Graphical User Interfaces (GUI) (Chapter 12& 13)
UNK CSIT 150

Objectives
• Learn the concepts of JFC, AWT, Swing, and event-driven programming.
• Use basic window components, such as JPanel, JLabel, JTextField, and JButton.
• Learn the steps to create a GUI application, such as creating windows and reacting to events.
• Learn about the layout design and various GUI components.

1. Introduction to GUI

1. JFC
   - Java programmers use Java Foundation Classes (JFC) to create GUI applications
   - These classes can be found in two packages, which are AWT (Abstract Windowing Toolkit) and Swing.

2. AWT
   - `import java.awt.*;`
   - AWT classes do not draw user interface components on the screen.
   - Instead, they rely on OS to draw its own built-in components.
   - Not all OS offer the same set of GUI components.
   - A component may show slightly different behavior on different OS.
   - It is not easy for a programmer to customize a component.
   - Slower.

3. Swing
   - `import javax.swing.*;`
   - Swing classes are not intended to replace AWT.
   - Swing classes draw most of the components on the screen directly. Thus, the components can have consistent look and predictable behavior on any OS, and can be easily customized.
   - Swing classes provide extra components that are not in AWT. But AWT also has something that Swing doesn't provide.
   - Faster.

4. Event-driven programming
   - An object works in the way of reacting events from outside world.
   - An event is an action that takes place within a program
   - Part of the programming is to create event listeners.
   - An event listener is an object that automatically executes one of its methods when a specific event occurs.
   - We often use Swing classes to create components and use AWT classes to handle events.

2. The steps to create a GUI application

1. Creating a Window

   A window is also called a frame that works as a container for other components.
   There are two ways to create a window:
   1. Create one directly from javax.swing.JFrame, and then decorate it.
   2. Derive one from javax.swing.JFrame, and then decorate the derived one.

   See the examples of *ShowWindow.java* and *SimpleWindow.java.*

2. Adding components to a window

- A JFrame object (or an object that extends JFrame) has a *content pane*. This pane is a container for everything that displayed in this frame.
- Sometimes, we need to define a couple of *panels* (see javax.swing.JPanel). We put our stuff in the panels and then put the panels into the *content pane* of the frame.

See the example of *KiloConverterWindow.java and NameWin.*

3. Handling events with listeners

- In the above example, an event is fired when the "calculate" button is hit. Now, what we need to do is to create a listener and then hook the listener to the event source.

(a) Inner class

```
public class Outer {
Fields and methods of the outer class

  private class Inner {
                  Fields and methods of the inner class
  }
}
```

The *private* inner class can only be accessible by fields and methods of the outer class.

(b) Create an event listener

The event listener must implement an appropriate interface to react to the event to be raised. For example, to respond to the event generated by clicking button, we need to implement the `java.awt.event.ActionListener` interface.

```
public interface ActionListener
{
  public void actionPerformed(ActionEvent e);
}
```

Define you own listener, implementing the above interface

```
private class MyButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent e)
  {
        write the event-handling code;
  }
}
```

(c) Adding the above listener to the event source calcButton.

```
calcButton.addActionListener(new CalcButtonListener());
```

Show the example of *KiloConverterWindow.java* again.

(d) Hooking one listener to multiple event sources.

`ActionEvent` has two methods that can be used to identify where this event is from:

- `getActionCommand` returns the action command.
- `getSource` returns the reference to the event source

See example *ColorWindow.java*

3. Layout Managers

3.1 Introduction

- In Java, we don't specify the exact location of a component within a window.
- Instead, we let a layout manager control the positions.
- Three commonly used layout managers:

  – `FlowLayout` Components are arranged in rows. Default for JPanel.
  – `BorderLayout` Components are arranged in five regions: North, South, East, West, and Center. Default for JFrame.
  – `GridLayout` Components are arranged in a grid with rows and columns.

- These layout managers are defined in *java.awt*

- Adding a layout manager to a container:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

3.2 FlowLayout manager

• Constructors of FlowLayout

– *FlowLayout()*
– *FlowLayout(int align).* The values of *align* include LEFT, RIGHT, CENTER, LEADING, and TRAILING.
– *FlowLayout(int align, int hGap, int vGap)* hGap and vGap specify the number of pixels to separate components horizontally and vertically, respectively.

3.3 BorderLayout manager

• We can use *BorderLayout(int hGap, int vGap)* to specify the gap between components.
• The original size of the components is not maintained.
• We usually nest panels in the *BorderLayout.*
• When putting a component into a container with border layout, specify the position. For example:

```
panel.add(btn1, BorderLayout.NORTH);
```

3.4 GridLayout manager

• GridLayout(int rows, int cols)
• All cells have equal size.
• The components are resized.
• Again, we usually nest panels in these cells.

See the example of *WindowLayout.java*

4. Creating radio buttons

The steps to using radio buttons are:

1. Create radio buttons.

```
JRadioButton radio1 = new JRadioButton(“Choice1”);
JRadioButton radio2 = new JRadioButton(“Choice2”, true);
```

2. Put these radio buttons into a group.
```
ButtonGroup group = new ButtonGroup();
group.add(radio1);
group.add(radio2);
```

3. Put these radio buttons into a panel (note: you do *not put the group into a panel*).
```
JPanel panel = new JPanel();
panel.add(radio1);
panel.add(radio2);
```

4. Create a listener to properly respond to the events.

Some useful tidbits:

1. To check the status of a radio button
```
radio.isSelected();
```

2. A radio button can be selected from code
```
radio.doClick();
```

See this process in *MetricConverter.java.*

5. Adding borders to a panel

1. Use BorderFactory to create the border that you like, e.g.,
```
Border b = BorderFactory.createTitledBorder(“button”);
```

2. Set the border of some panel with the newly created one
```
panel.setBorder(b);
```

3. Check out the static methods provided by BorderFactory from *http://java.sun.com/j2se/1.5.0/docs/api/*.

Add border for the radio buttons of example *MetricConverter.java.*

6. Creating check boxes

The steps to using check boxes are:

1. Create the check boxes.
```
JCheckBox check1 = new JCheckBox(“macaroni”);
JCheckBox check2 = new JCheckBox(“spaghetti”, true);
```

2. Add these check boxes to a panel.
```
JPanel panel = new JPanel();
panel.add(check1);
panel.add(check2);
```

3. Create a listener for the events of check boxes
   - Implement `ItemListener` interface
   - Implement method: `public void itemStateChanged(ItemEvent e);`

4. Hookup the listener to the check box.
```
check1.addItemListener(new MyListener());
```

Some useful tidbits:

•The status of the checkbox can be determined by
```
check1.isSelected()
```

• You can select a box from code
```
check1.doClick();
```

See the example of *ColorCheckBoxWindow.java.*

7. Creating a JList

The steps to using a list are:

1. Create a list of items of some type (like String)
```
JList<type> list = new JList<type>();

type[] items = new type[value];
JList<type> list = new JList<type>(items);

 Vector<type> v = new Vector<type>();
 JList<type> list2 = new JList<type>(v);
```

2. Put the list in a scroll pane if it is too long
```
list.setVisibleRowCount(num);
JScrollPane sp = new JScrollPane(list);
JPanel panel = new JPanel(); panel.add(sp);
```

3. Set the selection mode of the list
```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
list.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

4. Build a listener for the list
```
private class MyListener implements ListSelectionListener {
      public void valueChanged(ListSelectionEvent e){
      }
}
```

5. Handling the selected items properly
   - `list.getSelectedValue()` returns the selected item (object) for SINGLE SELECTION, or the first selected item for SINGLE INTERVAL SELECTION and MULTIPLE INTERVAL SELECTION.
   - `list.getSelectedIndex`() returns the index of selected item for SINGLE SELECTION, or the index of the first selected item for SINGLE INTERVAL SELECTION and MULTIPLE INTERVAL SELECTION.
   - `list.getSelectedIndices`() returns the indices of all selected items as an array for SINGLE INTERVAL SELECTION and MULTIPLE INTERVAL SELECTION.

View the process of building lists in *ListWindow.java* and *ListWindowMultiple.java.*

8. Creating a Combo Box

1. A Combo Box is similar to JList.

2. JComboBox raises events that are to be processed by actionPerformed in ActionListener.

3. Selected items are retrieved by box.getSelectedItem() or box.getSelectedIndex().

4. You can make it editable, and type in new items (box.setEditable())

Show students the difference between *ListWindow.java*, and *ComboWindow.java.*

9. Displaying images in labels and buttons

The steps to showing images in a label or button are:

1. Create an image icon

```
ImageIcon image = new ImageIcon("filename");
```

The type of the file has to be JPEG, GIF, and PNG.

2. Put the image icon into a button

```
JButton button = new JButton(Icon image);
JButton button = new JButton(String text, Icon image);
JButton button = new JButton(String text);
button.setIcon(image);
```

3. Put the image icon into a label

```
JLabel label = new JLabel(Icon image);
JLabel label = new JLabel(String text, Icon image);
JLabel label = new JLabel(String text);
label.setIcon(image);
```

See example *ListWindowMultiple.java.*

10. Set mnemonics

1. Also called shortcut keys or hot keys.
2. Used to quickly access a button, a menu item, etc.
3. Check out java.awt.event.KeyEvent for predefined virtual keys.

4. All components inherited from AbstractButton has a method called setMnemonic to set the hot key.

For example, `button.setMnemonic(KeyEvent.VK_X);`

See how it works in *ListWindowMultiple*.java.

11. Set tool tips

1. Show a short description of the components currently under the mouse.

2. All components inherited from JComponent have a method called `setToolTipText(String text)`. For example, `button.setToolTipText("show a short description").`

See how it works in *ListWindowMultiple*.java.

12. Menu

The steps to creating menus are:

1. Create menu items

```
JMenuItem menuItem = new JMenuItem("open file");
JMenuItem menuItem = new JRadioButtonMenuItem("red");
JMenuItem menuItem = new JCheckBoxMenuItem("visible");
```

2. Create menus and put menu items into menus

```
JMenu fileMenu = new JMenu("File");
fileMenu.add(menuItem);
```

3. Separate menu items properly

```
fileMenu.addSeparator();
```

4. Create a menu bar and put the menus in

```
JMenuBar menuBar = new JMenuBar();
menuBar.add(fileMenu);
setJMenuBar(menuBar);
```

5. Create listener for menu items

```
private class MyListener implements ActionListener{
  public void actionPerformed(ActionEvent e){
  }
}
```

6. Hook listener to menu items

```
menuItem.addActionListener(new MyListener());
```

See how it works in *MenuWindow*.java and *SimpleEditor*.java.

13. Choose a color

Open a dialog to choose a color:

```
Color selectedColor = JColorChooser.showDialog(Component
  parent, String title, Color initial);
```

It returns null if cancel button is pressed.

See example *MenuWindow.java*

14. Using Text Areas

1. `JTextField` can only allow users to enter a single line; whereas, `JTextArea` can accept multiple lines of input.

2. Create a text area

```
JTextArea textInput = new JTextArea(int rows, int cols);
JTextArea textInput = new JTextArea(String initial, int rows, int
  cols);
```

3. Obtain the user input
```
String input = textInput.getText();
```

4. Set text into the text area
```
textInput.setText(String info);
```

5. Set scrollbars to the text area
```
JScrollPane sp = new JScrollPane(textInput);
sp.setHorizontalScrollBarPolicy(int policy);
sp.setVerticalScrollBarPolicy(int policy);
```

The available policies for horizontal scroll bar include
- JScrollPane.HORIZONTAL SCROLLBAR AS NEEDED
- JScrollPane.HORIZONTAL SCROLLBAR NEVER
- JScrollPane.HORIZONTAL SCROLLBAR ALWAYS

The available policies for vertical scroll bar include
- JScrollPane.VERTICAL SCROLLBAR AS NEEDED
- JScrollPane.VERTICAL SCROLLBAR NEVER
- JScrollPane.VERTICAL SCROLLBAR ALWAYS

6. Set line wrapping
By default, when the end of a line is reached, the text does not wrap around to the next line
```
textInput.setLineWrap(true);
```

By default, a line is broken between characters. If you want to wrapping happens at words, then
```
textInput.setWrapStyleWord(true);
```

See the example of *ListWindowWithTextArea.java.*

15. Setting the font

All components that inherits from JComponent have a method of setFont.

```
Font f = new Font(String typeface, int Style, int size);
component.setFont(Font f);
```

Java guarantees that you have typefaces of Dialog, DialogInput, Monospaced, SansSerif and Serif (*not case-sensitive*).

- Font class provides constants of Font.PLAIN, Font.BOLD, and Font.ITALIC. You can use Font.BOLD|Font.ITALIC if you want a font to be both *bold* and *italic*

- The *size* is specified in points (72 points = 1inch)

See the example of *ListWindowWithTextArea.java.*

16. Using Sliders

1. Create the slider and decorate it

```
JSlider slider = new JSlider(int orientation, int minValue, int
  maxValue, int initialValue);

slider.setMajorTickSpacing(int spacing);
  slider.setMinorTickSpacing(int spacing);

slider.setPaintTicks(boolean showMinorTicks);
slider.setPaintLabels(boolean showMajorLabels);
slider.setMinimum(int min);
slider.setMaximum(int max);
```

2. Create a listener to handle the changeEvent raised by the slider.

```
private class MyListener implements ChangeListener {
  public void stateChange(ChangeEvent e) {
  ...
  int val = slider.getValue();
  }
}
```

3. Hook up the slider with the listener.

```
slider.addChangeListener(new MyListener());
```

See the example of *TempConverter.java.*

17. Setting the look and feel of windows

1. You want to make the look and feel of you GUI compatible with the Operating System.
2. The default look and feel of Java 5 is called *Ocean.*
3. You can change the look and feel using:

```
try {
  UIManager.setLookAndFeel(String className);
  SwingUtilities.updateComponentTreeUI(Component c);
}
catch (Exception e) {
}
```

• Other look and feels class names include:

Class Name	Look and Feel
javax.swing.plaf.metal.MetalLookAndFeel	*Metal*
com.sun.java.swing.plaf.windows.WindowsLookAndFeel	*Window*
com.sun.java.swing.plaf.motif.MotifLookAndFeel	*Motif*

• Exceptions can be thrown, which include `ClassNotFoundException`, `InstantiationException`, `IllegalAccessException`, and `UnsupportedLookAndFeelException`.

• You need to use `SwingUtilities`.updateComponentTreeUI to change the look and feel of the components that you have created.

Change the look and feel of the application of *TempConverter.*