CSC2620 Object Oriented Design — Spring 2024
Unit 3 - Graphical User Interfaces (GUIs)

3.1 GUI Programming

- There are two main methods of user interfacing using a GUI:
  1. Dialog boxes
  2. GUI components in a panel or or frame.
- We will be using the second method for the majority of our GUI programming.
- Additional note: VS Code does have extensions for WYSIWYG GUI editing. It will not produce great code but would allow you to click-and-drag design your UIs. You are welcome to use these, but I have never done so and will not be able to help you with them.

3.2 Displaying a Window

- We'll now talk about a framework to build GUI applications. This general framework will be applicable to *all GUI applications* you build in Java.
- Windows that you create will almost always be instances of the `JFrame` class.
  - `JFrames` provide basic behaviors of a window - title bar, and maximize, minimize, and close buttons.
  - Most applications will consist of two classes: a main or application class which contains `main` that creates and displays the GUI, and a **subclass** of the `JFrame` class used to build the user interface.
- Your `GUIDemo` class will now be the driver for our application examples.
- Create a new Java project and replace the `main` method with:

```java
import javax.swing.JFrame;

public class DemoMain {

  public static void main(String[] args) {
    GUIDemo guiDemo = new GUIDemo();
    // What to do when the window closes:
    guiDemo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Size of the window, in pixels
    guiDemo.setSize(300, 180);
    // Make the window "visible"
    guiDemo.setVisible(true);
  } // end of main
} // end of GUIDemo
```

- We will build up class GUIDemo now.

3.3 Building a JFrame

- We'll start with some simple text display. To display text, we use a `JLabel`.
- Create a new class called `GUIDemo`:

```java
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

// Inheriting from JFrame allows polymorphism
//   to be used to display the window
public class GUIDemo extends JFrame
{
  // Create a "label", just some text in the GUI
  private JLabel label1;

  // Constructor lays out the GUI entirely
  public GUIDemo()
  {
    super("JLabel Demo"); // The name of the window
    setLayout( new FlowLayout() ); // A type of layout

    // JLabel constructor with string argument
    label1 = new JLabel( "Here's a JLabel" );
    // Mouse-over text
    label1.setToolTipText( "This is label1" );
    add( label1 ); // Add the label to the JFrame
  } // end of GUIDemo
} // end of GUIDemo
```

3.3.1 Adding an Image

- We'll start with some simple text display. To display text, we use a `JLabel`.
- Create a new class called `GUIDemo`:

```java
public class GUIDemo extends JFrame {

    private JLabel label1;
    private JLabel label2;

    // Constructor
    public GUIDemo() {
        super("JLabel Demo"); // The name of the window
        setLayout(new FlowLayout()); // A type of layout

        // JLabel constructor with string argument
        label1 = new JLabel("Here's an image");
        // Mouse-over text
        label1.setToolTipText("This is label1");
        add(label1); // Add the label to the JFrame

        // Load the image
        Image img = null;
        img = ImageIO.read(new File("image.jpg"));
        label2 = new JLabel(new ImageIcon(img));
        add(label2);
    } // end of GUIDemo
} // end of GUIDemo
```

3.4 Layouts

- There are several layout options for your GUI.

  1. FlowLayout - The standard layout, components are added left to right, top to bottom.
  2. GridLayout - Determine how many rows and columns, each component takes up a single grid space
  3. CardLayout - Have multiple components share the same space, rotate between them
  4. BorderLayout - Components live in different designated areas of the panel or frame
  5. GridBagLayout - Used to align components vertically, horizontally or along their baseline.

- The simplest for many applications is the grid layout.

- To use the grid layout, create a new instance of the `GridLayout` class, a private field within the GUI class.

- The `GridLayout` constructor takes, as input, either two arguments (rows and columns), or four arguments (rows and columns, and the gaps between them, in pixels).

- The current layout is a state machine, meaning that when the "current" layout is set as a particular grid layout, any components "added" (using the `add` method) adds the buttons, left to right, top to bottom, in the grid spaces.

```
import java.awt.*;
import javax.swing.*;

public class GUIDemo extends JFrame {
  // 6 Labels
  private JLabel[] labels = { new JLabel("This is Label1"),
      new JLabel("This is Label2"), new JLabel("This is Label3"),
      new JLabel("This is Label4"), new JLabel("This is Label5"),
      new JLabel("This is Label6") };

  // An instance of the grid layout: 3 rows, 2 columns, 5 pixel
   gaps
  private GridLayout gridLayout1 = new GridLayout(3, 2, 5, 5);

  // Constructor
  public GUIDemo() {
    super("GridLayout Demo"); // The name of the window
    // Set the current layout of the JFrame
    setLayout(gridLayout1);
    // Now add the labels
    for( int i = 0 ; i < labels.length ; i++ )
      add( labels[ i ] );

  } // end of GUIDemo
} // end of GUIDemo
```

3.5 Components of the JFrame

- Each `JFrame` actually consists of three layers
  1. Background
  2. Content Pane
  3. Glass Pane
- The content pane appears in front of the background and is where the GUI components in the `JFrame` are displayed.
- The glass pane is displays tool tips and other items that should appear in front of the GUI components on the screen.
- The content pane completely hides the background of the `JFrame`.
- To change the background color behind the GUI components, you must change the content pane's background color.
- Method `getContentPane` returns a reference to the `JFrame`'s content pane (an object of class `Container`).

3.6 JPanels and Nested Layouts

- Complex GUIs require that each component be placed in an exact location. This often consist of multiple panels, with each panel's components arranged in a specific layout.
- Every `JPanel` may have components, including other panels, attached to it with the `Container` method add that we are used to using with the `JFrame` class.
- `JPanel` can be used to create a more complex layout in which several components are in a specific area of another container.

```java
import java.awt.*;
import javax.swing.*;

public class GUIDemo extends JFrame {
    // 5 Labels
    private JLabel[] labels = {new JLabel("This is Label1"),
       new JLabel("This is Label2"), new JLabel("This is Label3"),
       new JLabel("This is Label4"), new JLabel("This is Label5")
   };

    // Constructor
    public GUIDemo() {
        super("GridLayout Demo"); // The name of the window
        // Set the current layout of the JFrame
        // Create a JPanel with a 2x2 layout
        JPanel tinyPanel = new JPanel(new GridLayout(2, 2, 5, 5));
        setLayout(new GridLayout(3, 2, 5, 5) ); // Sets the layout
        // Add four labels to the panel;
        tinyPanel.add(new JLabel("inside the panel! 0,0"));
        tinyPanel.add(new JLabel("inside the panel! 0,1"));
        tinyPanel.add(new JLabel("inside the panel! 1,0"));
        tinyPanel.add(new JLabel("inside the panel! 1,1"));
        // Set the background color of the panel to be red
        tinyPanel.setBackground(Color.red);
        // Add the panel to the JFrame
        this.add(tinyPanel);
        // Now add the labels to the rest of the JFrame sections
        for (int i = 0; i < labels.length; i++) {
            add(labels[i]);
        }
    } // end of GUIDemo
} // end of GUIDemo
```

- In this example, the frame and the panel within the frame have different layouts. This nesting idea allows for powerfully designed GUIs.
- You can nest panels as deeply as you want (you can add panels to panels, each with their own layouts, etc.).

3.7 Practice

- Create a new GUI project.
- Display in a 2x3 grid (2 rows, 3 columns) 3 images (either find some online or use ones on your computer) with captions (use labels).
- Now, instead of your `JFrame` being a 2x3 grid, make it a 2x1 grid and *nest* your previous 2x3 `GridLayout` within this new 2x1 `GridLayout`. Add a label to the outer layout with a title for the series of pictures. Make the text large and bold (how can you do this?) to show that it's a title for the GUI.

3.8 Event-Driven Interfaces

- GUIs are **event driven**, meaning that they require an **event** to perform an action (such as clicking a button). The **event handler** deals with the action to be performed after an event has taken place.

- Event Handlers (also called Listeners) are needed to catch the event and process it. This requires the use of a **nested class**.

  - We can define a class within another class. These classes are declared **private**, meaning that they can only be used within the class in which they are defined.
  - The definition looks largely the same as it would if the class were public.

```
public class testClass {

  // The nested class:
  private class nestedClass {
    private int field1;
    public int field2;
  } // end of class nestedClass

  public static void main( String[] args )
  {
    nestedClass instanceOfNestedClass = new nestedClass();

  } // end of main
} / end of class testClass
```

- So we can add a text input field to our GUI:

```java
import java.awt.GridLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import javax.swing.*;

public class GUIDemo extends JFrame {

    private JLabel label1;
    private JPanel inputPanel;
    // Text fields allow small amounts of text input
    private JTextField textField1;
    private JTextField textField2;

    // Constructor
    public GUIDemo() {
        super("ActionListener Demo"); // The name of the window
        setLayout(new FlowLayout()); // A type of layout

        inputPanel = new JPanel(new GridLayout(2, 1, 5, 5));
        // Add a text field...the two parameters to this method
   are
        //  the ’default text’, and the width of the text input
        textField1 = new JTextField("Enter name here", 20);
        textField2 = new JTextField("Enter name here", 20);
        inputPanel.add(textField1);
        inputPanel.add(textField2);
        // Add the input panel to the GUI
        add(inputPanel);

        // We have to add event handlers
        TextFieldHandler handler = new TextFieldHandler();
        textField1.addActionListener(handler);
        textField1.addFocusListener(handler);
        textField2.addActionListener(handler);
        textField2.addFocusListener(handler);
    } // end of GUIDemo
    // The private class to handle events
    private class TextFieldHandler implements ActionListener,
   FocusListener {
        // When action is generated ("enter" is pressed)

        @Override
        public void actionPerformed(ActionEvent event) {
            String string =
                String.format("%s", event.getActionCommand());

            // Show a message to the users with the text input
            JOptionPane.showMessageDialog(null, string);
        } // end of actionPerformed


        // If you click inside the text field and it gains focus,
        //   remove the text
        @Override
        public void focusGained(FocusEvent event) {
            ((JTextField) event.getSource()).setText("");
        }

        // But if you leave the text field and it loses focus
        //   without text, reset the text
        @Override
        public void focusLost(FocusEvent event) {
           if (((JTextField) event.getSource()).getText().equals("
   "))
           {
             ((JTextField) event.getSource()).setText("Enter name"
   );
           }
        }
    } // end of class TextFieldHandler
} // end of GUIDemo class
```

3.9 Buttons

- We can add buttons to the GUI as well. Each button is an object of the `JButton` class. They also require event handlers.

```
public class GUIDemo extends JFrame {
  private JLabel label1;
  private JTextField textField1;
  private JButton button1;

  // Constructor
  public GUIDemo() {
    super("JButton Demo"); // The name of the window
    setLayout(new FlowLayout()); // A type of layout

    // JLabel constructor with string argument
    label1 = new JLabel("Here's a JLabel"); // The label text
    label1.setToolTipText("This is label1"); // Mouse-over text
    this.add(label1); // Add the label to the JFrame

    // Add a button
    button1 = new JButton( "Click here!");
    this.add(button1);
    // Add the event handler to the button
    ButtonHandler buttonHandler = new ButtonHandler();
    button1.addActionListener( buttonHandler );
  } // end of GUIDemo

  // A class to handle the button event
  private class ButtonHandler implements ActionListener {

    public void actionPerformed(ActionEvent event) {
      String string = "";

      if(event.getSource() == button1) {
        string = String.format("buttonField1 was pressed");
      } // end of if

      // Show a message to the users with the button
      JOptionPane.showMessageDialog(null, string);
    } // end of actionPerformed
  }
} // end of GUIDemo
```

- If we want two objects (components) to interact with one another (such as a button and a text field), their action listeners should be the same class.

- So what would it look like if we were to make a button that displayed the text in the current text field?

```
public class GUIDemo extends JFrame {
    private JTextField textField1;
    private JButton button1;

    // Constructor
    public GUIDemo() {
        super("JButton Demo"); // The name of the window
        setLayout(new FlowLayout()); // A type of layout

        // Text field
        textField1 = new JTextField("Enter name here", 20);
        this.add(textField1);
        // We have to add event handlers
        TextFieldHandler handler = new TextFieldHandler();
        textField1.addActionListener(handler);
        textField1.addFocusListener(handler);

        // Add a button
        button1 = new JButton("Click here!");
        this.add(button1);
        // Add the event handler to the button
        button1.addActionListener(handler);
    } // end of GUIDemo

    // The private class to handle events
    private class TextFieldHandler implements
               ActionListener, FocusListener {
        @Override
        public void actionPerformed(ActionEvent event) {
            // If the button is pressed, get text in textField1
            String string = String.format("textField1: %s",
                   textField1.getText());
            // Show a message to the users with the text input
            JOptionPane.showMessageDialog(GUIDemo.this, string);
        } // end of actionPerformed

        // If the text field gains focus, remove the text
        @Override
        public void focusGained(FocusEvent event) {
            ((JTextField) event.getSource()).setText("");
        }

        // If the textfield loses focus without text, reset
        @Override
        public void focusLost(FocusEvent event) {
          if(((JTextField)event.getSource()).getText().equals(""))
            ((JTextField) event.getSource()).
              setText("Enter name here");
        }
    } // end of class TextFieldHandler
} // end of class GUIDemo
```

- Notice that the showMessageDialog call does not take in `null`, it takes in the parent class's `this` field. Doing so makes the dialog box appear centered on the `JFrame`.

- The private nested class has access to the fields within the parent class. Use this fact to interface the GUI class with other classes.

3.10 JTextAreas

- A `JTextArea` provides an area for manipulating multiple lines of text.
- `JTextArea` is a subclass of `JTextComponent`, which declares common methods for `JTextFields`, `JTextAreas` and several other text-based GUI components.

```java
public class GUIDemo extends JFrame {
    private JTextArea textArea1;
    private JTextArea textArea2;
    private JButton copyJButton;

    // Constructor
    public GUIDemo() {
        super("JTextArea Demo"); // The name of the window
        // Create a "box" with FlowLayout
        Box box = Box.createHorizontalBox();
        String demo = "This is a demo string that \n" +
                "  will be copied from one text area to \n" +
                "  the other. \n";

        // Create text area w/ # of visible of rows and columns
        textArea1 = new JTextArea( demo, 10, 15 );
        // Add the text area in a scroll pane
        box.add( new JScrollPane( textArea1 ) );

        // Create the copy button
        copyJButton = new JButton( "Copy >>>" );
        box.add( copyJButton );
        // Inner method to implement funtionality for copy button
        //   This is an "anonymous class" - it doesn't have a name
        copyJButton.addActionListener( new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                textArea2.setText( textArea1.getSelectedText() );
            }
        });

        // Set up textArea2
        textArea2 = new JTextArea( "", 10, 15 );
        // Can't edit the second text area
        textArea2.setEditable( false );
        box.add( new JScrollPane( textArea2 ) );

        // Add the box to the JFrame
        this.add( box );
    } // end of GUIDemo
}
```

- `Box` is a subclass of `Container` that uses a `BoxLayout` to arrange the GUI components horizontally or vertically.
- `Box`'s `static` method `createHorizontalBox` creates a `Box` that arranges components left to right in the order that they are attached.

- `JTextArea`'s method `getSelectedText` (inherited from `JTextComponent`) returns the selected text from a `JTextArea`.
- `JTextArea`'s method `setText` changes the text in a `JTextArea`.
- When text reaches the right edge of a `JTextArea` the text can wrap to the next line. This is referred to as **line wrapping**. By default, `JTextArea` does not wrap lines.
- You can set the horizontal and vertical scrollbar policies of a `JScrollPane` when it's constructed, as we saw in the examples from last class.
- You can also use `JScrollPane` methods `setHorizontalScrollBarPolicy` and `setVerticalScrollBarPolicy` to change the scrollbar policies.
- Class `JScrollPane` declares the constants
  - `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS` and `JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS` indicate that a scrollbar should always appear.
  - `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED` and `JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED` indicate that a scrollbar should appear only if necessary (the defaults).
  - `JScrollPane.VERTICAL_SCROLLBAR_NEVER` and `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER` indicate that a scrollbar should never appear.
  - If policy is set to `HORIZONTAL_SCROLLBAR_NEVER`, a `JTextArea` attached to the `JScrollPane` will automatically wrap lines.

3.11 Practice

1. Start a new project and copy the code from the previous `JTextArea` example into the new project. Run it and make sure it works.
2. Add a button which, when pressed, switches which of the two `JTextArea`s from the previous example is editable.
3. Add a "save" button to your GUI.
   (a) When the user presses this button, it should prompt the user for a filename. Prompt the user with `JOptionPane.showInputDialog`, a method that takes, as input, a message to prompt the user with, and returns the string that the user responds with. This is one additional way to take input with specific timing (such as when saving a file).
   (b) Then, the program should save the text in *whichever text area is editable* to a file with the name the user provided.
   (c) Finally, this action should clear the text in the text area.
4. Add a button which, when pressed, *adds* a `JTextArea` to the GUI. Do you run into any difficulty with this behavior?

3.12 State Buttons

- We can also create a series of state buttons:
  1. Radio Buttons
  2. Check Boxes
  3. Toggle Buttons

3.12.1 JCheckBox

- A check box (instance of class `JCheckBox`) is simply a button with two states: on and off.
- It looks like a check box.
- When the state is changed, that is an event, and the event handler acts accordingly:

```
public class GUIDemo extends JFrame {
  private JTextField textField1;
  private JCheckBox boldJCheckBox;
  private JCheckBox italicJCheckBox;

  // Constructor
  public GUIDemo() {
    super("JCheckBox Demo"); // The name of the window
    setLayout(new FlowLayout()); // A type of layout

    // Add a text field...the two parameters to this method are
    // the 'default text', and the width of the text input
    textField1 = new JTextField("Watch the Font Change", 20);
    // Set the default font
    textField1.setFont(new Font("Serif", Font.PLAIN, 14));
    this.add(textField1);

    // Add the check boxes (with labels)
    boldJCheckBox = new JCheckBox("Bold");
    italicJCheckBox = new JCheckBox("Italic");
    this.add(boldJCheckBox);
    this.add(italicJCheckBox);

    // We have to add event handlers
    CheckBoxHandler handler = new CheckBoxHandler();
    boldJCheckBox.addItemListener(handler);
    italicJCheckBox.addItemListener(handler);
  } // end of GUIDemo
  // The private class to handle events (not an ActionListener)
  private class CheckBoxHandler implements ItemListener {
    // MUST implement the method itemStateChanged:
    public void itemStateChanged( ItemEvent event ) {
      // Create a new Font object
      Font font = null;

      // Set the font object accordingly
      if(boldJCheckBox.isSelected()&&italicJCheckBox.isSelected())
        font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14);
      else if( boldJCheckBox.isSelected() )
        font = new Font( "Serif", Font.BOLD, 14);
      else if( italicJCheckBox.isSelected() )
        font = new Font( "Serif", Font.ITALIC, 14);
      else
        font = new Font("Serif", Font.PLAIN, 14);

      // Set the font
      textField1.setFont( font );
    } // end of itemStateChanged

  } // end of class CheckBoxHandler
} // end of class GUIDemo
```

- Notice that the nested class now implements ItemListener, not ActionListener. The imports must reflect this difference.

- Also pay attention to the application of the Font class. Most objects with text have a `setFont` method. If you want to change the way the font looks, consider experimenting with this.

- The `Font` class has a series of `final` fields declared within to determine the type of font style (`PLAIN`, `ITALIC`, and `BOLD`, among others).

3.12.2 Radio Buttons

- Radio Buttons are buttons where the value is toggled between a series of possibilities.
- Working with radio buttons requires that you create a series of buttons, group them together (so only one is selectable at a time), and register events for each one:

```
public class GUIDemo extends JFrame {
  private JTextField textField1;
  private Font plain, bold, italic, boldItalic;
  private JRadioButton plainButton, boldButton, italicButton,
                       boldItalicButton;
  private ButtonGroup radioGroup;

  // Constructor
  public GUIDemo() {
    super("JRadioButton Demo"); // The name of the window
    setLayout(new FlowLayout()); // A type of layout

    // Add a text field...the two parameters to this method are
    // the 'default text', and the width of the text input
    textField1 = new JTextField("Watch the Font Change", 20);
    this.add(textField1);

    // Create the radio buttons, making one ‘true’ so selected
    plainButton = new JRadioButton( "Plain", true );
    boldButton = new JRadioButton( "Bold", false );
    italicButton = new JRadioButton( "Italic", false );
    boldItalicButton = new JRadioButton( "Bold Italic", false );
    // Add them to the GUI
    this.add( plainButton );
    this.add( boldButton );
    this.add( italicButton );
    this.add( boldItalicButton );

    // Create a relationship between the buttons
    radioGroup = new ButtonGroup();
    radioGroup.add( plainButton );
    radioGroup.add( boldButton );
    radioGroup.add( italicButton );
    radioGroup.add( boldItalicButton );

    // Create a series of font objects to use for the group
    plain = new Font( "Serif", Font.PLAIN, 14 );
    bold = new Font( "Serif", Font.BOLD, 14 );
    italic = new Font( "Serif", Font.ITALIC, 14 );
    boldItalic = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
    // Set the original font
    textField1.setFont( plain );

    // Finally, register the events
    plainButton.addItemListener(new RadioButtonHandler(plain));
    boldButton.addItemListener(new RadioButtonHandler(bold));
    italicButton.addItemListener(new RadioButtonHandler(italic));
    boldItalicButton.addItemListener(
        new RadioButtonHandler(boldItalic));
  } // end of GUIDemo
  // Private class to handle events (not ActionListener this time)
  private class RadioButtonHandler implements ItemListener {
    // Need a private field for the font associated with the
   listener
    private Font font;

    // Implement a constructor to set the font
    public RadioButtonHandler(Font f) {
      font = f;
    } // constructor for RadioButtonHandler

    // MUST implement the method itemStateChanged:
    public void itemStateChanged(ItemEvent event) {
      textField1.setFont(font);
    } // end of itemStateChanged

  } // end of class CheckBoxHandler
} // end of GUIDemo
```

- Check out toggle buttons.

3.13 A Summary of How Events Work

- GUI components are event driven. This is the way in which users interact with GUI components and tell the program what actions they want performed.
  - Clicking a button causes an Action Event.
  - Selecting a text field (giving it **focus**) and hitting the Enter key causes an Action Event.
  - Selected a check box or radio button causes an Item Event.
- In order to deal with events, two steps must be taken:
  1. You must create a class that representes the event handler and implements the appropriate interface, known as the **event-listener interface**.
  2. You must indicate that an object of the class from step 1 should be notified when the event occurs, known as **registering the event handler**.
- We've seen this throughout our examples thus far.
- In general, we use **nested classes** to implement the event handler interfaces.
  - Before this, all of the classes we had seen were **top-level classes**.
  - Sometimes nested classes are `static` and thus accessible without an instance of the parent class. If they are not `static`, as is the case with event handlers, they are called **inner classes**.
  - The inner class object must be created by the top-level class that it resides within.
- ActionListeners *must* implement the method `actionPerformed` which takes, as an input argument, an event of type `ActionEvent`. This event contains the information necessary to respond to the event, such as the component in the GUI that caused the event, called the **event source**.
- Every `JComponent` has an instance variable called `listenerList`.
  - It is an object of the `EventListener` class.
  - It represents a list of all of the action listeners that have been assigned to the component.
  - You can assign any number of event listeners to a single component.
- When an event occurs, the event is **dispatched** only to the event listeners of the appropriate type. Since the component knows the types of its listeners, this is possible.
- When the event occurs, the component receives a unique **event ID**.
  - The component uses the ID to determine which listener type to dispatch the event to.
  - For instance, when an `ActionEvent` is detected, *every* registered `ActionListener`'s `actionPerformed` method is called.
  - This is possible due to the fact that the components all keep a list of all event handlers. The components make the decisions.

3.14 Practice

1. Open your IDE and create a new Java application. Download the starter code for this practice from Classroom.

2. Create a new GUI class, as discussed in lecture. This class should have an instance of `Bank`, initialized in the constructor. You can assume there will not be more than 100 accounts.

3. Create a GUI that has a button that, when pressed, allows the user to add an account. The GUI should take input (the initial balance) via a pop up dialogue box. To do this, use `JOptionPane`'s `showInputDialog` method. The button's listener should call `Bank`'s add method.

4. Add another radio button group that has a radio button for each of the accounts, allowing the user to select which account she is depositing into or withdrawing from. Adjust the "add account" functionality to accommodate this.

5. Add a text field to your GUI labeled "Amount".

6. Add a radio button group, and two radio buttons, one marked "deposit" and the other marked "withdrawl". Add a button labeled "Go" that, when pressed, should perform the appropriate function (deposit or withdrawl), given the amount provided by the user in the "Amount" text field.

7. Add a pop-up dialog window that provides a message to the user saying the amount was deposited or withdrawn successfully.

8. Test deposit functionality. Then test withdrawl functionality, with both a valid withdraw amount and an invalid withdraw amount.

9. Create a TextArea and a button labeled "PrintAccounts". When the user clicks this button, all accounts should be printed to the TextArea. Remember that this functionality should exist in `Bank`'s `toString` method. The `String` returned should be displayed in the TextArea when the button is pressed.

3.15 Combo Boxes

- A **combo box** is a drop-down menu from which the user can make choices.

- The first item added to a `JComboBox` appears as the currently selected item when the `JComboBox` is displayed. Other items are selected by clicking the `JComboBox`, then selecting an item from the list that appears.

- `JComboBox` method `setMaximumRowCount` sets the maximum number of elements that are displayed when the user clicks the `JComboBox`. If there are additional items, the `JComboBox` provides a scrollbar that allows the user to scroll through all the elements in the list.

- For event handling, let's do something different for combo boxes, and use an anonymous inner class
  - An **anonymous inner class** is an inner class that is declared without a name and typically appears inside a method declaration.
  - As with other inner classes, an anonymous inner class can access its top-level class's members.
  - An anonymous inner class has limited access to the local variables of the method in which it's declared.
  - Since an anonymous inner class has no name, one object of the anonymous inner class must be created at the point where the class is declared.
  - The `JComboBox` method `getSelectedIndex` returns the index of the selected item (the first item has index 0, the second had index 1, etc.).
  - For each item selected from a `JComboBox`, another item is first deselected. Therefore, two `ItemEvent`s occur when an item is selected.
  - The `ItemEvent` method `getStateChange` returns the type of state change.
    `ItemEvent.SELECTED` indicates that an item was selected.

```java
public class GUIDemo extends JFrame {
  private JComboBox stringBox; // Contains strings to show
  private JLabel label; // Label to display selected string

  private static final String[] labels =
     { "Hello!", "Hey?", "What's up?", "Hi!" };

  public GUIDemo() {
    super("Combo Box Testing");
    setLayout(new FlowLayout());

    // Create a new combo box, and provide an array of labels
    stringBox = new JComboBox(labels);
    stringBox.setMaximumRowCount(3);

    stringBox.addItemListener(new ItemListener() // The anon class
        {
          // handle the JComboBox event
          public void itemStateChanged(ItemEvent event) {
            // determine if this was selected
            if (event.getStateChange() == ItemEvent.SELECTED)
              label.setText(labels[stringBox.getSelectedIndex()]);
          } // end of itemStateChanged
        } // end of ItemListener
        );

    // Now, add the box
    this.add( stringBox );
    label = new JLabel( labels[ 0 ] );
    this.add( label );
  } // end of GUIDemo
} // end of class GUIDemo
```

3.16 JList

- A list displays a series of items from which the user may select one or more items.
- Lists are created with class JList, which directly extends class JComponent.
- Supports single-selection lists (only one item to be selected at a time) and multiple-selection lists (any number of items to be selected).
- JLists generate ListSelectionEvents in single-selection lists.

```java
public class GUIDemo extends JFrame {
  private JList colorList; // Holds colors

  // Make arrays for the color names and the colors
  private static String[] names = {"Black","Blue","Cyan","Gray"};
  private static Color[] colors =
          { Color.BLACK, Color.BLUE, Color.CYAN, Color.GRAY };

  public GUIDemo() {
    super("List Test");
    setLayout(new FlowLayout());

    colorList = new JList(names); // Make a new list
    colorList.setVisibleRowCount(3);
    colorList.setFixedCellWidth(100);

    // Do not allow multiple selections:
    colorList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Add a JScrollPane for scrolling functionality
    this.add(new JScrollPane(colorList));

    // Add an action listener
    colorList.addListSelectionListener(new ListSelectionListener() {
      // handle the list selection by changing background color
      public void valueChanged(ListSelectionEvent event) {
        getContentPane().setBackground(
            colors[colorList.getSelectedIndex()]);
      }
    });
  } // end of GUIDemo
} // end of class GUIDemo
```

- JLists have several interesting methods:
  - setVisibleRowCount specifies the number of items visible in the list.
  - setSelectionMode specifies the list's selection mode.
  - Class ListSelectionModel (of package javax.swing) declares selection-mode constants
    - SINGLE_SELECTION (only one item to be selected at a time)
    - SINGLE_INTERVAL_SELECTION (allows selection of several contiguous items)
    - MULTIPLE_INTERVAL_SELECTION (does not restrict the items that can be selected).
- Unlike a JComboBox, a JList does not provide a scrollbar if there are more items in the list than the number of visible rows. A JScrollPane object is used to provide the scrolling capability.
- addListSelectionListener registers a ListSelectionListener (package javax.swing.event) as the listener for aJList's selection events.

3.17 Multiple-Selection Lists

- A **multiple-selection list** enables the user to select many items from a `JList`.
- There are two primary types of multiple-selection lists.
  - A `SINGLE_INTERVAL_SELECTION` list allows selecting a contiguous range of items. To do so, click the first item, then press and hold the Shift key while clicking the last item in the range.
  - A `MULTIPLE_INTERVAL_SELECTION` list (the default) allows continuous range selection as described for a `SINGLE_INTERVAL_SELECTION` list and allows miscellaneous items to be selected by pressing and holding the `Ctrl` key while clicking each item to select. To deselect an item, press and hold the `Ctrl` key while clicking the item a second time.

```java
public class GUIDemo extends JFrame {
  private JList colorList; // Holds colors
  private JList copyList; // Copy color names into new list
  private JButton copyButton; // Button to commence copying

  // Make arrays for the color names and the colors
  private static String[] names = {"Black","Blue","Cyan","Gray"};

  public GUIDemo() {
    super("Multiple-Selection List Test");
    setLayout(new FlowLayout());

    colorList = new JList(names); // Make a new list
    colorList.setVisibleRowCount(3);
    colorList.setFixedCellWidth(100);
    // Now, we want to allow multiple selections:
    colorList.
        setSelectionMode(ListSelectionModel.
   MULTIPLE_INTERVAL_SELECTION);
    this.add(new JScrollPane(colorList));

    copyButton = new JButton("Copy >>>");
    copyButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        // Place the selected values in the copy list
        copyList.
            setListData(colorList.getSelectedValuesList().toArray());
      } // end of actionPerformed
    } // end of ActionListener
        ); // end of AddActionListener
    // Now, add the copy button
    this.add(copyButton);

    // Create the copy list
    copyList = new JList();
    copyList.setVisibleRowCount(3);
    copyList.setFixedCellWidth(100);
    copyList.setFixedCellHeight(15);
    copyList.
       setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
    this.add(new JScrollPane(copyList)); // Add list with scroll pane
  } // end of GUIDemo
} // end of class GUIDemo
```

- If a `JList` does not contain items it will not diplay in a `FlowLayout`. Use `JList` methods `setFixedCellWidth` and `setFixedCellHeight` to set the item width and height

- There are no events to indicate that a user has made multiple selections in a multiple-selection list. If you need this information, check the length of the selection list.
- You can also used an **external event**. An event generated by another GUI component (known as an external event) specifies when the multiple selections in a `JList` should be processed.
- Method `setListData` sets the items displayed in a `JList`.
- Method `getSelectedValues` returns an array of Objects representing the selected items in a `JList`.

3.18 Practice

- Combine your code from the two JList examples to allow for the user to change the background color using the copied-to list. This takes some interesting understanding of the indexing, as well as a use of the `Arrays` class.
- Make sure there are no exceptions thrown by your code.

3.19 Using Card Layout

- Using `CardLayout` (From Oracle's web site)

```java
public class GUIDemo extends JFrame implements ItemListener {

    private JPanel cards; //a panel that uses CardLayout
    private static String BUTTONPANEL = "Card with JButtons";
    private static String TEXTPANEL = "Card with JTextField";

    public GUIDemo() {
        // Put the JComboBox in a JPanel to get a nicer look.
        JPanel comboBoxPanel = new JPanel(); //use FlowLayout
        String comboBoxItems[] = {BUTTONPANEL, TEXTPANEL};
        JComboBox cb = new JComboBox(comboBoxItems);
        cb.setEditable(false);
        cb.addItemListener(this);
        comboBoxPanel.add(cb);

        // Create the "cards"
        JPanel card1 = new JPanel();
        card1.add(new JButton("Button 1"));
        card1.add(new JButton("Button 2"));
        card1.add(new JButton("Button 3"));
        JPanel card2 = new JPanel();
        card2.add(new JTextField("TextField", 20));

        // Create the panel that contains the "cards"
        cards = new JPanel(new CardLayout());
        cards.add(card1, BUTTONPANEL);
        cards.add(card2, TEXTPANEL);

        // Add the panels to the frame
        this.add(comboBoxPanel, BorderLayout.PAGE_START);
        this.add(cards, BorderLayout.CENTER);
    }

    public void itemStateChanged(ItemEvent evt) {
        CardLayout cl = (CardLayout) (cards.getLayout());
        cl.show(cards, (String) evt.getItem());
    }
}
```

3.20 Practice

- Create a new GUI project that uses a card layout.
- Create two panes (cards) that are selectable via radio buttons.
- Put a `JList` in each of the cards.
- Whenever you switch cards from the first card to the second, the highlighted elements of the `JList` on the first card should populate the second card's `JList`.

3.21 Using GridBagLayout

- `https://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html`
- View demo videos "Using GridBagLayout"
- View video `https://www.youtube.com/watch?v=lZxoWZpP52c`
  - There's no sound in this video
  - Ignore the database portion at the beginning...we'll get to that eventually.

3.22 Practice

- Create a new IDE project.
- Create a class called `Pet` with three fields (`name`, `age`, and `species`) and a constructor that sets them.
- Create a GUI frame that uses a `GridBagLayout` that will eventually show 1 row and 3 columns.
- In the first, column, place a GUI that allows the user to input information for a pet.
  1. Create a new `JPanel` using `GridBagLayout`
  2. Add to the `JPanel` labels and buttons that allow the user to input the names, ages, and species of a pet.
  3. Add to the `JPanel` a button for submitting the data. The button should clear the textfields and save the data in a `Pet` object (save an `ArrayList` of them).
- The second column should display a `JList` populated with the names of all the pets.
  1. Store an `ArrayList` of `String` objects as a field of the GUI class used to populate the `JList`
  2. Update the list whenever a pet is added.
  3. Update the `JList` of names. To do this, you can simply repopulate the list's data using `myList.setListData()`.
- The final column (doesn't need a `JPanel` or layout) should display the information for a single pet selected from the list of names in the second column. If the user clicks on a name, that pet's information should appear in a `JTextArea` in the right column.
  1. Display blank information to begin with. It's probably easiest to write a `toString()` method in `Pet` and call and display that.
  2. Write a callback method for the `JList` of pet names so that when one is selected, your program searches the list of `Pet` objects for one with the same name and calls that object's `toString`.
  3. Populate the `JTextArea` with this returned `String` data.
- Add a combo box anywhere on the GUI that allows the user to select between two options: `Multiple Selection` and `Single Selection`.
- Change your `JList` so it allows multiple selections, and all the selected pets' information appears in the `JTextArea`. When the combo box selection is currently `Single Selection`, then the `JList` should only allow single selection, but if the user switches the selection mode to `Multiple Selection` then the `JList` should allow multiple pets to be selected and displayed.
- Change your GUI such that all of the above is on a single card. Add a second card that contains only a `JTextArea` that lists *all* of the pets' information. Allow the user to switch between cards using any method you want.

3.23 Keyboard Events

- To handle keyboard events, we need to implement a `KeyListener` interface.
- Key events are generated when keys on the keyboard are pressed and released.
- A `KeyListener` must define methods `keyPressed`, `keyReleased` and `keyTyped` each of which receives a `KeyEvent` as its argument
- Class `KeyEvent` is a subclass of `InputEvent`.
  - Method `keyPressed` is called in response to pressing any key.
  - Method `keyTyped` is called in response to pressing any key that is not an action key.
  - Method `keyReleased` is called when the key is released after any `keyPressed` or `keyTyped` event.

```
public class GUIDemo extends JFrame implements KeyListener {
    private String line1 = "", line2 = "", line3 = "";
    private JTextArea textArea;

    public GUIDemo() {
        super( "Keyboard Demo" );

        textArea = new JTextArea(10, 15);
        textArea.setText("Press any key.");
        textArea.setEnabled(false);
        textArea.setDisabledTextColor(Color.BLACK);
        this.add( textArea );

        addKeyListener(this);
    }
    @Override
    public void keyTyped(KeyEvent e) {
        line1 = String.format("Key Typed: %c", e.getKeyChar() );
        setLines2And3( e );
    }

    @Override
    public void keyPressed(KeyEvent e) {
        line1 = String.format("Key Pressed: %s",
                KeyEvent.getKeyText( e.getKeyCode() ) );
        setLines2And3( e );
    }

    @Override
    public void keyReleased(KeyEvent e) {
        line1 = String.format("Key Released: %s",
                KeyEvent.getKeyText( e.getKeyCode() ) );
        setLines2And3( e );
    }

    private void setLines2And3( KeyEvent e) {
        line2 = String.format("The key pressed %s an action key",
                (e.isActionKey() ? "is" : "is not"));
        String temp =
                KeyEvent.getModifiersExText(ICONIFIED);
        line3 = String.format( "Modifier keys pressed: %s",
                (temp.equals("") ? "none" : temp ) );
        textArea.setText(String.format("%s\n%s\n%s",
                line1, line2, line3 ) );
    }
}
```

- The `KeyListener` registers key event handlers with method `addKeyListener` from class `Component`.
- `KeyEvent` method `getKeyCode` gets the virtual key code of the pressed key.
- `KeyEvent` contains virtual key-code constants that represents every key on the keyboard.
- Value returned by `getKeyCode` can be passed to `static KeyEvent` method `getKeyText` to get a string containing the name of the key that was pressed.
- `KeyEvent` method `getKeyChar` (which returns a `char`) gets the Unicode value of the character typed.
- `KeyEvent` method `isActionKey` determines whether the key in the event was an action key.
- Method `getModifiers` determines whether any modifier keys (such as `Shift`, `Alt` and `Ctrl`) were pressed when the key event occurred. Result can be passed to `static KeyEvent` method `getKeyModifiersText` to get a string containing the names of the pressed modifier keys.
- `InputEvent` methods `isAltDown, isControlDown, isMetaDown` and `isShiftDown` each return a `boolean` indicating whether the particular key was pressed during the key event.

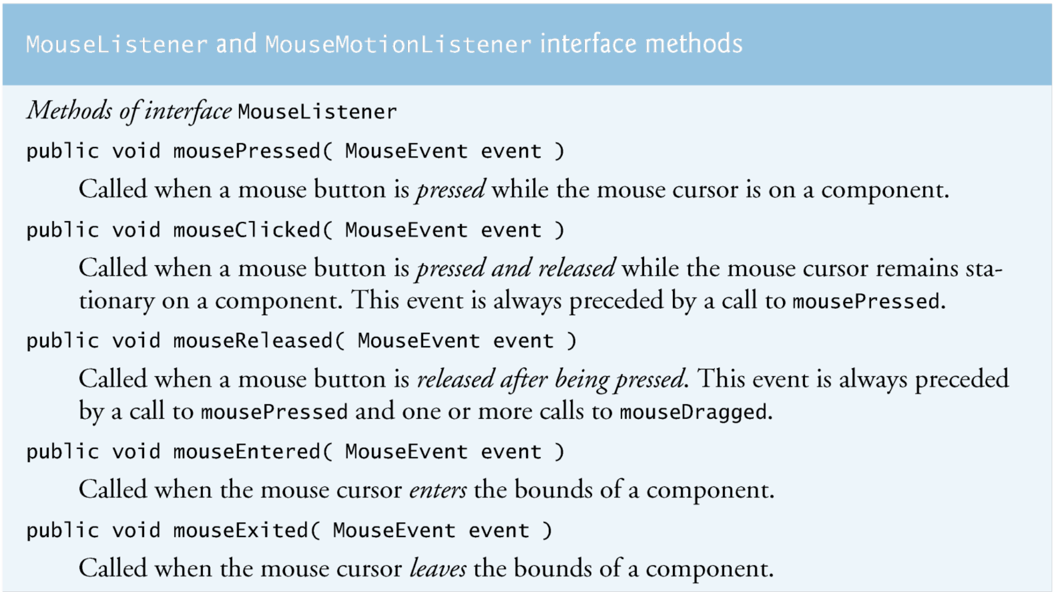
3.24 Mouse Events

- We use the `MouseListener` and `MouseMotionListener` event-listener interfaces for handling mouse events.

- Package `javax.swing.event` contains interface `MouseInputListener`, which extends interfaces `MouseListener` and `MouseMotionListener` to create a single interface containing all the methods.

- `MouseListener` and `MouseMotionListener` methods are called when the mouse interacts with `Component` if appropriate event-listener objects are registered for that `Component`.

- The following is a list of methods used to interface with the mouse:



`MouseListener` and `MouseMotionListener` interface methods

Methods of interface `MouseListener`

`public void mousePressed( MouseEvent event )`

Called when a mouse button is *pressed* while the mouse cursor is on a component.

`public void mouseClicked( MouseEvent event )`

Called when a mouse button is *pressed and released* while the mouse cursor remains stationary on a component. This event is always preceded by a call to `mousePressed`.

`public void mouseReleased( MouseEvent event )`

Called when a mouse button is *released after being pressed*. This event is always preceded by a call to `mousePressed` and one or more calls to `mouseDragged`.

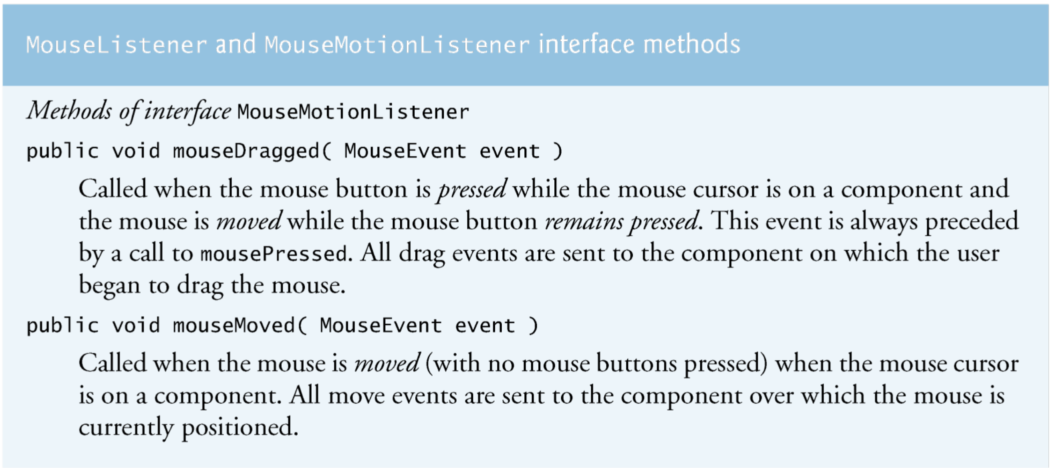
`public void mouseEntered( MouseEvent event )`

Called when the mouse cursor *enters* the bounds of a component.

`public void mouseExited( MouseEvent event )`

Called when the mouse cursor *leaves* the bounds of a component.

Fig. 14.27 | `MouseListener` and `MouseMotionListener` interface methods. (Part 1 of 2.)



`MouseListener` and `MouseMotionListener` interface methods

Methods of interface `MouseMotionListener`

`public void mouseDragged( MouseEvent event )`

Called when the mouse button is *pressed* while the mouse cursor is on a component and the mouse is *moved* while the mouse button *remains pressed*. This event is always preceded by a call to `mousePressed`. All drag events are sent to the component on which the user began to drag the mouse.

`public void mouseMoved( MouseEvent event )`

Called when the mouse is *moved* (with no mouse buttons pressed) when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned.

Fig. 14.27 | `MouseListener` and `MouseMotionListener` interface methods. (Part 2 of 2.)

- Each mouse event-handling method receives a `MouseEvent` object that contains information about the mouse event that occurred, including the x- and y-coordinates of the location where the event occurred.

- Coordinates are measured from the upper-left corner of the GUI component on which the event occurred. The x-coordinates start at 0 and increase from left to right. The y-coordinates start at 0 and increase from top to bottom.

- The methods and constants of class `InputEvent` (`MouseEvent`'s superclass) enable you to determine which mouse button the user clicked.

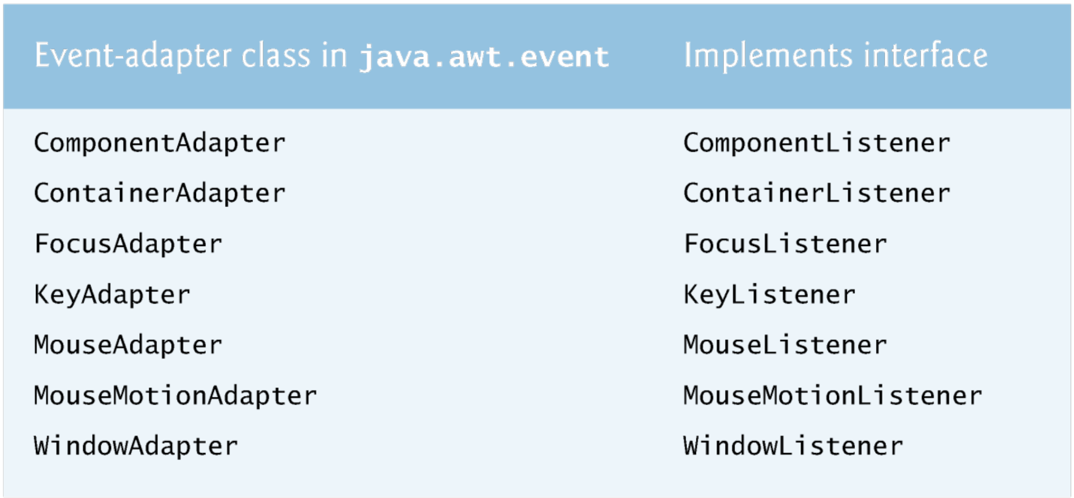
- Interface `MouseWheelListener` enables applications to respond to the rotation of a mouse wheel.
- Method `mouseWheelMoved` receives a `MouseWheelEvent` as its argument.
- Class `MouseWheelEvent` (a subclass of `MouseEvent`) contains methods that enable the event handler to obtain information about the amount of wheel rotation.

```java
public class GUIDemo extends JFrame {
    private JPanel mousePanel;  // Panel which mouse events occur
    private JLabel statusBar;   // Label that displays event info

    public GUIDemo() {
        super( "Demonstrating Mouse Events" );

        mousePanel = new JPanel();
        mousePanel.setBackground( Color.WHITE );
        this.add( mousePanel, BorderLayout.CENTER );

        statusBar = new JLabel("Mouse Outside JPanel");
        this.add( statusBar, BorderLayout.SOUTH );

        // Create and register listener for mouse and mouse motion
        MouseHandler handler = new MouseHandler();
        mousePanel.addMouseListener( handler );
        mousePanel.addMouseMotionListener( handler );
    }
    private class MouseHandler implements MouseListener,
            MouseMotionListener {

        @Override
        public void mouseClicked(MouseEvent e) {
            statusBar.setText(String.format("Clicked at [%d %d]",
                    e.getX(), e.getY()));
        }

        @Override
        public void mousePressed(MouseEvent e) {
            statusBar.setText(String.format("Pressed at [%d %d]",
                    e.getX(), e.getY()));        }

        @Override
        public void mouseReleased(MouseEvent e) {
            statusBar.setText(String.format("Released at [%d %d]",
                    e.getX(), e.getY()));
        }

        @Override
        public void mouseEntered(MouseEvent e) {
            statusBar.setText(String.format("Entered [%d %d]",
                    e.getX(), e.getY()));
            mousePanel.setBackground( Color.GREEN );
        }

        @Override
        public void mouseExited(MouseEvent e) {
            statusBar.setText(String.format("Outside JPanel"));
            mousePanel.setBackground( Color.WHITE );
        }

        @Override
        public void mouseDragged(MouseEvent e) {
            statusBar.setText(String.format("Dragged at [%d %d]",
                    e.getX(), e.getY()));
        }

        @Override
        public void mouseMoved(MouseEvent e) {
            statusBar.setText(String.format("Moved at [%d %d]",
                    e.getX(), e.getY()));        }
    }
}
```

- `BorderLayout` arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER.
- `BorderLayout` sizes the component in the CENTER to use all available space that is not occupied
- Methods `addMouseListener` and `addMouseMotionListener` register `MouseListeners` and `MouseMotionListeners`, respectively.
- `MouseEvent` methods `getX` and `getY` return the x- and y-coordinates of the mouse at the time the event occurred.

3.25 Adapter Classes

- Many event-listener interfaces contain multiple methods.
- An **adapter class** implements an interface and provides a default implementation (with an empty method body) of each method in the interface.
- You extend an adapter class to inherit the default implementation of every method and override only the method(s) you need for event handling.



Event-adapter class in `java.awt.event`	Implements interface
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

3.26 Practice

1. Open your IDE and create a new Java application.
2. Create a new GUI with two `JTextFields`. The one on the left should be editable, but the one on the right should not.
   - When the user right-clicks, whatever text is written in the editable text field should be written in the non-editable text field in all lower case.
   - When the user left-clicks, whatever text is written in the editable text field should be written in the non-editable text field in all upper case.
3. Create a new non-editable `JTextArea`.
   - Each time a keyboard *or* mouse event is triggered, save that event in a `Collection`.
   - Populate the `JTextArea` with the message for each of the events that is triggered, so that it's a running list of all triggered events.
   - Add a `JScrollPane` to allow the user to scroll through all of the event messages after the number of events has exceeded the size of the text area.
4. Create a `JTextField` for the user to input her phone number.
   - Take keyboard input that writes a phone number into the text field, but anytime the user puts an unexpected character (for instance, if there are three digits in the field, the next expected character is a '-'), open a dialogue box with an error message explaining to the user that an error has occurred, *and* informing her of the type of character that was expected when she input the wrong character. This is a form of user feedback.

3.27 Graphics!

This lab (the moving snowman part only):
`http://ict.siit.tu.ac.th/~sun/dw/lib/exe/fetch.php?media=lab09s.pdf`

- The `JPanel` class inherits the method `paintComponent( Graphics g )` from `JComponent`. Anything drawn within `paintComponent` is drawn to the `JPanel`.
- Take a look at an example (from Google Classroom) and watch the associated video

3.28 Practice

- Open your IDE and create a new Java application.
- Create a new class called `MyRectangle` that stores the top-left and bottom-right corner of a Rectangle (needing 4 `int` member variables to represent it). Its constructor should take in values for these for member variables.
- Test this class in `main`.
- Create a `Canvas` class that extends `JPanel`. This class should contain a list of `MyRectangle` objects.
- Update the class so that it takes mouse input such that the user can click and drag her mouse across the JPanel and that creates a new Rectangle, adding it to the Collection of `Rectangle`s. Remember to override the `paintComponent` method inside your `JPanel`. (**NOTE:** Anytime you want the canvas to redraw, you need to call the `repaint()` method.)
- Inside `paintComponent`, cycle through all the `MyRectangle` objects and draw them.
- Update your software so that the user can click a keyboard button and change the color the rectangles are drawn.
  - Add a `color` variable to the `MyRectangle` class.
  - Take keyboard input and change the variable representing the "current color"
  - When a new `MyRectangle` object is created, set the `color` of that `MyRectangle` object to the current color.