JAVA PROGRAMMING

Week 7: Swing

Lecturer:

- HO Tuan Thanh, M.Sc.

Plan

1. Origins of Swing
2. Two key Swing features
3. Components and Containers

Introduction

- Swing is the most widely used Java GUI.
- Swing defines a collection of classes and interfaces
  - Support a rich set of visual components, such as buttons, text fields, scroll panes, check boxes, trees, and tables, etc.
- These controls can be used to construct powerful, easy-to-use graphical interfaces.
- Because of its widespread use, Swing is something with which all Java programmers should be familiar.

Origins and design philosophy [1]

- Swing did not exist in the early days of Java.
  - It was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit (AWT).
- AWT defines a basic set of components that support a usable, but limited, graphical interface.
  - One reason: AWT translates its various visual components into their corresponding, platform-specific equivalents, or peers → **heavyweight**.
- Problems of the use of native peers:
  - Threatened the philosophy of Java: write once, run anywhere.
  - The look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed.
  - The use of heavyweight components caused some frustrating restrictions.

Origins and design philosophy [2]

- Swing is a solution for AWT problems.
- Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC).
- Swing was initially available for use with Java 1.1 as a separate library.
- From Java 1.2: Swing (and the rest of JFC) was fully integrated into Java.
- Swing addresses the limitations associated with the AWT's components through the use of two key features:
  - lightweight components and
  - a pluggable look and feel.

Plan

1. Origins of Swing
2. Two key Swing features
3. Components and Containers

Swing components are lightweight

- With very few exceptions, Swing components are lightweight.
  - A component is written entirely in Java and platform-independent → efficiency and flexibility.
  - Lightweight components do not translate into platform-specific peers → the look and feel of each component is determined by Swing, not by the underlying operating system → each component can work in a consistent manner across all platforms.
- Each Swing component is rendered by Java code rather than by platform specific peers. Swing separates the look and feel of a component from the logic of the component → it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component.

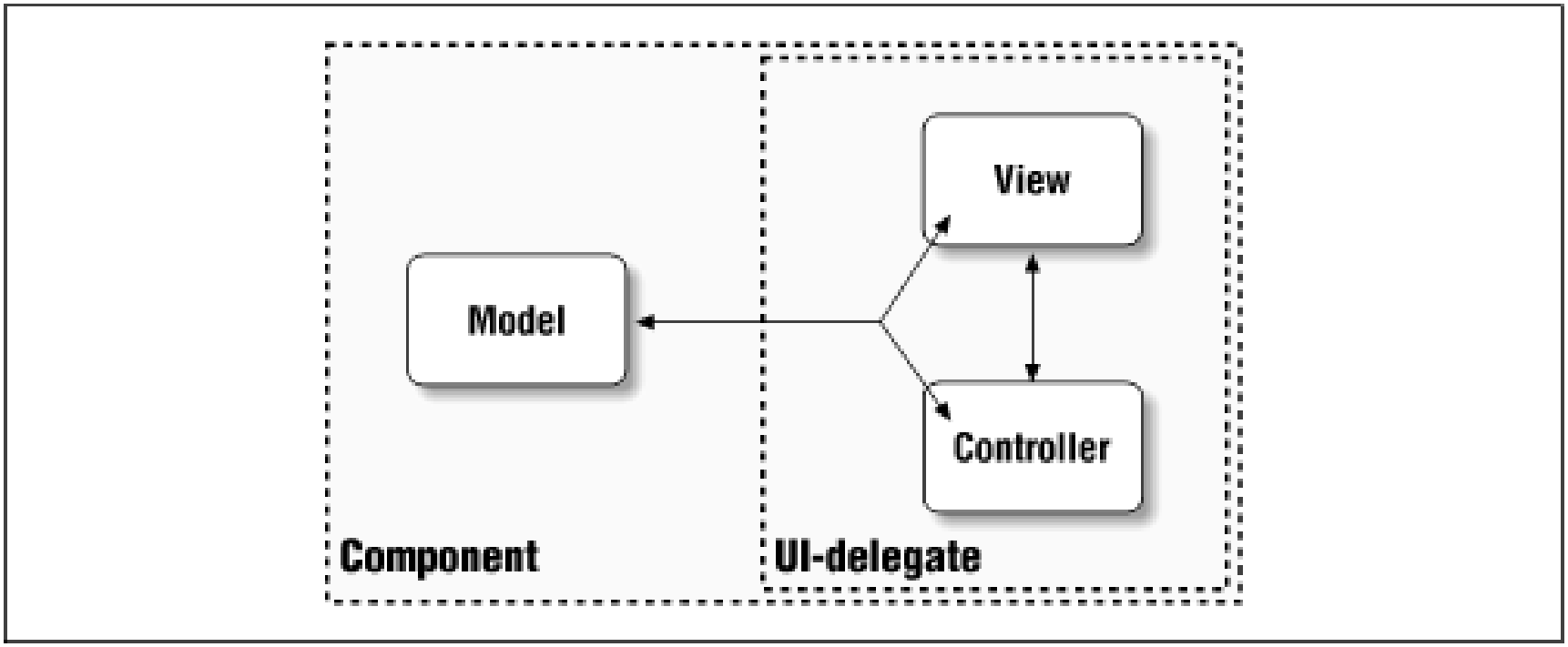
Pluggable look and feels

- Java provides look-and-feels that are available to all Swing users.
  - The metal look and feel is also called the Java look and feel → default look and feel.
  - It is a platform independent look and feel that is available in all Java execution environments.
- Swing's pluggable look and feel is made possible
  - Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate

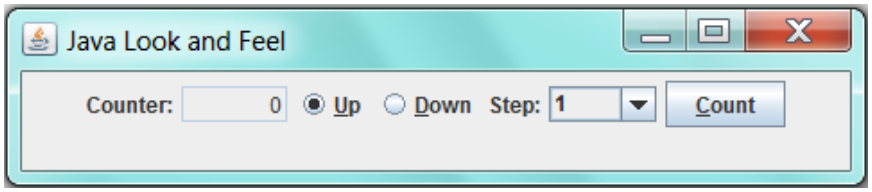
→Swing's approach is called either the model-delegate architecture or the separable model architecture

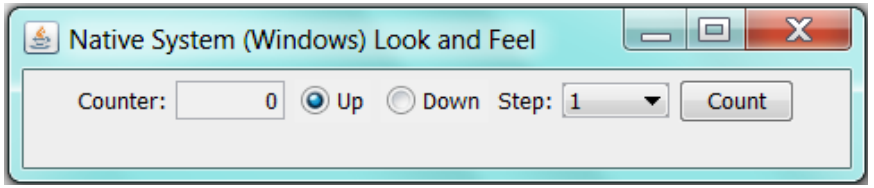
→ although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

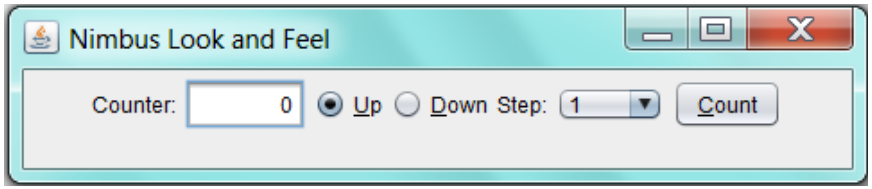
Modified MVC

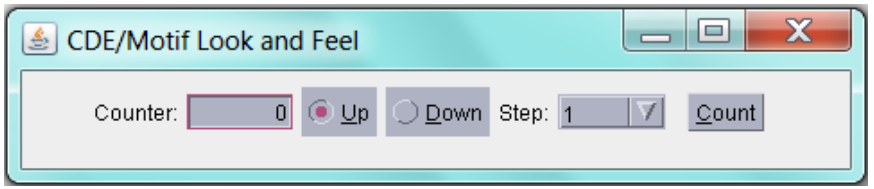


Pluggable look and feels







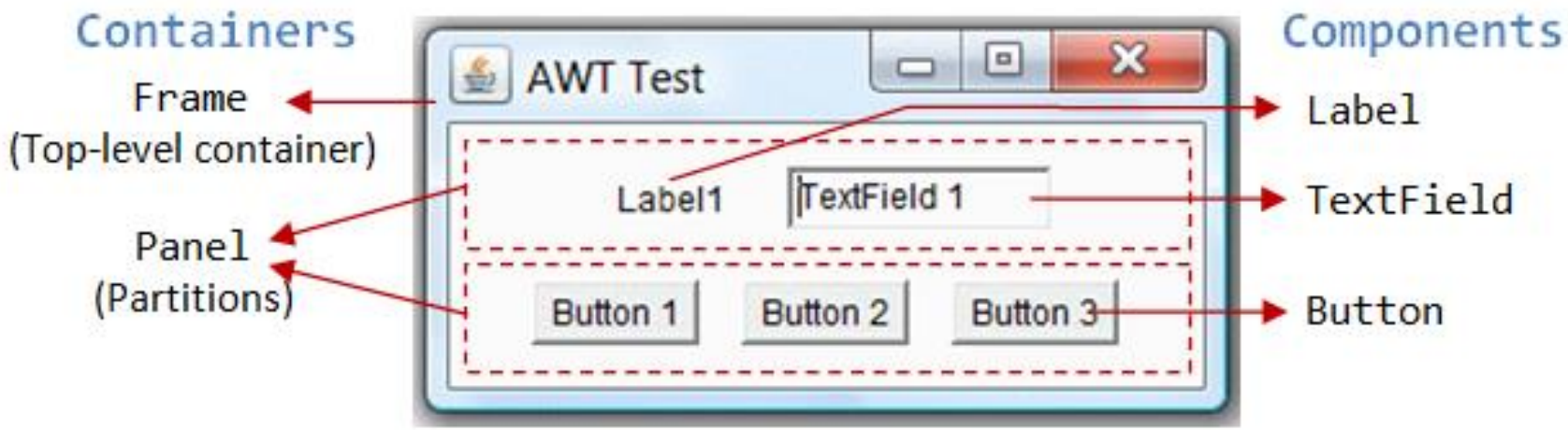


Plan

1. Origins of Swing
2. Two key Swing features
3. Components and Containers

Key items

- A Swing GUI consists of two key items:
  - components and
  - containers.
- This distinction is mostly **conceptual** because all containers are also components.
- The difference is found in their intended purpose:
  - A component is an independent visual control, such as a push button or text field.
  - A container holds a group of components → a container is a special type of component that is designed to hold other components.
  - In order for a component to be displayed, it must be held within a container → all Swing GUIs will have at least one container.


Containers
Frame
(Top-level container)
Panel
(Partitions)
AWT Test
Label1
TextField 1
Button 1
Button 2
Button 3
Components
Label
TextField
Button

Components

- Swing components are derived from the JComponent class.
- JComponent provides the functionality that is common to all components.
- JComponent inherits the AWT classes Container and Component → A Swing component is built on and compatible with an AWT component.
- All of Swing's components are represented by classes defined within the package javax.swing.

Swing components

JApplet (deprecated by JDK 9)	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JTogglebutton	JToolBar	JToolTip	JTree
JViewport	JWindow		

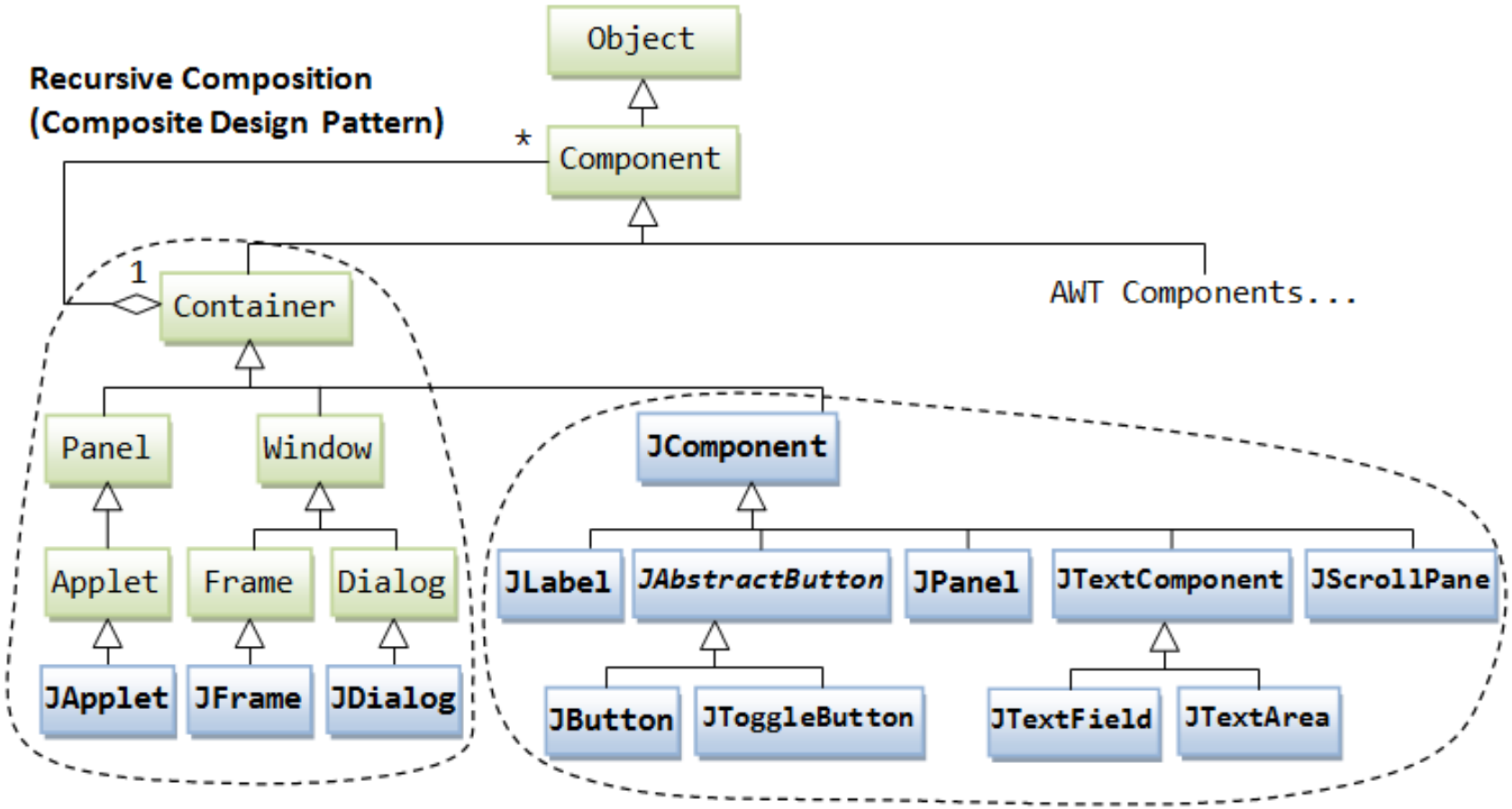
Containers [1]

- Swing defines two types of containers: **Top-level containers** and **lightweight container**.
- Top-level containers:
  - JFrame, JApplet, JWindow, and JDialog. (JApplet has been deprecated since JDK 9)
  - **Do not inherit JComponent**. They **inherit the AWT** classes Component and Container.
  - The top-level containers are heavyweight → This makes the top-level containers a special case in the Swing component library.
  - A top-level container must be at the top of a containment hierarchy → is not contained within any other container.
  - Every containment hierarchy must begin with a top-level container.
  - Most commonly used container: JFrame.

Containers [2]

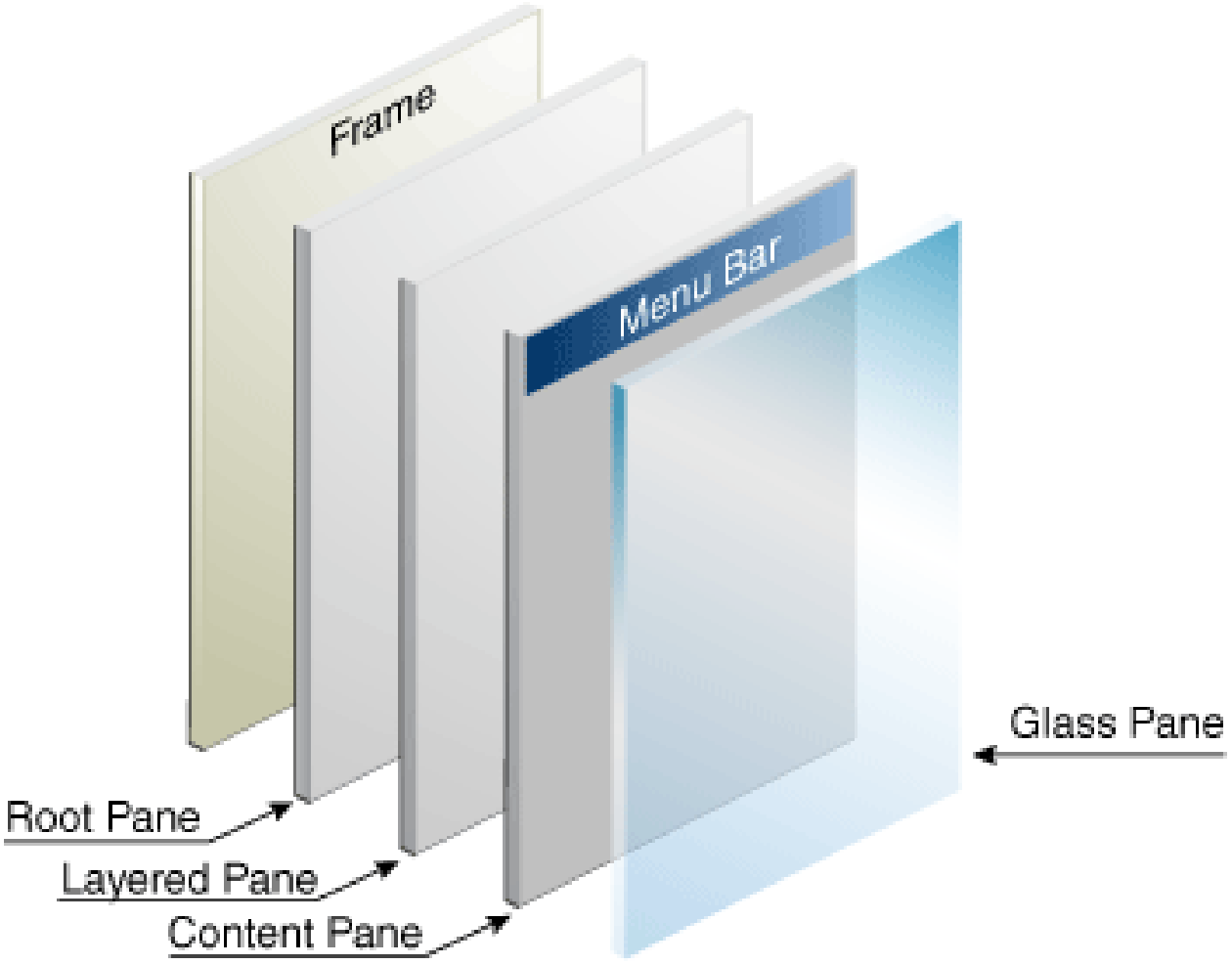
- Lightweight container
  - Inherit JComponent.
  - Example: JPanel, JScrollPane, and JRootPane.
  - Are often used to collectively organize and manage groups of related components because a lightweight container can be contained within another container.

  → use lightweight containers to create subgroups of related controls that are contained within an outer container.


Recursive Composition
(Composite Design Pattern)
Object
*
Component
1
Container
AWT Components...
Panel
Window
Applet
Frame
Dialog
JApplet
JFrame
JDialog
JComponent
JLabel
JAbstractButton
JPanel
JTextComponent
JScrollPane
JButton
JToggleButton
JTextField
JTextArea

Top-Level Container Panes

- Each top-level container defines a set of panes.
- At the top of the hierarchy is an instance of JRootPane.
  - A lightweight container to manage the other panes.
  - Helps manage the optional menu bar.
- The panes that comprise the root pane
  - **The glass pane**
    - Sits above and completely covers all other panes.
    - Enables you to manage mouse events that affect the entire container or to paint over any other component, for example.
  - **The layered pane**
    - Allows components to be given a depth value.
    - Holds the content pane and the (optional) menu bar.
  - **The content pane**
    - Is the one that you will interact the most.
    - Is the pane to which you will add visual components

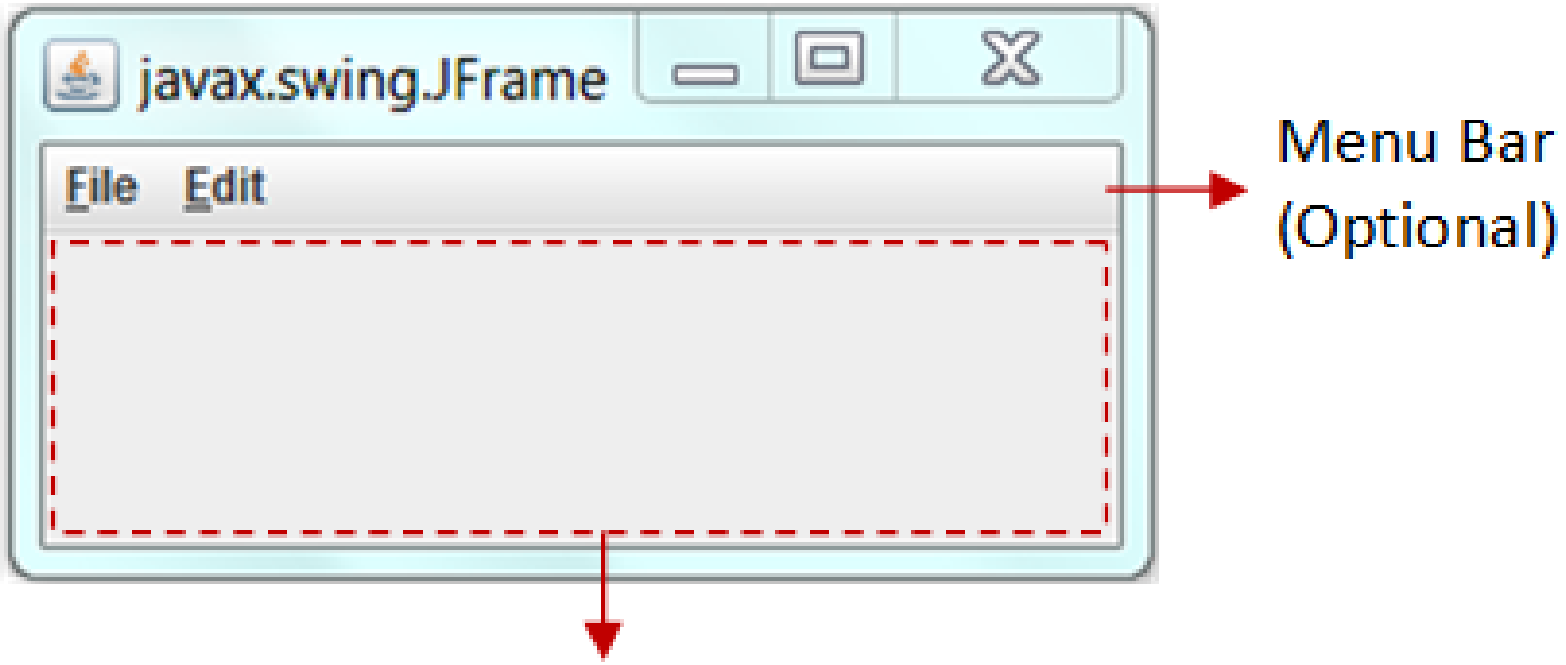

Frame
Menu Bar
Glass Pane
Root Pane
Layered Pane
Content Pane

Swing packages

javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth	javax.swing.tree
javax.swing.plaf	javax.swing.table	javax.swing.undo

A Simple Swing Application [1]

javax.swing.JFrame




```
Container cp = aJFrame.getContentPane();
aJFrame.setContentPane(aPanel);
```

A Simple Swing Application [2]


```
1.  //A simple Swing application.
2.  import javax.swing.*;
3.  class SwingDemo {
4.    SwingDemo() {
5.        // Create a new JFrame container.
6.        JFrame jfrm = new JFrame("A Simple Swing Application");
7.        // Give the frame an initial size.
8.        jfrm.setSize(275, 100);
9.        // Terminate the program when the user closes
10.       // the application.
11.       jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12.       // Create a text-based label.
13.       JLabel jlab = new JLabel(" Swing means powerful GUIs.");
```

A Simple Swing Application [2]

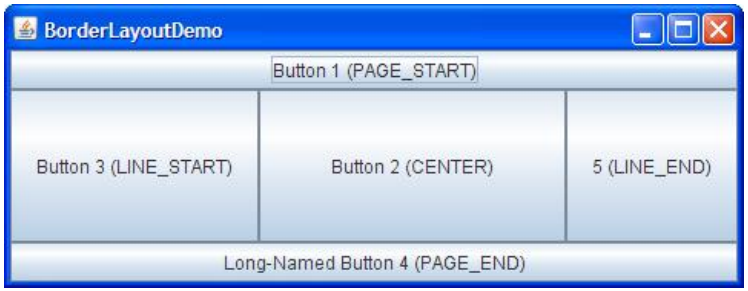

```
14.         // Add the label to the content pane.
15.         jfrm.add(jlab);
16.         // Display the frame.
17.         jfrm.setVisible(true);
18.     }
19.     public static void main(String[] args) {
20.         // Create the frame on the event dispatching thread.
21.         SwingUtilities.invokeLater(new Runnable() {
22.             public void run() {
23.                 new SwingDemo();
24.             }
25.       });
26.     }
27. }
```

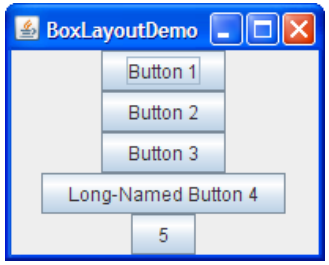
Understanding Layout Managers

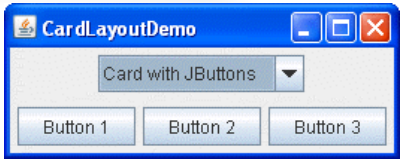
Layout managers

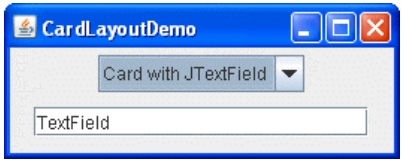
- Controls the position of components within a container.
- Java offers several layout managers.
  - Most are provided by the AWT (within java.awt), but Swing adds a few of its own.
  - All layout managers are instances of a class that implements the LayoutManager interface.

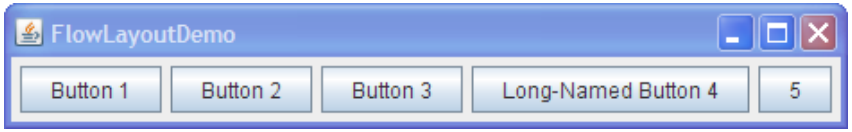
FlowLayout	A simple layout that positions components left-to-right, top-to-bottom. (Positions components right-to-left for some cultural settings.)
BorderLayout	Positions components within the center or the borders of the container. This is the default layout for a content pane.
GridLayout	Lays out components within a grid.
GridBagLayout	Lays out different size components within a flexible grid.
BoxLayout	Lays out components vertically or horizontally within a box.
SpringLayout	Lays out components subject to a set of constraints.

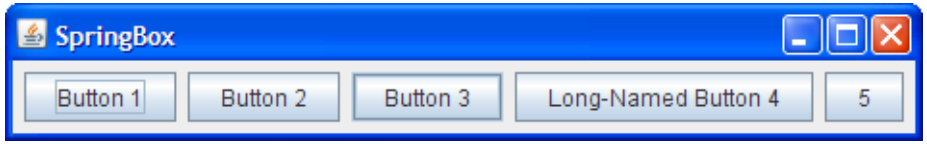


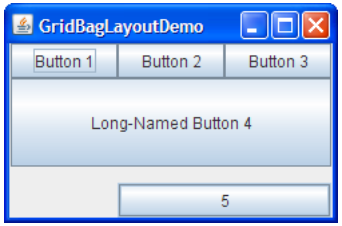


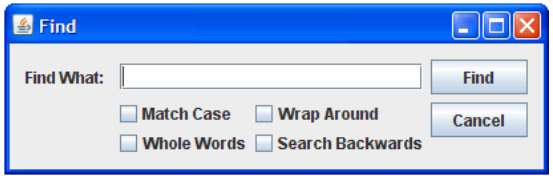




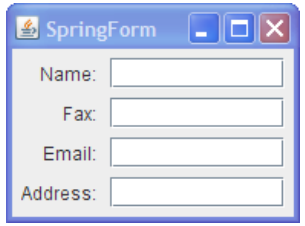








GroupLayout



https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

- Each **Container** object has a layout manager associated with it.
- A layout manager is an instance of any class that implements the **LayoutManager** interface.
  - The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used.

    void setLayout(LayoutManager *layoutObj*)

  - *layoutObj* is a reference to the desired layout manager.
  - If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj* → you will need to determine the shape and position of each component manually, using the **setBounds( )** method defined by **Component**.
  - Normally, you will want to use a layout manager.

- Each layout manager keeps track of a list of components that are stored by their names.
  - The layout manager is notified each time you add a component to a container.
  - Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize( )** and **preferredLayoutSize( )** methods.
  - Each component that is being managed by a layout manager contains the **getPreferredSize( )** and **getMinimumSize( )** methods.

FlowLayout

- FlowLayout implements a simple layout style, which is similar to how words flow in a text editor.
  - The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom → by default, components are laid out line-by-line beginning at the upper-left corner.
- Constructors :

  FlowLayout( )

  FlowLayout(int how)

  FlowLayout(int how, int horz, int vert)

  - Valid values for *how* : **FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT, FlowLayout.LEADING, FlowLayout.TRAILING**
- Example: setLayout(new FlowLayout(FlowLayout.LEFT));

BorderLayout

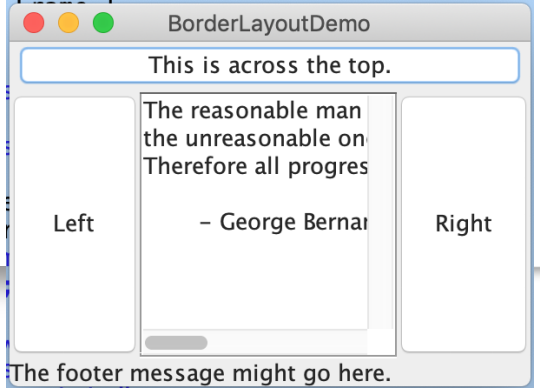
- The **BorderLayout** class implements a layout style that has four narrow, fixed-width components at the edges and one large area in the center.
  - The four sides are referred to as north, south, east, and west. The middle area is called the center.
  - **BorderLayout** is the default layout manager for **Frame**.
- Constructors :

  BorderLayout( )

  BorderLayout(int *horz*, int *vert*)

  - *horz* and *vert* specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.
  - Commonly used constants that specify the regions: **BorderLayout.CENTER, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, BorderLayout.NORTH**

Example: BorderLayout

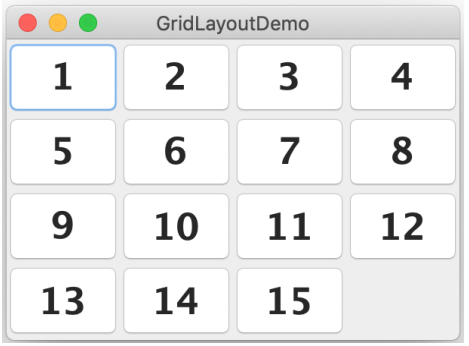



```
1.  // ...
2.  public class BorderLayoutDemo extends Frame {
3.      public BorderLayoutDemo() {
4.          add(new Button("This is across the top."),
5.                                  BorderLayout.NORTH);
6.          add(new Label("The footer message might go here."),
7.                                  BorderLayout.SOUTH);
8.          add(new Button("Right"), BorderLayout.EAST);
9.          add(new Button("Left"), BorderLayout.WEST);
10.         String msg = "The reasonable man adapts ... George Bernard Shaw\n\n";
11.         add(new TextArea(msg), BorderLayout.CENTER);
12.         // ...
13.     }
14.
```

GridLayout

- **GridLayout** lays out components in a two-dimensional grid.
  - When you instantiate a **GridLayout**, you define the number of rows and columns.
- Constructors:
  - GridLayout( )
  - GridLayout(int *numRows*, int *numColumns*)
  - GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)
  - *horz* and *vert* specify the horizontal and vertical space left between components.
  - Either *numRows* or *numColumns* can be zero.
  - Specifying *numRows* as zero allows for unlimited-length columns.
  - Specifying *numColumns* as zero allows for unlimited-length rows.

```
1.  public class GridLayoutDemo extends Frame {
2.      static final int n = 4;
3.      public GridLayoutDemo() {
4.          // Use GridLayout.
5.          setLayout(new GridLayout(n, n));
6.          setFont(new Font("SansSerif", Font.BOLD, 24));
7.          for(int i = 0; i < n; i++) {
8.              for(int j = 0; j < n; j++) {
9.                  int k = i * n + j;
10.                 if(k > 0) add(new Button("" + k));
11.             }
12.         }
13.     // ...
14.
```



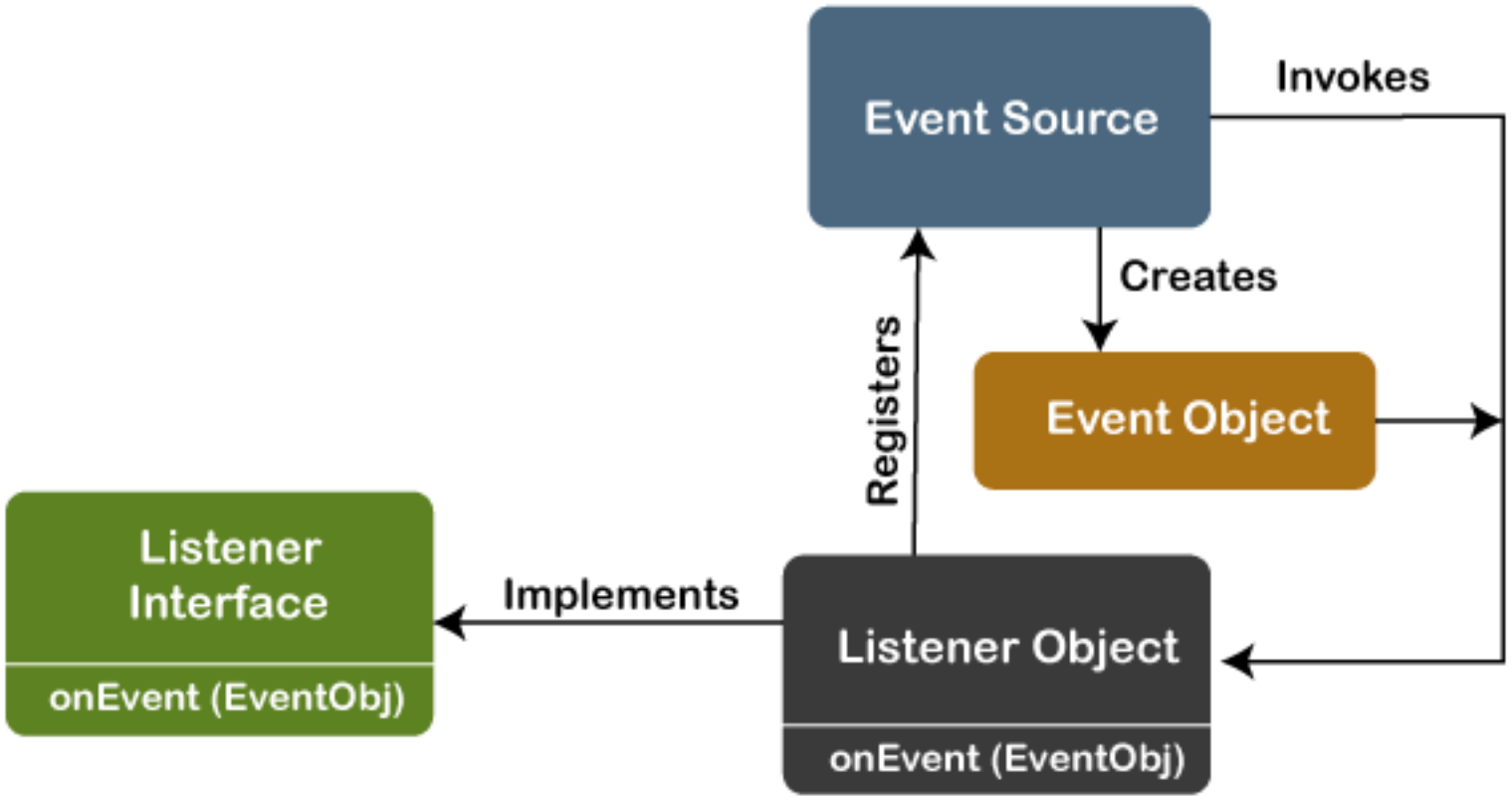
Events

Swing event handling [1]

- Swing programs are event-driven, with components interacting with the program through events.
  - When an event is sent to a program, the program responds to the event by use of an event handler → event handling is an important part of nearly all Swing applications.
- The event handling mechanism used by Swing is called the delegation event model.
  - An event source generates an event and sends it to one or more listeners. With this approach, the listener simply waits until it receives an event.
  - Once an event arrives, the listener processes the event and then returns → the application logic that processes events is cleanly separated from the user interface logic that generates the events.

Swing event handling [2]

- In the delegation event model:
  - Listeners must register with a source in order to receive an event notification.
  - This provides an important benefit: notifications are sent only to listeners that want to receive them.


Event Source
Invokes
Creates
Registers
Event Object
Listener Interface
onEvent (EventObj)
Implements
Listener Object
onEvent (EventObj)

EVENTS	SOURCE	LISTENERS
Action Event	Button, List, MenuItem, Text field	ActionListener
Component Event	Component	Component Listener
Focus Event	Component	FocusListener
Item Event	Checkbox, CheckboxMenuItem, Choice, List	ItemListener
Key Event	when input is received from keyboard	KeyListener
Text Event	Text Component	TextListener
Window Event	Window	WindowListener
Mouse Event	Mouse related event	MouseListener

Events

- An event is an object that describes a state change in an event source.
- It can be generated as a consequence of a person interacting with an element in a graphical user interface or generated under program control.
- Inherit from java.util.EventObject.
- Many events are declared in java.awt.event.
- Events specifically related to Swing are found in javax.swing.event.

Event Sources [1]

- An event source is an object that generates an event.
- When a source generates an event, it sends that event to all registered listeners → the listener must register with the source of that event to receive an event.
- In Swing, listeners register with a source by calling a method on the event source object.
  - Each type of event has its own registration method.
- Typically, events use the following naming convention:

public void addTypeListener(TypeListener el)

  - Type is the name of the event and
  - el is a reference to the event listener.

Event Sources [2]

- A source must also provide a method that allows a listener to unregister an interest in a specific type of event.
- The naming convention of such a method is:

```
public void removeTypeListener(TypeListener el)
```

Event Listeners

- A listener is an object that is notified when an event occurs.
- Two major requirements:
  - It must have registered with one or more sources to receive a specific type of event.
  - It must implement a method to receive and process that event.
- The methods that receive and process events applicable to Swing are defined in a set of interfaces, such as those found in java.awt.event and javax.swing.event.
- An event handler should do its job quickly and then return.
  - In most cases, it should not engage in a long operation because doing so will slow down the entire application.
  - If a time-consuming operation is required, then a separate thread should be created for this purpose.

Event Classes and Listener Interfaces

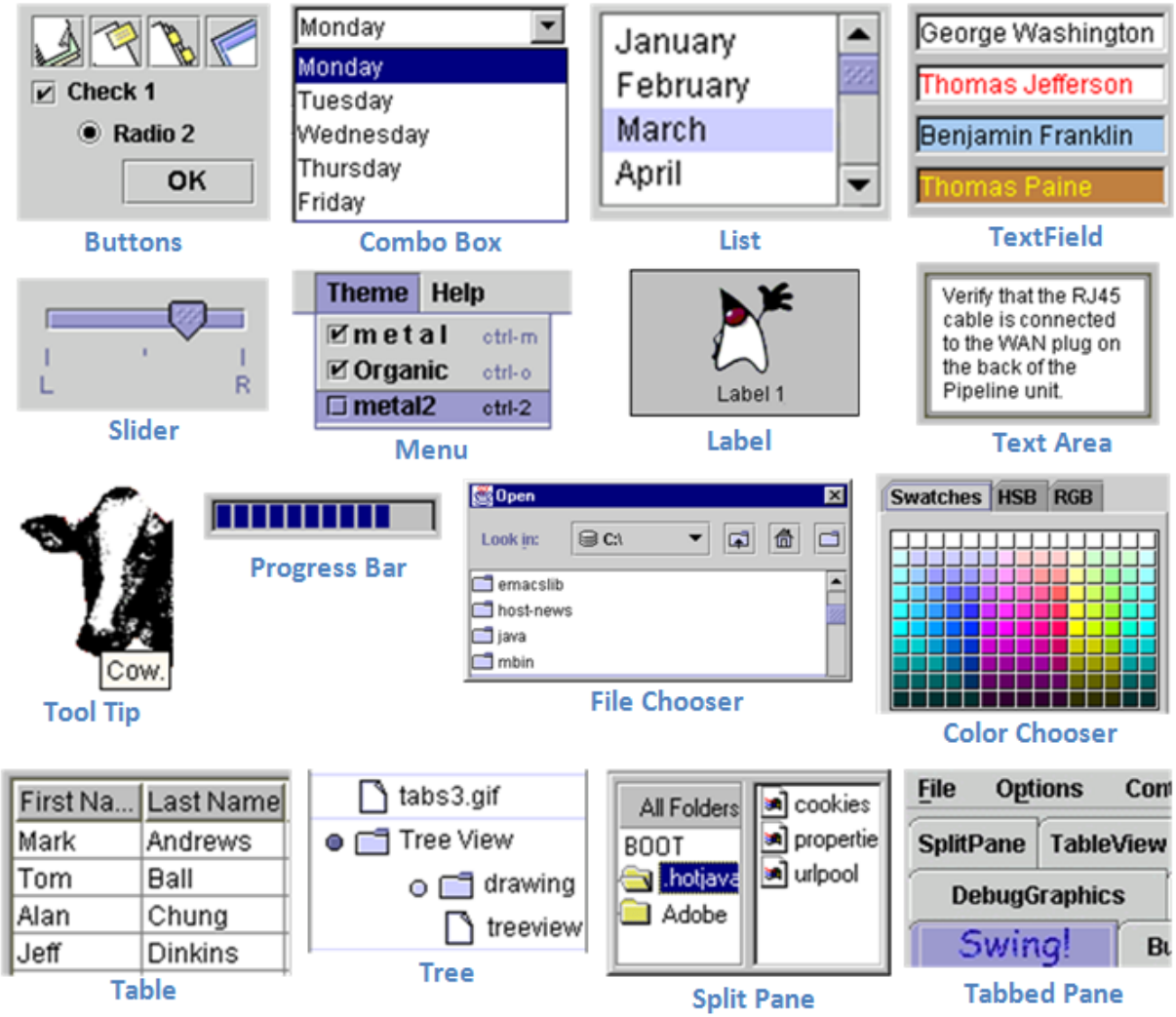
- At the root of the event class hierarchy is java.util.EventObject.
  - It is the superclass for all events in Java.
  - The class java.awt.AWTEvent is a subclass of EventObject. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.
- Although Swing uses the AWT events, it also adds several of its own: javax.swing.event →Swing supports a large number of events.

Commonly Used Event Listener Interfaces

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

GUI components

GUI components



JLabel

- Is Swing's easiest-to-use component.
- Can be used to display text and/or an icon.
- It is a passive component in that it does not respond to user input.
- Constructors:

  JLabel(Icon icon)

  JLabel(String str)

  JLabel(String str, Icon icon, int align)

  - str and icon are the text and icon used for the label.
  - align specifies the horizontal alignment of the text and/or icon within the dimensions of the label.
  - Its values: LEFT, RIGHT, CENTER, LEADING, or TRAILING.

ImageIcon

- ImageIcon implements Icon and encapsulates an image.
  - An object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor.
- Constructors:

  ImageIcon(String filename)
  - It obtains the image in the file named filename.
- Methods:

  Icon getIcon( )

  String getText( )

  void setIcon(Icon icon)

  void setText(String str)

Example


```
1.  public class JLabelDemo extends JFrame {
2.      public JLabelDemo() {
3.          Container cp = getContentPane();
4.          ImageIcon ii = new ImageIcon("hourglass.png");
5.          JLabel jl = new JLabel("Hourglass", ii,
6.                                                  JLabel.CENTER);
7.          cp.add(jl);
8.          setSize(200, 200);
9.          setVisible(true);
10.     }
11.     public static void main(String[] args) {
12.         new JLabelDemo();
13.     }
14. }
```

JTextField

JTextField(int cols)

JTextField(String str, int cols)

JTextField(String str)

- cols specifies the width of the text field in columns.
- str is the string to be initially presented

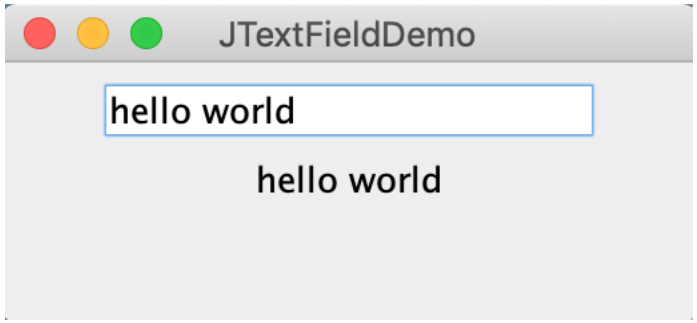
- When you press enter when inputting into a text field, an ActionEvent is generated → JTextField provides the addActionListener() and removeActionListener() methods.
- To handle action events: implement the actionPerformed() method defined by the ActionListener interface.
- To obtain the string: String getText()
- You can set the text void setText(String text)

Example: JTextField [1]


```
1.  public class JTextFieldDemo {
2.      public JTextFieldDemo() {
3.          JFrame jfrm = new JFrame("JTextFieldDemo");
4.          jfrm.setLayout(new FlowLayout());
5.          jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6.          jfrm.setSize(260, 120);
7.          JTextField jtf = new JTextField(15); jfrm.add(jtf);
8.          JLabel jlab = new JLabel(); jfrm.add(jlab);
9.          jtf.addActionListener(new ActionListener() {
10.           public void actionPerformed(ActionEvent ae) {
11.                   jlab.setText(jtf.getText());
12.           }
13.         });
14.         jfrm.setVisible(true);
15.     }
```

Example: JTextField [2]


```
15.     // ...
16.         public static void main(String[] args) {
17.             SwingUtilities.invokeLater(
18.                 new Runnable() {
19.                     public void run() {
20.                         new JTextFieldDemo();
21.                     }
22.                 }
23.             );
24.         }
25.     }
```



Swing Buttons

- Swing defines four types of buttons:
  - JButton,
  - JToggleButton,
  - JCheckBox, and
  - JRadioButton.
- All are subclasses of the AbstractButton class, which extends JComponent.

AbstractButton methods

- AbstractButton contains many methods that allow you to control the behavior of buttons.
- Methods set icons:

  void setDisabledIcon(Icon di)

  void setPressedIcon(Icon pi)

  void setSelectedIcon(Icon si)

  void setRolloverIcon(Icon ri)

  - *di, pi, si*, and *ri* are the icons to be used for the indicated purpose.
- Text associated with a button can be read and written by:

  String getText( )

  void setText(String *str*)

  - *str* is the text to be associated with the button.

JButton

- JButton class provides the functionality of a push button.
- JButton allows an icon, a string, or both to be associated with the push button.
- Constructors:

  JButton(Icon icon)

  JButton(String str)

  JButton(String str, Icon icon)

  - *str* and *icon* are the string and icon used for the button.

- JButton provides methods to add or remove an action listener:

  void addActionListener(ActionListener al)

  void removeActionListener(ActionListener al)

  - al specifies an object that will receive event notifications.
  - This object must be an instance of a class that implements the ActionListener interface.

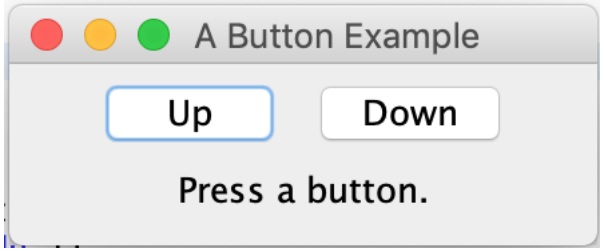
- The ActionListener interface defines only one method:

void actionPerformed(ActionEvent ae)

  - This method is called when a button is pressed.
  - Using the ActionEvent object passed to actionPerformed(), you can obtain several useful pieces of information relating to the button-press event.

```
1.  class ButtonDemo implements ActionListener {
2.      JLabel jlab;
3.      ButtonDemo() {
4.          JFrame jfrm = new JFrame("A Button Example");
5.          jfrm.setLayout(new FlowLayout());
6.          jfrm.setSize(220, 90);
7.          jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8.          JButton jbtnUp = new JButton("Up");
9.          JButton jbtnDown = new JButton("Down");
10.         jbtnUp.addActionListener(this);
11.         jbtnDown.addActionListener(this);
12.         jfrm.add(jbtnUp);
13.         jfrm.add(jbtnDown);
14.         jlab = new JLabel("Press a button.");
15.         jfrm.add(jlab);
16.         jfrm.setVisible(true);
17.     }
18.     // ....
```

```
1.          //...
2.          public void actionPerformed(ActionEvent ae) {
3.              if (ae.getActionCommand().equals("Up"))
4.                  jlab.setText("You pressed Up.");
5.              else
6.                  jlab.setText("You pressed down. ");
7.          }
8.          public static void main(String args[]) {
9.              // Create the frame on the event dispatching thread.
10.             SwingUtilities.invokeLater(new Runnable() {
11.                 public void run() {
12.                     new ButtonDemo();
13.                 }
14.             });
15.         }
16.     }
```



```
1.  class ButtonDemo extends JFrame {
2.      JLabel jlab;
3.      ButtonDemo() {
4.          setTitle("A Button Example");
5.          Container container = getContentPane();
6.          setLayout(new FlowLayout());
7.          setSize(220, 90);
8.          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9.          JButton jbtnUp = new JButton("Up");
10.         JButton jbtnDown = new JButton("Down");
11.         jbtnUp.addActionListener(new ActionListener() {
12.             @Override
13.             public void actionPerformed(ActionEvent e) {
14.                 jlab.setText("You pressed Up.");
15.             }
16.         });
17.         //....
```

```
1.              //....
2.              jbtnDown.addActionListener(new ActionListener() {
3.                  @Override
4.                  public void actionPerformed(ActionEvent e) {
5.                      jlab.setText("You pressed down.");
6.                  }
7.              });
8.              add(jbtnUp);
9.              add(jbtnDown);
10.             jlab = new JLabel("Press a button.");
11.             add(jlab);
12.             setVisible(true);
13.         }
14.     }
```

Create a JCheckBox

- JCheckBox inherits AbstractButton and JToggleButton → a special type of button.

JCheckBox(String str)

- When a check box is selected or deselected (that is, checked or unchecked), an item event is generated.
- Item events are represented by the ItemEvent class.
- Item events are handled by classes that implement the ItemListener interface.

void itemStateChanged(ItemEvent ie)

- To obtain a reference to the item that changed:

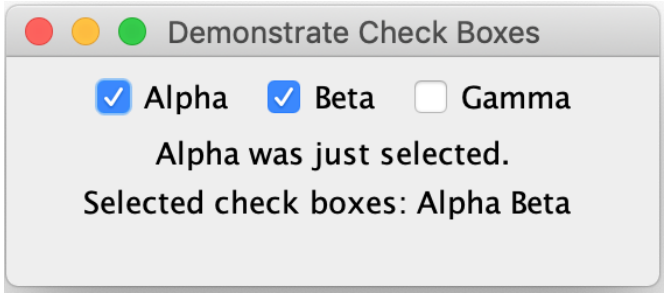
  Object getItem( )

- To obtain the text associated with a check box by calling getText( ).
- You can set the text after a check box is created by calling setText( ).
- The easiest way to determine the state of a check box:

  boolean isSelected( )

  - It returns true if the check box is selected and false otherwise.

```
1.  class CBDemo implements ItemListener {
2.      JLabel jlabSelected; JLabel jlabChanged;
3.      JCheckBox jcbAlpha;
4.      JCheckBox jcbBeta; JCheckBox jcbGamma;
5.      CBDemo() {
6.          JFrame jfrm = new JFrame("Demonstrate Check Boxes");
7.          jfrm.setLayout(new FlowLayout());jfrm.setSize(280, 120);
8.          jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9.          jlabSelected = new JLabel("");
10.         jlabChanged = new JLabel("");
11.         jcbAlpha = new JCheckBox("Alpha");
12.         jcbBeta = new JCheckBox("Beta");
13.         jcbGamma = new JCheckBox("Gamma");
14.         jcbAlpha.addItemListener(this);
15.         jcbBeta.addItemListener(this);
16.         jcbGamma.addItemListener(this);
17.         jfrm.add(jcbAlpha); jfrm.add(jcbBeta);
18.         jfrm.add(jcbGamma); jfrm.add(jlabChanged);
19.         jfrm.add(jlabSelected); jfrm.setVisible(true);
20.     }
```



```
1.      public void itemStateChanged(ItemEvent ie) {
2.          String str = "";
3.          JCheckBox cb = (JCheckBox) ie.getItem();
4.          if (cb.isSelected()) jlabChanged.setText(cb.getText()
5.                                  + " was just selected.");
6.          else jlabChanged.setText(cb.getText() +
7.                                                  " was just cleared.");
8.          if (jcbAlpha.isSelected()) { str += "Alpha "; }
9.          if (jcbBeta.isSelected()) { str += "Beta "; }
10.         if (jcbGamma.isSelected()) { str += "Gamma"; }
11.         jlabSelected.setText("Selected check boxes: " + str);
12.     }
13.     public static void main(String args[]) {
14.         SwingUtilities.invokeLater(new Runnable() {
15.             public void run() { new CBDemo(); }
16.         });
17.     }
18. }
```

JList

- This is Swing's basic list class.
- It supports the selection of one or more items from a list.
- JList is a generic class that is declared as below :

```
class JList<E>
```

- E represents the type of the items in the list.
- Constructor:

```
JList(E[] items)
```

- It is possible to wrap a JList inside a JScrollPane, which is a container that automatically provides scrolling for its contents.

JScrollPane(Component comp)

- comp specifies the component to be scrolled, which in this case will be a JList.
- When you wrap a JList in a JScrollPane, long lists will automatically be scrollable.

- A JList generates a ListSelectionEvent when the user makes or changes a selection.
  - This event is also generated when the user deselects an item.
  - It is handled by implementing ListSelectionListener, which is packaged in javax.swing.event.
  - This listener specifies only one method, called valueChanged():

void valueChanged(ListSelectionEvent le)

- By default: a JList allows the user to select multiple ranges of items within the list, but you can change this behavior:

  void setSelectionMode(int mode)

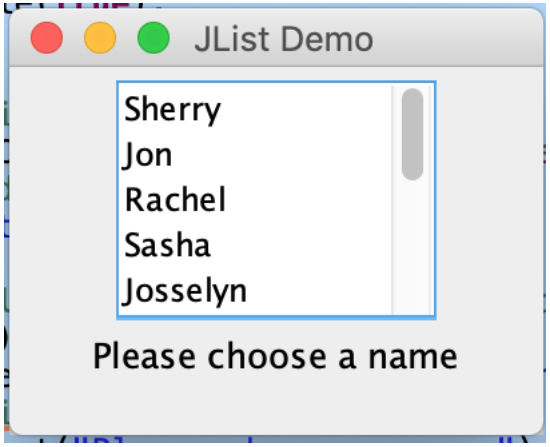
  - mode specifies the selection mode.
  - It must be one of these values defined by the ListSelectionModel interface (which is packaged in javax.swing):
  - SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, MULTIPLE_INTERVAL_SELECTION

int getSelectedIndex( )

int[] getSelectedIndices( )

```
1.  class ListDemo implements ListSelectionListener {
2.          JList<String> jlst; JLabel jlab; JScrollPane jscrlp;
3.          String names[] = { "Sherry", "Jon", "Rachel", "Sasha",
4.          "Josselyn", "Randy", "Tom", "Mary", "Ken", "Andrew",
5.          "Matt", "Todd" };
6.          ListDemo() {
7.              JFrame jfrm = new JFrame("JList Demo");
8.              jfrm.setLayout(new FlowLayout());
9.              jfrm.setSize(200, 160);
10.             jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11.             jlst = new JList<String>(names);
12.             jlst.setSelectionMode(
13.                             ListSelectionModel.SINGLE_SELECTION);
14.             jscrlp = new JScrollPane(jlst);
15.             jscrlp.setPreferredSize(new Dimension(120, 90));
16.             jlab = new JLabel("Please choose a name");
17.             jlst.addListSelectionListener(this);
18.             jfrm.add(jscrlp); jfrm.add(jlab);
19.             jfrm.setVisible(true);
20.         }
```

```
1.      public void valueChanged(ListSelectionEvent le) {
2.          int idx = jlst.getSelectedIndex();
3.          if (idx != -1)
4.              jlab.setText("Current selection: " + names[idx]);
5.          else
6.              jlab.setText("Please choose an name");
7.      }
8.      public static void main(String args[]) {
9.          // Create the frame on the event dispatching thread.
10.         SwingUtilities.invokeLater(new Runnable() {
11.             public void run() {
12.                 new ListDemo();
13.             }
14.         });
15.     }
16. }
```



QUESTION ?