COURSE:- Java Programming
Sem:- III
Module: 6

By:
Dr. Ritu Jain,
Associate Professor,

Swing and Event Handling

AWT and Swing Class

- Swing is a set of classes that is used to *to develop GUI or* create window-based applications *in java*.
- Swing is built on the top of Abstract Window Toolkit (AWT) and entirely written in java. It is platform independent.
- The **javax.swing** package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu etc.
- Swing uses MVC architecture.

AWT and Swing

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Features of Swing Class

- **Pluggable look and feel:** 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.
- **Lightweight Components:** Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible.
- **Purely Java-based (Lightweight Components):** Swing components are written entirely in Java. Do **not depend on native OS components**, unlike AWT.
- **Platform Independent:** Swing are written in java, therefore it is platform independent.
- **MVC architecture**

Features ofSwing Class

- **Rich Set of GUI Components**
  - Swing provides a wide variety of UI components beyond AWT, such as: JButton, JLabel, JTextField, JTextArea, JTable, JList, Jtree, JMenuBar, JToolBar, JColorChooser, JFileChooser
  - This allows developers to build complex professional GUIs.
- **Highly Customizable:** Components can be customized by Subclassing, Overriding paintComponent(), Using custom renderers (JTable, JList). Helps in creating modern GUI designs.
- **Event-Driven Programming: S**wing follows a **delegation event model** similar to AWT:
  - ActionListener
  - MouseListener
  - KeyListener
  - WindowListener, etc.
  - Makes GUI interactive and responsive.

Features of Swing Class

- **Double Buffering:** Swing uses **double buffering** to reduce flickering. Smooth rendering of animations, graphics, and component updates.

- **Support for Advanced Components:** Swing provides advanced interactive components: JTree for hierarchical data, JTable for tabular data, Built-in dialogs: JOptionPane
- **Improved Layout Management:** Swing supports multiple powerful layout managers: BorderLayout, GridBagLayout, BoxLayout, FlowLayout, GroupLayout. Allows building complex and flexible UI layouts.

Swing

Swing is a lightweight, platform-independent GUI toolkit in Java. It provides a rich set of components, supports pluggable look-and-feel, and follows a modified MVC architecture. Swing is highly customizable, uses double buffering for smooth rendering, and provides advanced components like JTable and JTree. It supports event-driven programming and integrates with Java 2D for custom graphics.

MVC Architecture

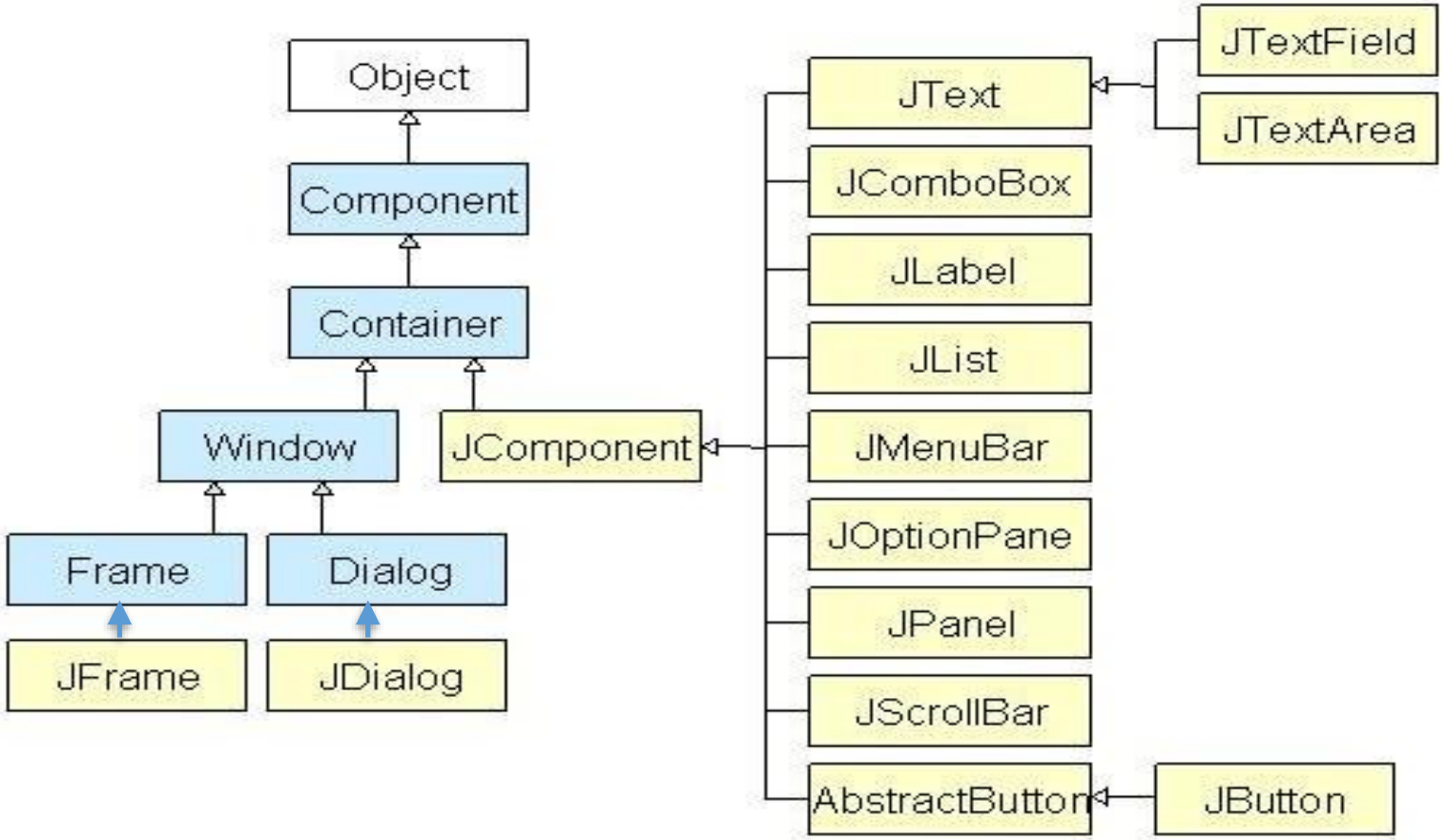
- In general, a visual component is a composite of **three distinct aspects:**
  1. The way that the component looks when rendered on the screen.——>View
  2. The way such that the component reacts to the user. ——————-> Controller
  3. The state information associated with the component. ——————> Model
- In MVC architecture, each piece of the design corresponds to an aspect of a component.
  - *model* corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.
  - *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.
  - *controller* determines how the component reacts to the user.

- For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two.

MVC Architecture

- Although MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components.
- Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the ***UI delegate*.**
- For this reason, Swing's approach is called either the ***Model-Delegate*** **architecture or** ***Separable Model*** architecture. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

Hierarchy Of Swing Classes



Components and Containers

- A Swing GUI consists of two key items:
  - *components* and
  - *containers*.
- However, this distinction is mostly conceptual because all containers are also components.
- *A component* is an independent visual control, such as a button or checkbox.
- A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components.
- Furthermore, in order for a component to be displayed, it must be held within a container.
- Thus, all Swing GUIs will have at least one container.
- A container can also hold other containers. This is known as *containment hierarchy*.

Components

- In general, Swing components are derived from the **JComponent** class. (The only exceptions to this is the top-level container **Jframe, JApplet**, **JWindow**, and **JDialog**.).
- **JComponent** provides the functionality that is common to all components.
- All of Swing's components are represented by classes defined within the package **javax.swing**.

Containers

- Swing defines two types of containers:
  - **Top level containers (Heavy Weight)**: **JFrame**
    - Do not inherit **Jcomponent class**
    - Top-level container is not contained within any other container.
  - **Light weight containers: JPanel**
    - inherit **Jcomponent class**
    - often used to organize and manage groups of related components
    - lightweight container can be contained within another container.
    - **JPanel** is used to create subgroups of related controls that are contained within an outer container.

Commonly used Methods of Component class

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

JFrame

- A JFrame is an independent window that can, for example, act as the main window of an application.
- JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.
- Constructors:

Constructor	Description
JFrame()	It constructs a new frame that is initially invisible.
JFrame(String title)	It creates a new, initially invisible Frame with the specified title.

Jframe Methods

Methods	Description
setTitle(String title)	Sets the title of the JFrame.
setSize(int width, int height)	Sets the size of the JFrame.
setDefaultCloseOperation(int operation)	•Defines the action performed when the user clicks the close button. Common options include: •JFrame.EXIT_ON_CLOSE: Exits the entire application (recommended for applications). •JFrame.HIDE_ON_CLOSE: Hides the frame but the application keeps running. •JFrame.DISPOSE_ON_CLOSE: Disposes of the frame's resources but may not exit the application.
setVisible(boolean b)	Sets the visibility of the JFrame. Pass true to make it visible and false to hide it.
setBounds(int x, int y, int width, int height)	Positions and resizes the frame in a single call using specified coordinates and dimensions.
setLayout(LayoutManager manager)	Sets the layout manager for the JFrame, which controls how components are arranged within the frame.
add(Component comp)	Adds a Swing component to the JFrame.

Two ways to create a frame

There are two ways to create a frame:

- By creating the object of Frame class (association)
  - Refer java program: Ex1_JFrameAss.java, Ex2_JFrameAssCon.java
- By extending Frame class (inheritance)
  - Refer java program: Ex3_JFrameInh.java

setDefaultCloseOperation( )

- By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated.
- If you want the entire application to terminate when its top-level window is closed, call **setDefaultCloseOperation( ) on the reference of frame, if jfrm is the instance of JFrame**:
  - jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
- After this call executes, closing the window causes the entire application to terminate.

JComboBox

- JComboBox is a part of Java Swing package.
- JComboBox inherits JComponent class .
- JComboBox shows a popup menu that shows a list and the user can select a option from that specified list .
- JComboBox can be editable or read- only depending on the choice of the programmer .

Event Handling

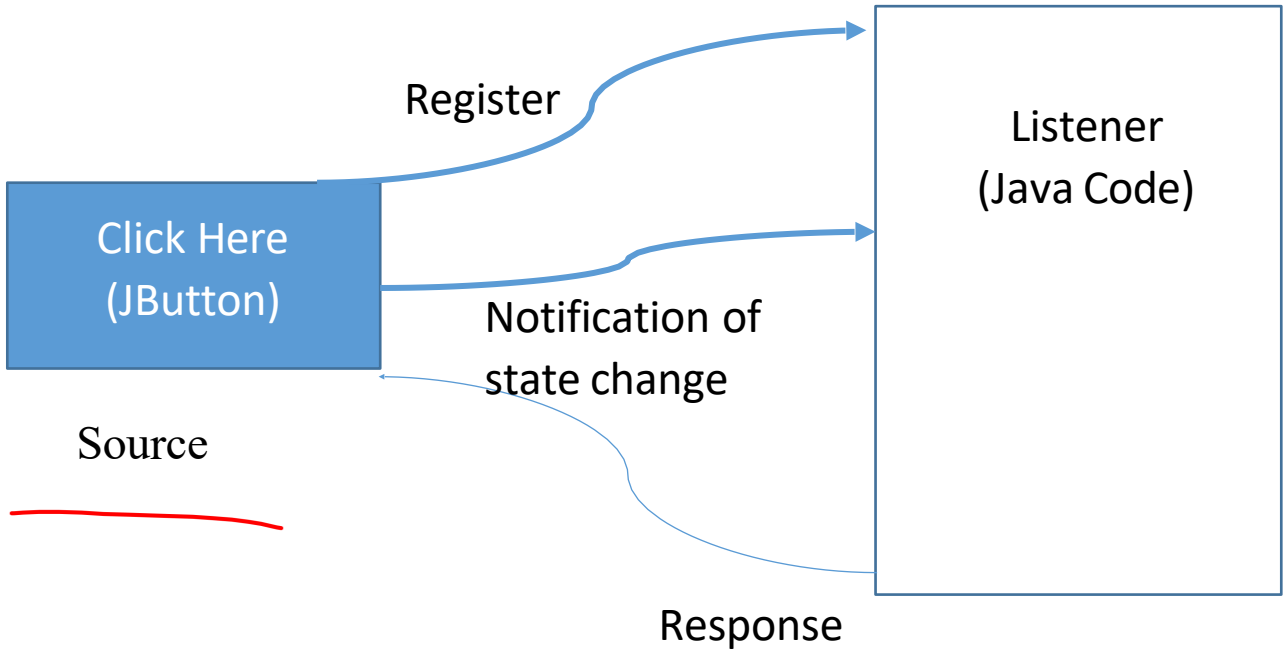
- **Event**: Change in the state of an object or source. For example, click on button.
- Event Handling is a mechanism that controls/ handles the event and decide what should happen when an event occurs.

- If the button is clicked the state of the button is changed from unclicked to clicked. So, if we want to perform some action (eg, I want to print “Button is clicked”) on clicking we will use event handling mechanism.

Java Event Handling

- **Delegation Event Model** define standard mechanisms to generate and process events. It contain two different entities
  - **Source:** An object on which an event occurs and it provides information of the occurred event to one or more Listeners. Eg. Jbutton
  - **Listener (Event Handler):** code responsible for generating response to the event.
    - Simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- Every source has to be registered with the listener. So, when the state of the source changes, notification is sent to Listener. Listener code will be executed in response to occurrence of event.
- **Events:** In the delegation model, an *event* is an object that describes a state change in a source. Among other causes, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface. Eg.: pressing a button

Event Handling: Delegation Event Model



Benefits of Delegation Event Model

- Application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to “delegate” the processing of an event to a separate piece of code.

Java Event Handling

- The **java.awt.event** package provides many event classes and Listener interfaces for event handling.
- **Following steps are required to perform event handling:**
  1. Register the source (component) with the Listener.
  2. Implement the listener methods to receive and process notifications

Registration Methods

- For registering the component with the Listener, many classes provide the registration methods. For example:
- **Button**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
  - public void addActionListener(ActionListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Different Sources of Events

Event Source	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Classes	Listener Interfaces
ActionEvent	ActionListener

Example of Event Handling for JButton

- For registering the **JButton** component with the Listener, For example:
  - public void addActionListener(ActionListener a){}
- Implement the listener method to receive and process notifications.
- ActionListener: The listener interface for receiving action events.
- The Java ActionListener is notified whenever you click on the button. It is notified against ActionEvent.
- The ActionListener interface is found in java.awt.event package. It has only one method: actionPerformed().
- You have to implement actionPerformed() and write the code that you want to do when button is clicked.

Swing Buttons

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

JButton Class

- The JButton class is used to create a labeled button.
- An object of class JButton is a push button that the user can click to trigger some action.
- The application result in some action when the button is pushed.
- **Constructors:**

Constructor	Description
JButton()	It creates a button with no text and icon.
JButton(String s)	It creates a button with the specified text.

Methods of JButton

Methods	Description
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object. (Registering the component with the listener)

Refer java program: All of the previously referred java programs have created a Jbutton and Ex4_JButtonList.java

JButton Class

- Considering an example that we have created
  - JButton b1 = new JButton("Go");
- **Events:** When the user clicks on a button, the button generates an event of type Action-Event. This event is sent to any listener that has been registered with the button as an ActionListener.
- **Listeners:** An object that wants to handle events generated by buttons must implement the ActionListener interface. This interface defines just one method, "**public void actionPerformed(ActionEvent evt)",** which is called to notify the object of an action event.
- **Registration of Listeners:** In order to actually receive notification of an event from a button, an ActionListener must be registered with the button. This is done with the button's **addActionListener()** method. For example: stopGoButton.addActionListener(buttonHandler);

JButton Class

- **Event methods:** When actionPerformed(evt) is called by the button, the parameter, evt, contains information about the event. This information can be retrieved by calling methods in the ActionEvent class.
  - **evt.getActionCommand()** *returns a String giving the command associated with the button.*
  - The method **evt.getSource()** *returns a reference to the Object that produced the event,* that is, to the JButton that was pressed. The return value is of type Object, not JButton, because other types of components can also produce ActionEvents.

JLabel

- The object of JLabel class is a component for placing text in a container.
- It is used to display a single line of read only text.
- It is a passive component in that it does not respond to user input.
- The text can be changed by an application but a user cannot edit it directly.
- It inherits JComponent class.

Commonly used Constructors of JLabel

Constructor	Description
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.

Commonly used Methods of JLabel

Methods	Description
String getText()	It returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.

Refer program: Ex5_LabelEx1.java

JTextField

- The object of a JTextField class is a text component that allows the editing of a single line text. It has two uses:
  - **Input**. The user can enter one line of text (a String)
  - **Output**. To display one line of text.
- **JTextField** generates **ActionEvent** when user presses Enter.

JTextField

Constructor	Description
JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.
JTextField(String text, int columns)	Creates a new TextField initialized with the specified text and columns.
JTextField(int columns)	Creates a new empty TextField with the specified number of columns.

Commonly used Methods of JTextField

Methods	Description
void addActionListener(ActionListener l)	It is used to add the specified action listener to receive action events from this textfield.
Void setText(String text)	It display the text string.
String getText()	It retrieves the single line of text that the user enters.

JTextField

- **Event:**
- When the user is typing in a JTextField and presses enter key, an ActionEvent is generated.
- If you want to respond to such events, you can register an ActionListener with the text field, using the text field's addActionListener() method.

Refer Ex6_SimpleCalculator.java

Radio Buttons

- The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.
- Radio buttons are used in groups (use ButtonGroup class) where at most one can be selected.
-

Constructor	Description
JRadioButton()	Creates an unselected radio button with no text.
JRadioButton(String s)	Creates an unselected radio button with specified text.
JRadioButton(String s, boolean selected)	Creates a radio button with the specified text and selected status.

Refer Programs: JRadioButtonEvent1.java

Commonly used Methods of JRadioButton Class

Methods	Description
void setText(String s)	It is used to set specified text on button.
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Refer Programs: JRadioButtonEvent1.java

JCheckBox

- The JCheckBox class is used to create a checkbox.
- It is used to turn an option on (true) or off (false). Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".
- Constructors:

Constructor	Description
JCheckBox()	Creates an initially unselected check box button with no text, no icon.
JCheckBox(String s)	Creates an initially unselected check box with text.
JCheckBox(String text, boolean selected)	Creates a check box with text and specifies whether or not it is initially selected.

JCheckBox

- A JCheckBox is a component that has two states:
  - selected or unselected.
- The user can change the state of a check box by clicking on it. The state of a checkbox is represented by a Boolean value that is true if the box is selected and false if the box is unselected.
- JCheckBox chk1 = new JCheckBox("C++");

JCheckBox

- When the user selects or deselects a check box, an **ItemEvent** is generated.
- You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem( )** on the **ItemEvent** passed to the **itemStateChanged( )** method defined by **ItemListener**.
- The easiest way to determine the selected state of a check box is to call **isSelected( )** on the **JCheckBox** instance.
- **public int getStateChange():** Returns the type of state change (selected or deselected).
- Returns an integer that indicates whether the item was selected or deselected

Refer program: Ex14_JCheckBoxEvent.java

JComboBox class

- ComboBox is a **Swing component** used to create a **drop-down list** that allows the user to **select one item at a time**.
- JComboBox is a part of Java Swing package.
- JComboBox inherits JComponent class .
- JComboBox can be editable or read- only depending on the choice of the programmer .

Constructor of the JComboBox are:

- **JComboBox()** : creates a new empty JComboBox
- **JComboBox(Object[] items)** Creates combo box with given array

Commonly used Methods of JComboBox are:

- **addItem(E item)** : adds the item to the JComboBox
- **getItemAt(int i)** : returns the item at index i
- **getItemCount()**: returns the number of items from the list
- **getSelectedItem()** : returns the item which is selected
- **removeItemAt(int i)** : removes the element at index I
- **setSelectedIndex(int i)**: selects the element of JComboBox at index i.
- **setEnabled(boolean b)**: enables the combo box so that items can be selected.
- **removeItem(Object anObject)** : removes an item from the item list.
- **removeAllItems()**: removes all items from the item list.

Basic Layout Managers

- A layout manager is an object associated with a container that implements some policy for laying out the components in that container.
- Different types of layout manager implement different policies.
- Every container has an instance method, setLayout(), that takes a parameter of type LayoutManager and that is used to specify the layout manager that will be responsible for laying out any components that are added to the container.
- Components are added to a container by calling an instance method named add() in the container object.
- There are actually several versions of the add() method, with different parameter lists.
- Different versions of add() are appropriate for different layout managers, as we will see below.

Basic Layout Managers

- Java has a variety of standard layout managers that can be used as parameters in the setLayout() method.
- BorderLayout is the **default layout for Frame**
- They are defined by classes in the package java.awt.
- Here, we will look at just three of these layout manager classes:
  - FlowLayout,
  - BorderLayout
  - GridLayout.

Flow Layout

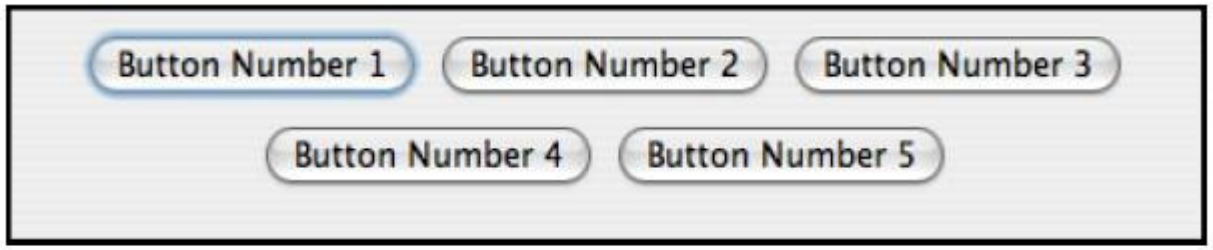
- A FlowLayout simply lines up components in a row across the container.
- The size of each component is equal to that component's "preferred size."
- After laying out as many items as will fit in a row across the container, the layout manager will move on to the next row.

Flow Layout

- The components in a given row can be either left-aligned, right-aligned, or centered within that row, and there can be horizontal and vertical gaps between components.
- If the default constructor, "new FlowLayout()", is used, then the components on each row will be centered and both the horizontal and the vertical gaps will be five pixels.
- The constructor **public FlowLayout(int align, int hgap, int vgap)** can be used to specify alternative alignment and gaps. The possible values of align are FlowLayout.LEFT, FlowLayout.RIGHT, and FlowLayout.CENTER.

Flow Layout

- For example, this picture shows five buttons in a panel that uses a FlowLayout:



- Note that since the five buttons will not fit in a single row across the panel, they are arranged in two rows. In each row, the buttons are grouped together and are centered in the row. The buttons were added to the panel using the statements:

```
panel.add(button1);
panel.add(button2);
panel.add(button3);
panel.add(button4);
panel.add(button5);
```

Grid Layout

- A grid layout lays out components in a grid of equal sized rectangles. This illustration shows how the components would be arranged in a grid layout with 3 rows and 2 columns:

#1	#2
#3	#4
#5	#6

- If a container uses a GridLayout, the appropriate add method for the container takes a single parameter of type Component (for example: cntr.add(comp)).
- Components are added to the grid in the order shown; that is, each row is filled from left to right before going on the next row.

Grid Layout

- The constructor for a GridLayout takes the form

"new GridLayout(R,C)"

- where R is the number of rows and C is the number of columns. If you want to leave horizontal gaps of H pixels between columns and vertical gaps of V pixels between rows, use "new GridLayout(R,C,H,V)" instead.

Refer Program discussed in Lecture

JOptionPane

- The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box. These dialog boxes are used to display information or get input from the user. The JOptionPane class inherits JComponent class.
- Constructors:

Constructor	Description
JOptionPane()	It is used to create a JOptionPane with a test message.
JOptionPane(Object message)	It is used to create an instance of JOptionPane to display a message.
JOptionPane(Object message, int messageType)	It is used to create an instance of JOptionPane to display a message with specified message type and default options.

Commonly Used JOptionPane Methods

Message Dialog:

- Used to display information, warning, error, or plain messages.
  JOptionPane.showMessageDialog(null,"Operation Successful");

- **With Message Type:**
  JOptionPane.showMessageDialog(null, "Invalid Login!","Error",
  JOptionPane.ERROR_MESSAGE);

Message Types

Constant	Purpose
INFORMATION_MESSAGE	Information
WARNING_MESSAGE	Warning
ERROR_MESSAGE	Error
QUESTION_MESSAGE	Question
PLAIN_MESSAGE	Simple message

Menu using Swing in Java

- Before you can create a menu, you need to know something about the three
- core menu classes: **JMenuBar**, **JMenu**, and **JMenuItem**.
- These form the minimum set of classes needed to construct a main menu for an application.

JMenuBar, JMenu, JMenuItems

- JMenuBar, JMenu and JMenuItems are a part of Java Swing package.
- **Core Menu Components:** To create a functional menu system in Java Swing, you need three main classes:
  - **JMenuBar**: A horizontal bar typically located at the top of a JFrame, which holds the top-level JMenu components.
  - **JMenu**: The actual menu header (e.g., "File", "Edit") that, when clicked, displays a list of options.
  - **JMenuItem**: The individual, clickable items within a JMenu that trigger specific actions when selected. Other subclasses like JCheckBoxMenuItem and JRadioButtonMenuItem can also be used.

JMenuBar, Jmenu, JMenuItems

- **Constructors :**
  - **JMenuBar() :** Creates a new MenuBar.
  - **JMenu() :** Creates a new Menu with no text.
  - **JMenu(String name) :** Creates a new Menu with a specified name.

- **Commonly used methods:**
  - **add(JMenu c) :** Adds menu to the menu bar. Adds JMenu object to the Menu bar.
  - **add(Component c) :** Add component to the end of JMenu
  - **add(Component c, int index) :** Add component to the specified index of JMenu
  - **add(JMenuItem menuItem) :** Adds menu item to the end of the menu.
  - **add(String s) :** Creates a menu item with specified string and appends it to the end of menu.
  - **getItem(int index) :** Returns the specified menuitem at the given index

  - Refer MenuDemo.java

Thanks!!!