundColor = c; }

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

  private Color backgroundColor;
}
```

How can we do this?

Registering Our Listener Class

- Create ActionListener objects and register them with the buttons…

```
ColorAction yellowAction = new ColorAction(Color.yellow);
ColorAction blueAction   = new ColorAction(Color.blue);
ColorAction redAction    = new ColorAction(Color.red);

yellow.addActionListener(yellowAction);
blue.addActionListener(blueAction);
red.addActionListener(redAction);
```

These are `JButtons`

Registering Our Listener Class

- When a user clicks the button with the label "Yellow", the yellow JButton object generates an ActionEvent
  - Passes this event object to the yellowAction's actionPerformed() method
  - Method can then set frame's background color

Any problems?

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:
  - 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
  - Make `ColorAction` an ***inner*** class of ButtonPanel1

Listener as an Inner Class

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

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

Where are these coming from?

The actionPerformed() Method

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

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

Event Listeners as Inner Classes

- A common and beneficial practice
- Event listener objects typically need to do something to 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
  - Not violating 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() {
      . . .
      yellow.addActionListener(this);
      blue.addActionListener(this);
      red.addActionListener(this);
   }
   . . .

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


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 == yellow) . . .
      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 gets its own listener, which can directly modify panel as it needs to do
- 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
- Consider: How easy to add additional event sources for each case?

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:
  - Construct the button with a label string
  - Add the button to the panel
  - Construct an action listener with the appropriate color
  - Register that listener with the button
- What does this mean we should do?

Simplification of our Event Handlers

```
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…

```
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?

Simplifying Further

- Make the `ColorAction` class an ***anonymous inner class***
  - Only use this class at one point
  - Define it 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();
         }
      } );
}
```