

## UNIT – V

### SYLLABUS:

**AWT & Event Handling:** The AWT class hierarchy, user interface components - labels, buttons, canvas, scrollbars, text components, checkbox, checkbox groups, choices, lists.

Events, event sources, event classes, event listeners, delegation event model, handling mouse and key board events, adapter classes.

**Layout manager:** Border, Grid, Flow, Card and Grid Bag layouts.

**Swings:** Introduction, limitations of AWT, components, containers,

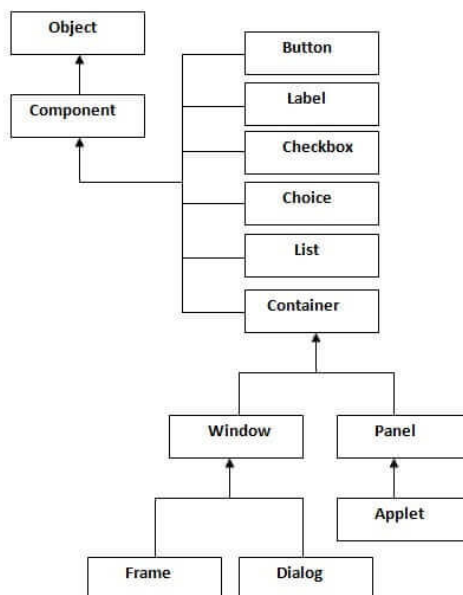
Exploring Swing Components - JApplet, JFrame and JComponent, Icons and Labels, Text fields, JButton class, Checkboxes, Radio buttons, ScrollPanes.

## Java Abstract Window Toolkit (AWT)

- ✓ Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.
- ✓ AWT contains large number of classes and methods that allows you to create and manage graphical user interface (GUI) applications, such as windows, buttons, scroll bars etc.
- ✓ The AWT was designed to provide a common set of tools for GUI design that could work on a variety of platforms. AWT is heavyweight i.e. its components are using the resources of OS.
- ✓ The tools provided by the AWT are implemented using each platform's native GUI toolkit, hence preserving the look and feel of each platform. This is an advantage of using AWT.
- ✓ But the disadvantage of such an approach is that GUI designed on one platform may look different when displayed on another platform. Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- ✓ AWT is the foundation upon which Swing is made i.e., Swing is a set of GUI interfaces that extends the AWT. But now a days AWT is merely used because most GUI Java programs are implemented using Swing because of its rich implementation of GUI controls and light-weighted nature.
- ✓ Java's newest GUI framework is JavaFX. It is anticipated that at some point in the near future, JavaFX will replace Swing as Java's most popular GUI.

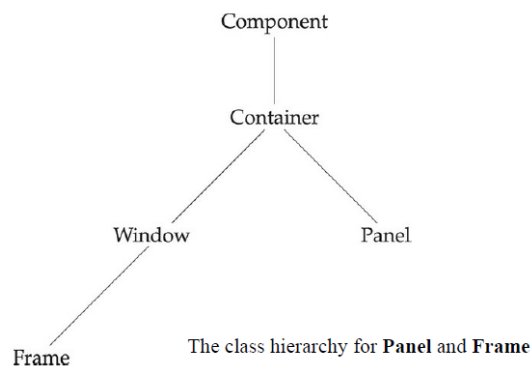
### Java AWT Class Hierarchy:

- ✓ The hierarchy of Java AWT classes are given below.
- ✓ The AWT classes are contained in the java.awt package.



## Window Fundamentals

- ✓ The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level.
- ✓ Two most important window-related classes are Frame and Panel.
- ✓ Frame encapsulates a top-level window and it is typically used to create what would be thought of as a standard application window.
- ✓ Panel provides a container to which other components can be added. Panel is also a superclass for Applet (deprecated as of JDK 9).
- ✓ Much of the functionality of Frame and Panel is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding.
- ✓ The following figure shows the class hierarchy for Panel and Frame.



### java.awt.Component class:

- ✓ Component class is at the top of AWT hierarchy.
- ✓ Component is an abstract class that encapsulates all the attributes of visual component.
- ✓ Except for menus, all user interface elements that are displayed on the screen and that interact with the user are subclasses of component.
- ✓ A component object is responsible for remembering the current foreground and background colors and the currently selected text font.

### java.awt.Container class:

- ✓ Container is a component in AWT that contains another component like button, text field, tables etc.
- ✓ Container is a subclass of component class.
- ✓ Container class keeps track of components that are added to another component.
- ✓ Container is responsible for laying out (that is, positioning) any components that it contains.

### java.awt.Panel class:

- ✓ Panel class is a concrete subclass of Container.
- ✓ Panel does not contain title bar, menu bar or border.
- ✓ Panel may be thought of as a recursively nestable, concrete screen component.
- ✓ Other components can be added to a Panel object by its add( ) method (inherited from Container).
- ✓ Once these components have been added, you can position and resize them manually using the setLocation( ), setSize( ), setPreferredSize( ), or setBounds( ) methods defined by Component.

### java.awt.Window class:

- ✓ The Window class creates a top-level window.
- ✓ A top-level window is not contained within any other object; it sits directly on the desktop.
- ✓ Generally, you won't create Window objects directly.
- ✓ Instead, you will use a subclass of Window called Frame, described next.

### java.awt.Frame class:

- ✓ Frame encapsulates a "window."
- ✓ It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.
- ✓ The precise look of a Frame will differ among environments.

### java.awt.Canvas class:

- ✓ Canvas is not part of the hierarchy for Panel or Frame.
- ✓ Canvas is derived from Component, and it encapsulates a blank window upon which you can draw.

## Working with Frame Windows

- ✓ Generally, AWT-based application window is derived from Frame.
- ✓ It creates a standard-style, top-level window that has all of the features normally associated with an application window, such as a close box and title.

Here are two of Frame's constructors:	
Frame() throws HeadlessException	It creates a standard window that does not contain a title.
Frame(String title) throws HeadlessException	It creates a window with the title specified by title.
A <b>HeadlessException</b> is thrown if an attempt is made to create a Frame instance in an environment that does not support user interaction.	

- ✓ There are several key methods you will use when working with Frame windows.

void setSize(int newWidth, int newHeight) void setSize(Dimension newSize)	This methods is used to set the dimensions of the window.
Dimension getSize( )	This method is used to obtain the current size of a window. Returns the current size of the window contained within the width and height fields of a Dimension object.
void setVisible(boolean visibleFlag)	After a frame window has been created, it will not be visible until you call setVisible(). The component is visible if the argument to this method is true. Otherwise, it is hidden.
void setTitle(String newTitle)	To change the title in a frame window. Here, newTitle is the new title for the window.
void paint(Graphics context)	The paint() method is called each time an AWT-based application's output must be redrawn.
void drawString(String message, int x, int y)	To output a string to a Frame. Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0, 0.
void setBackground(Color newColor)	To set the background color.

<code>void setForeground(Color newColor)</code>	To set the foreground color.
<code>Color getBackground()</code>	To obtain the current settings for the background colors.
<code>Color getForeground()</code>	To obtain the current settings for the foreground colors.
<code>void repaint()</code> <code>void repaint(int left, int top, int width, int height)</code> <code>void repaint(long maxDelay)</code> <code>void repaint(long maxDelay, int x, int y, int width, int height)</code>	To repaint the output on the frame window.

## Introducing Graphics

- ✓ The AWT includes several methods that support graphics.
- ✓ All graphics are drawn relative to a window.
- ✓ This can be the main window of an application or a child window. (These methods are also supported by Swing-based windows.)
- ✓ The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels.
- ✓ A graphics context is encapsulated by the Graphics class. Here are two ways in which a graphics context can be obtained:
  - I. It is passed to a method, such as `paint()` or `update()`, as an argument.
  - II. It is returned by the `getGraphics()` method of Component.

<code>void drawLine(int startX, int startY, int endX, int endY)</code>	Displays a line in the current drawing color that begins at startX, startY and ends at endX, endY.
<code>void drawRect(int left, int top, int width, int height)</code>	Display an outlined of a rectangle. The upper-left corner of the rectangle is at left, top. The dimensions of the rectangle are specified by width and height.
<code>void fillRect(int left, int top, int width, int height)</code>	Display a filled rectangle. The upper-left corner of the rectangle is at left, top. The dimensions of the rectangle are specified by width and height.
<code>void drawOval(int left, int top, int width, int height)</code>	To draw an ellipse. The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by left, top and whose width and height are specified by width and height.
<code>void fillOval(int left, int top, int width, int height)</code>	To fill an ellipse.
<code>void drawArc(int left, int top, int width, int height, int startAngle, int sweepAngle)</code> <code>void fillArc(int left, int top, int width, int height, int startAngle, int sweepAngle)</code>	To draw and fill the arcs.
<code>void drawPolygon(int x[ ], int y[ ], int numPoints)</code> <code>void fillPolygon(int x[ ], int y[ ], int numPoints)</code>	To draw and fill the polygons.

### //Demonstrating Graphics:

```
import java.awt.event.*;
import java.awt.*;

public class GraphicsDemo extends Frame
{
    public GraphicsDemo()
    {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
    public void paint(Graphics g)
    {
        //Draw Lines
        g.drawLine(20, 40, 100, 90);
        g.drawLine(20, 90, 100, 40);
        g.drawLine(40, 45, 250, 80);
        g.drawLine(75, 90, 400, 400);

        //Draw rectangle
        g.drawRect(20, 150, 60, 50);
        g.drawRoundRect(200, 150, 60, 50, 15, 15);

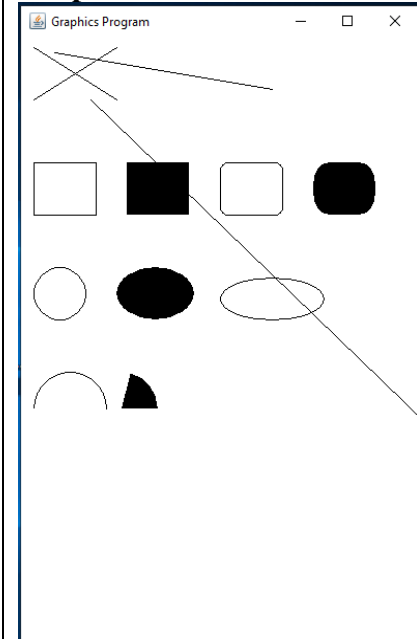
        //Draw Filled rectangle
        g.fillRect(110, 150, 60, 50);
        g.fillRoundRect(290, 150, 60, 50, 30, 40);

        //Draw eclipses and circles
        g.drawOval(20, 250, 50, 50);
        g.fillOval(100, 250, 75, 50);
        g.drawOval(200, 260, 100, 40);

        //Draw Arcs.
        g.drawArc(20, 350, 70, 70, 0, 180);
        g.fillArc(70, 350, 70, 70, 0, 75);
    }

    public static void main(String args[])
    {
        GraphicsDemo appwin=new GraphicsDemo();
        appwin.setSize(new Dimension(400, 700));
        appwin.setTitle("Graphics Program");
        appwin.setVisible(true);
    }
}
```

## Output:



## AWT Controls:

- ✓ Controls are components that allow a user to interact with your application in various ways.  
**For Example:** A commonly used control is the push button.
- ✓ A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.
- ✓ In addition to the controls, a frame window can also include a standard-style menu bar. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. This constitutes the main menu of an application. As a general rule, a menu bar is positioned at the top of a window. Menu bars are handled in much the same way as are the other controls.

### AWT Control Fundamentals

- ✓ The AWT supports the following types of controls:
  - a) Labels
  - b) Push buttons
  - c) Check boxes
  - d) Choice lists
  - e) Lists
  - f) Scroll bars
  - g) Text Editing
- ✓ All AWT controls are subclasses of Component.
- ✓ Although the set of controls provided by the AWT is not particularly rich, it is sufficient for simple applications, such as short utility programs intended for your own use.
- ✓ It is also quite useful for introducing the basic concepts and techniques related to handling events in controls.

## Adding and Removing Controls

- ✓ To include a control in a window, it must be added to the window. To do this, an instance of the desired control must be created first then it should be added to a window by calling `add()`, which is defined by **Container**.
- ✓ The `add()` method has several forms.

```
Component add(Component compRef)
```

- ✓ Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.
- ✓ To remove a control from a window when the control is no longer needed call `remove()` method. This method is also defined by **Container** class.
- ✓ Here is one of its forms:

```
void remove(Component compRef)
```

- ✓ Here, *compRef* is a reference to the control you want to remove. To remove all controls call `removeAll()` method.

## Responding to Controls

- ✓ Except for labels, which are passive, all other controls generate events when they are accessed by the user.  
**For Example:** When the user clicks on a push button, an event is sent that identifies the push button.
- ✓ In general, the program simply implements the appropriate interface and then registers an event listener for each control that needs to be monitored. Once a listener has been installed, events are automatically sent to it.

## The HeadlessException

- ✓ Most of the AWT controls have constructors that throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present).
- ✓ This exception can be used to write code that can adapt to non-interactive environments. (Of course, this is not always possible.)

## java.awt.Label class:

- ✓ The easiest control to use is a label.
- ✓ A label is an object of type **Label**, and it contains a string, which it displays.
- ✓ Labels are passive controls that do not support any interaction with the user.
- ✓ Label class declaration:

```
public class Label extends Component implements Accessible
```

- ✓ Label defines the following constructors:

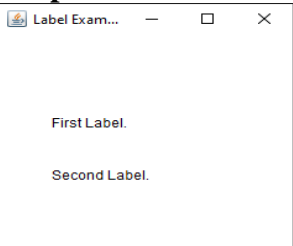
## Constructors:

<code>Label()</code> throws <b>HeadlessException</b>	It creates a blank label.
<code>Label(String str)</code> throws <b>HeadlessException</b>	It creates a label that contains the string specified by <i>str</i> . This string is left-justified.
<code>Label(String str, int how)</code> throws <b>HeadlessException</b>	It creates a label that contains the string specified by <i>str</i> using the alignment specified by <i>how</i> .

	The value of <i>how</i> must be one of these three constants: <b>Label.LEFT</b> , <b>Label.RIGHT</b> , or <b>Label.CENTER</b> .
--	---

## Methods

void setText(String str)	To set or change the text in a label. <i>str</i> specified the new text for the label.
String getText()	To obtain the current text in a label. A String object containing the text of the label will be returned.
void setAlignment(int <i>how</i> )	To set the alignment of the string within the label.
int getAlignment()	To obtain the current alignment.
setBounds(int x-coordinate, int y-coordinate, int width, int height)	The first two arguments are <b>x and y coordinates</b> of the <b>top-left corner</b> of the component, the third argument is the <b>width</b> of the component and the fourth argument is the <b>height</b> of the component. In the absence of a layout manager, the position and size of the components have to be set manually. The <b>setBounds()</b> method is used in such a situation to set the position and size.

<b>//Demonstrate Labels without a Layout Manger.</b>	
<pre>import java.awt.*; class LabelExample {     public static void main(String args[])     {         Frame f= new Frame("Label Example");         Label l1,l2;         l1=new Label("First Label.");         l1.setBounds(50,100, 100,30);         l2=new Label("Second Label.");         l2.setBounds(50,150, 100,30);         f.add(l1);         f.add(l2);         f.setSize(400,400);         f.setLayout(null);         f.setVisible(true);     } }</pre>	
<b>Output:</b> 	



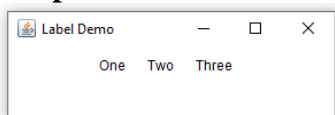
### //Demonstrate Labels.

```
import java.awt.*;
import java.awt.event.*;
class LabelDemo extends Frame
{
    public LabelDemo()
    {
        //Use a flow layout.
        setLayout(new FlowLayout());
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");

        //Add labels to frame.
        add(one);
        add(two);
        add(three);

        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
    public static void main(String[] args)
    {
        LabelDemo appwin = new LabelDemo();
        appwin.setSize(new Dimension(300, 100));
        appwin.setTitle("Label Demo");
        appwin.setVisible(true);
    }
}
```

### Output:



### java.awt.Button class:

- ✓ A push button is a component that contains a label and generates an event when it is pressed. They are the most widely used controls.
- ✓ Push buttons are objects of type Button class.
- ✓ Button class declaration:

```
public class Button extends Component implements Accessible
```

- ✓ Button class defines two constructors:

## Constructors:

Button() throws HeadlessException	It creates an empty button.
Button(String str) throws HeadlessException	It creates a button that contains <i>str</i> as a label.

## Methods:

void setLabel(String str)	To set the Button's label. Here, <i>str</i> becomes the new label for the button.
String getLabel()	To retrieve the label of a push button. An object of String will be returned containing the label.

## Handling Buttons

- ✓ Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component.
- ✓ Each listener implements the **ActionListener** interface. That interface defines the *actionPerformed()* method, which is called when an event occurs.
- ✓ An **ActionEvent** object is supplied as the argument to *actionPerformed()* method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button.
- ✓ By default, the *action command string* is the label of the button. Either the button reference or the *action command string* can be used to identify the button.
- ✓ The label(*action command string*) is obtained by calling the *getActionCommand()* method on the **ActionEvent** object passed to *actionPerformed()* method.

String getActionCommand();	Returns the command string associated with this action.
----------------------------	---

- ✓ In addition to comparing button *action command strings*, we can also determine which button has been pressed by comparing the object obtained from the *getSource()* method to the button objects that has been added to the window.

Object getSource();	Returns the object on which the Event initially occurred.
---------------------	---

### //Demonstrate Buttons.

// An example that creates three buttons labeled "Yes", "No", and "Undecided". Each time one is pressed, a message is displayed that reports which button has been pressed.

```
import java.awt.*;
import java.awt.event.*;
public class ButtonDemo extends Frame implements ActionListener
{
    String msg = "";
    Button yes, no, maybe;
    public ButtonDemo()
    {
        setLayout(new FlowLayout());
        yes = new Button("Yes");
        no = new Button("No");
        maybe = new Button("Undecided");

        add(yes);
```

```

        add(no);
        add(maybe);
        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

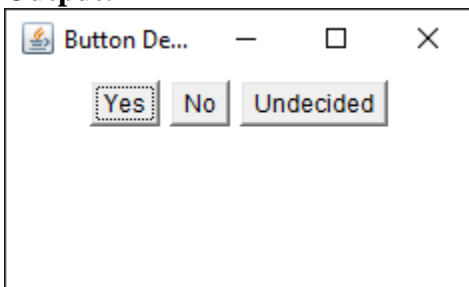
    public void actionPerformed(ActionEvent ae)
    {
        String str = ae.getActionCommand();
        if(str.equals("Yes"))
        {
            msg = "You pressed Yes.";
        }
        else if(str.equals("No"))
        {
            msg = "You pressed No.";
        }
        else
        {
            msg = "You pressed Undefined.";
        }
        repaint();
    }

    public void paint(Graphics g)
    {
        g.drawString(msg,20,100);
    }

    public static void main(String[] args)
    {
        ButtonDemo appwin = new ButtonDemo();
        appwin.setSize(new Dimension(250, 150));
        appwin.setTitle("Button Demo");
        appwin.setVisible(true);
    }
}

```

### Output:



## java.awt.Checkbox class:

- ✓ A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not.
- ✓ There is a label associated with each check box that describes what option the box represents. The state of a check box can be changed by clicking on it.
- ✓ Check boxes can be used individually or as part of a group.
- ✓ Check boxes are objects of the Checkbox class.
- ✓ Checkbox class declaration:

public class <b>Checkbox</b> extends <b>Component</b> implements <b>ItemSelectable</b> , <b>Accessible</b>
--

- ✓ **Checkbox** class supports the following constructors:

### Constructors:

Checkbox() throws HeadlessException	It creates a check box whose label is initially blank. The state of the check box is unchecked.
Checkbox(String str) throws HeadlessException	It creates a check box whose label is specified by <i>str</i> . The state of the check box is unchecked.
Checkbox(String str, boolean on) throws HeadlessException	It allows to set the initial state of the check box. If <i>on</i> is true, the check box is initially checked; otherwise, it is cleared.
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException	It create a check box whose label is specified by <i>str</i> and whose group is specified by <i>cbGroup</i> . If this check box is not part of a group, then <i>cbGroup</i> must be <i>null</i> . The value of <i>on</i> determines the initial state of the check box.
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException	

### Methods:

boolean getState()	To retrieve the current state of a check box.
void setState(boolean on)	To set the state of a check box. If <i>on</i> is <i>true</i> , the box is checked. If it is <i>false</i> , the box is cleared.
String getLabel()	To obtain the current label associated with a check box.
void setLabel(String str)	To set the label to the check box. The string passed in <i>str</i> becomes the new label associated with the invoking check box.

### Handling Check Boxes

- ✓ Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component.
- ✓ Each listener implements the **ItemListener** interface. That interface defines the *itemStateChanged()* method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event.

#### //Demonstrate Checkbox.

// The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
import java.awt.*;
import java.awt.event.*;
public class CheckboxDemo extends Frame implements ItemListener
{
```

```

String msg = "";
Checkbox windows, android, solaris, mac;
public CheckboxDemo()
{
    setLayout(new FlowLayout());
    windows = new Checkbox("Windows", true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);

    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent we) {
            System.exit(0);
        }
    });
}

public void itemStateChanged(ItemEvent ie)
{
    repaint();
}

public void paint(Graphics g)
{
    msg = "Current State:";
    g.drawString(msg,20,120);
    msg = "Windows:"+windows.getState();
    g.drawString(msg,20,140);
    msg = "Android:"+android.getState();
    g.drawString(msg,20,160);
    msg = "Solaris:"+solaris.getState();
    g.drawString(msg,20,180);
    msg = "Mac OS:"+mac.getState();
    g.drawString(msg,20,200);
}

public static void main(String[] args)
{
    CheckboxDemo appwin = new CheckboxDemo();

    appwin.setSize(new Dimension(250, 220));
    appwin.setTitle("Checkbox Demo");
}

```

appwin.setVisible(true);
}
}

**Output:**

**java.awt.CheckboxGroup class:**

- ✓ It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- ✓ These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time.
- ✓ To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes.
- ✓ Check box groups are objects of type CheckboxGroup.
- ✓ Checkbox class declaration:

public class **CheckboxGroup** extends **Object** implements **Serializable**

- ✓ Only the default constructor is defined, which creates an empty group.

**Constructors:**

CheckboxGroup()	Creates a new instance of CheckboxGroup.
-----------------	--

**Methods:**

Checkbox getSelectedCheckbox( )	To determine which check box in a group is currently selected.
void setSelectedCheckbox(Checkbox which)	To set a check box. Here, <i>which</i> is the check box that you want to be selected. The previously selected check box will be turned off.

//Demonstrate CheckboxGroup Class.

import java.awt.\*;
import java.awt.event.\*;
public class CBGroupDemo extends Frame implements ItemListener
{
String msg = "";
Checkbox windows, android, solaris, mac;

```

CheckboxGroup cbg;
public CBGroupDemo()
{
    setLayout(new FlowLayout());
    cbg = new CheckboxGroup();
    windows = new Checkbox("Windows", cbg, true);
    android = new Checkbox("Android", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);

    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent we) {
            System.exit(0);
        }
    });
}

public void itemStateChanged(ItemEvent ie)
{
    repaint();
}

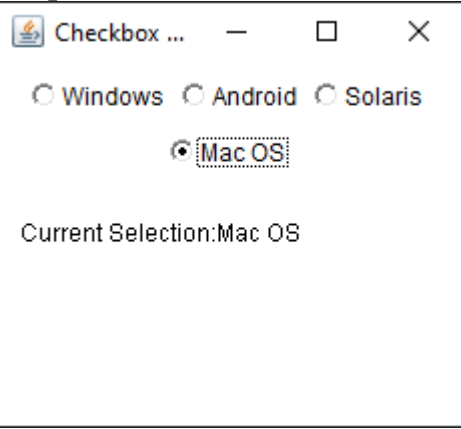
public void paint(Graphics g)
{
    msg = "Current Selection:";
    msg+= cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg,20,120);
}

public static void main(String[] args)
{
    CBGroupDemo appwin = new CBGroupDemo();

    appwin.setSize(new Dimension(250, 220));
    appwin.setTitle("Checkbox Demo");
    appwin.setVisible(true);
}
}

```

## Output:



### java.awt.Choice class:

- ✓ The Choice class is used to create a pop-up list of items from which the user may choose.
- ✓ A Choice control is a form of menu.
- ✓ When inactive, a Choice component takes up only enough space to show the currently selected item. But when the user clicks on it, the whole list of choices pops up, and a new selection can be made.
- ✓ Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object.
- ✓ Choice class declaration:

```
public class Choice extends Component implements ItemSelectable, Accessible
```

- ✓ Choice defines only the default constructor, which creates an empty list.

### Constructor:

Choice()	Creates a new choice menu.
----------	----------------------------

### Methods:

void add(String name)	To add a selection to the list. Here, <i>name</i> is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur.
String getSelectedItem()	To determine which item is currently selected. Returns a string containing the name of the item
int getSelectedIndex()	To determine which item is currently selected. Returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.
int getItemCount()	To obtain the number of items in the list.
void select(int index)	To set the currently selected item using with a zero-based integer index.
void select(String name)	To set the currently selected item with a string that will match a <i>name</i> in the list.
String getItem(int index)	Given an index, to obtain the name associated with the item at that index. Here, <i>index</i> specifies the index of the desired item.

### Handling Choice Lists

- ✓ Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component.



- ✓ Each listener implements the **ItemListener** interface. That interface defines the *itemStateChanged()* method. An **ItemEvent** object is supplied as the argument to this method.

#### //Demonstration of Choice Class

// Here is an example that creates two **Choice** menus. One selects the operating system. The other selects the browser.

```
import java.awt.*;
import java.awt.event.*;
public class ChoiceDemo extends Frame implements ItemListener
{
    String msg = "";
    Choice os, browser;
    public ChoiceDemo()
    {
        setLayout(new FlowLayout());
        os = new Choice();
        browser = new Choice();

        os.add("Windows");
        os.add("Android");
        os.add("Solaris");
        os.add("Mac OS");

        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Chrome");

        add(os);
        add(browser);

        os.addItemListener(this);
        browser.addItemListener(this);

        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public void itemStateChanged(ItemEvent ie)
    {
        repaint();
    }

    public void paint(Graphics g)
    {
        msg = "Current OS:";
        msg += os.getSelectedItem();
        g.drawString(msg, 20, 120);
        msg = "Current Browser:";
    }
}
```

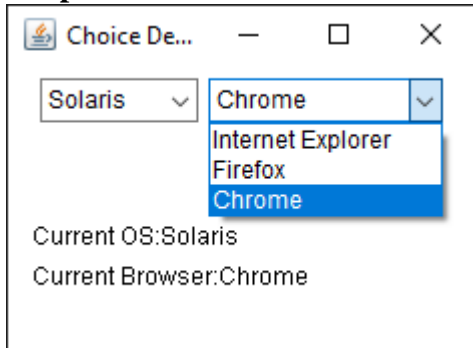
```

        msg += browser.getSelectedItem();
        g.drawString(msg, 20, 140);
    }
    public static void main(String[] args)
    {
        ChoiceDemo appwin = new ChoiceDemo();

        appwin.setSize(new Dimension(250, 180));
        appwin.setTitle("Choice Demo");
        appwin.setVisible(true);
    }
}

```

#### Output:



#### java.awt.Lists class:

- ✓ The List class provides a compact, multiple-choice, scrolling selection list.
- ✓ Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible window.
- ✓ It can also be created to allow multiple selections.
- ✓ List class declaration:

public class <b>List</b> extends <b>Component</b> implements <b>ItemSelectable</b> , <b>Accessible</b>
--

- ✓ List provides these constructors:

#### Constructors:

List() throws HeadlessException	To create a List control that allows only one item to be selected at any one time.
List(int numRows) throws HeadlessException	In this, the value of <i>numRows</i> specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed).
List(int numRows, boolean multipleSelect) throws HeadlessException	In this, if <i>multipleSelect</i> is true, then the user may select two or more items at a time. If it is false, then only one item may be selected.

#### Methods:

void add(String name)	It adds an item at the end of the list. Here, <i>name</i> is the name of the item added to the list.
-----------------------	--

void add(String name, int index)	It adds the item at the index specified by <i>index</i> , specified by <i>name</i> . Indexing begins at zero. Specify -1 to add the item to the end of the list.
String getSelectedItem()	To determine which item is currently selected, In lists that allow only single selection. It returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, <i>null</i> is returned.
int getSelectedIndex()	To determine which item is currently selected, In lists that allow only single selection. It returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, -1 is returned.
String[ ] getSelectedItems()	To determine the current selections, in lists that allow multiple selection. It returns an array containing the names of the currently selected items.
int[ ] getSelectedIndexes()	To determine the current selections, in lists that allow multiple selection. It returns an array containing the indexes of the currently selected items.
int getItemCount()	To obtain the number of items in the list.
void select(int index)	To set the currently selected item, with a zero-based integer index.
String getItem(int <i>index</i> )	Given an index, you can obtain the name associated with the item at that index. Here, <i>index</i> specifies the index of the desired item.

### Handling Lists:

- ✓ To process list events, it needs to implement the **ActionListener** interface.
- ✓ Each time a **List** item is double-clicked, an **ActionEvent** object is generated. Its *getActionCommand()* method can be used to retrieve the name of the newly selected item.
- ✓ Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated. Its *getStateChange()* method can be used to determine whether a selection or deselection triggered this event. *getItemSelectable()* returns a reference to the object that triggered this event.

#### //Demonstration of List class

// Here is an example that converts the Choice controls in the preceding section into List components, one multiple choice and the other single choice.

```
import java.awt.*;
import java.awt.event.*;
public class ListDemo extends Frame implements ActionListener
{
    String msg = "";
    List os, browser;
    public ListDemo()
    {
        setLayout(new FlowLayout());

        os = new List(4, true);
        browser = new List(4);

        //Add items to the OS list.
        os.add("Windows");
        os.add("Android");
```

```

        os.add("Solaris");
        os.add("Mac OS");

        //Add items to the browser list.
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Chrome");

        //Make initial selections
        os.select(0);
        browser.select(1);

        //add lists to the frame.
        add(os);
        add(browser);

        //Add action listeners.
        os.addActionListener(this);
        browser.addActionListener(this);

        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public void actionPerformed(ActionEvent ae)
    {
        repaint();
    }

    public void paint(Graphics g)
    {
        int idx[];

        msg = "Current OS:";
        idx = os.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
            msg += os.getItem(idx[i]) + " ";
        g.drawString(msg, 12, 120);

        msg = "Current Browser:";
        msg += browser.getSelectedItem();
        g.drawString(msg, 12, 140);
    }

    public static void main(String[] args)
    {
        ListDemo appwin = new ListDemo();

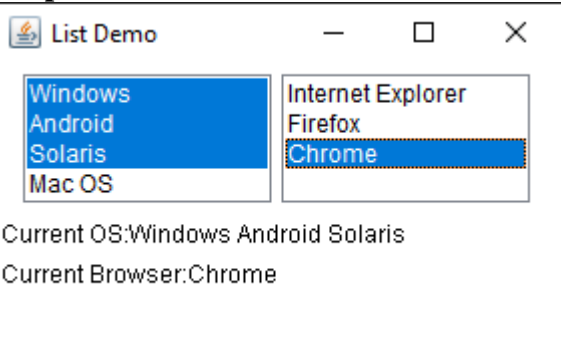
```

```

        appwin.setSize(new Dimension(300, 180));
        appwin.setTitle("List Demo");
        appwin.setVisible(true);
    }
}

```

**Output:**



### java.awt.TextField class:

- ✓ The TextField class implements a single-line text-entry area, usually called an edit control.
- ✓ Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- ✓ TextField is a subclass of TextComponent.
- ✓ TextField class declaration:

```
public class TextField extends TextComponent
```

- ✓ TextField defines the following constructors:

#### Constructors:

TextField() throws HeadlessException	It creates a default text field.
TextField(int numChars) throws HeadlessException	It creates a text field that is <i>numChars</i> characters wide.
TextField(String str) throws HeadlessException	The third form initializes the text field with the string contained in <i>str</i> .
TextField(String str, int numChars) throws HeadlessException	The fourth form initializes a text field and sets its width.

#### Methods:

String getText()	To obtain the string currently contained in the text field.
void setText(String <i>str</i> )	To set the text for the test field. Here, <i>str</i> is the new string.
String getSelectedText()	To select a portion of the text in a text field. It returns the selected text
void select(int startIndex, int endIndex)	To select a portion of text under program control. To selects the characters beginning at <i>startIndex</i> and ending at <i>endIndex</i> -1.
boolean isEditable()	To determine editability. It returns true if the text may be changed and false if not.
void setEditable(boolean <i>canEdit</i> )	To control whether the contents of a text field may be modified by the user. Here, if <i>canEdit</i> is <b>true</b> , the text may be changed. If it is <b>false</b> , the text cannot be altered.

void setEchoChar(char <i>ch</i> )	To disable the echoing of the characters as they are typed (Such as passwords). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). Here, <i>ch</i> specifies the character to be echoed. If <i>ch</i> is zero, then normal echoing is restored.
boolean echoCharIsSet()	To check a text field to see if it is in echo mode or not.
char getEchoChar()	To retrieve the echo character.

## Handling a TextField

- ✓ Since text fields perform their own editing functions, the program generally will not respond to individual key events that occur within a text field.
- ✓ However, it may want to respond when the user presses enter, when this occurs, an action event is generated.

### //Demonstrating TextField.

//Here is an example that creates the classic user name and password screen.

```
import java.awt.*;
import java.awt.event.*;
public class TextFieldDemo extends Frame implements ActionListener
{
    String msg = "";
    TextField name, pass;
    public TextFieldDemo()
    {
        setLayout(new FlowLayout());

        Label nameLabel = new Label("Name: ", Label.RIGHT);
        Label passLabel = new Label("Password: ", Label.RIGHT);

        name = new TextField(12);
        pass = new TextField(8);

        pass.setEchoChar('?');

        //Add the controls to the Frame.
        add(nameLabel);
        add(name);
        add(passLabel);
        add(pass);

        //Add action event handlers.
        name.addActionListener(this);
        pass.addActionListener(this);

        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }
}
```

```

    }

    //User pressed Enter.
    public void actionPerformed(ActionEvent ae)
    {
        repaint();
    }

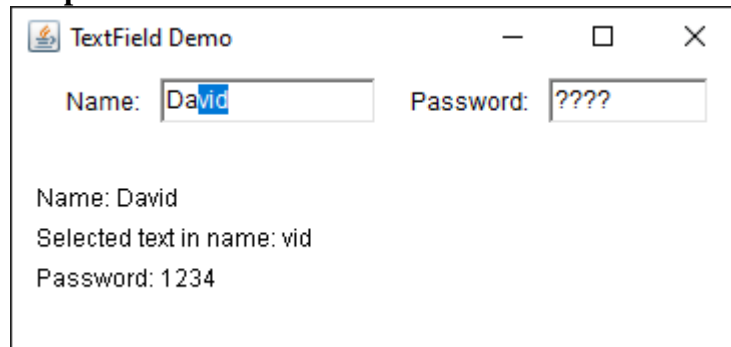
    public void paint(Graphics g)
    {
        g.drawString("Name: " + name.getText(), 20, 100);
        g.drawString("Selected text in name: " + name.getSelectedText(), 20, 120);
        g.drawString("Password: " + pass.getText(), 20, 140);
    }

    public static void main(String[] args)
    {
        TextFieldDemo appwin = new TextFieldDemo();

        appwin.setSize(new Dimension(380, 180));
        appwin.setTitle("TextField Demo");
        appwin.setVisible(true);
    }
}

```

#### Output:



#### java.awt.TextArea class:

- ✓ Sometimes a single line of text input is not enough for a given task.
- ✓ To handle these situations, the AWT includes a simple multiline editor called **TextArea**.
- ✓ **TextArea** class declaration:

```
public class TextArea extends TextComponent
```

- ✓ Following are the constructors for **TextArea**:

#### Constructors:

TextArea( ) throws HeadlessException	Here, <i>numLines</i> specifies the height, in lines, of the text area, and <i>numChars</i> specifies its width, in characters. Initial text can be specified by <i>str</i> . The
TextArea(int numLines, int numChars) throws HeadlessException	
TextArea(String str) throws HeadlessException	

TextArea(String str, int numLines, int numChars) throws HeadlessException	fifth form is used to specify the scroll bars. <i>sBars</i> must be one of these values:  SCROLLBARS_BOTH SCROLLBARS_NONE SCROLLBARS_HORIZONTAL_ONLY SCROLLBARS_VERTICAL_ONLY
TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException	

## Methods:

<ul style="list-style-type: none"> <li>✓ <b>TextArea</b> is a subclass of <b>TextComponent</b>.</li> <li>✓ Therefore, it supports the <b>getText( )</b>, <b>setText( )</b>, <b>getSelectedText( )</b>, <b>select( )</b>, <b>isEditable( )</b>, and <b>setEditable( )</b> methods described in the preceding section.</li> </ul>	
void append(String <i>str</i> )	Appends the string specified by <i>str</i> to the end of the current text.
void insert(String <i>str</i> , int <i>index</i> )	Inserts the string passed in <i>str</i> at the specified index
void replaceRange(String <i>str</i> , int <i>startIndex</i> , int <i>endIndex</i> )	Replaces the characters from <i>startIndex</i> to <i>endIndex</i> –1, with the replacement text passed in <i>str</i> .

## Handling a TextField

- ✓ Text areas are self-contained controls.
- ✓ The program incurs virtually no management overhead.
- ✓ Normally, the program simply obtains the current text when it is needed. However, it may listen for **TextEvents**, if chosen to do.

### //Demonstration of TextArea class.

// The following program creates a **TextArea** control.

```
import java.awt.*;
import java.awt.event.*;
public class TextAreaDemo extends Frame
{
    public TextAreaDemo()
    {
        setLayout(new FlowLayout());

        String val = "Welcome to java Programming\n" +
            "Widly used programming language for internet programming\n" +
            "This program is to demonstate TextArea.";

        TextArea text = new TextArea(val, 10, 30);
        add(text);

        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public static void main(String[] args)
```



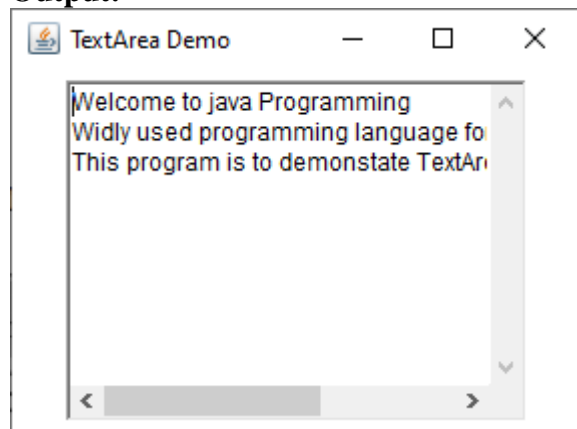
```

{
    TextAreaDemo appwin = new TextAreaDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("TextArea Demo");
    appwin.setVisible(true);
}
}

```

#### Output:



#### java.awt.ScrollBar class:

- ✓ Scroll bars are used to select continuous values between a specified minimum and maximum.
- ✓ Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.
- ✓ The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value.
- ✓ In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down.
- ✓ Scroll bars are encapsulated by the Scrollbar class.
- ✓ Scrollbar class declaration:

```
public class Scrollbar extends Component implements Adjustable, Accessible
```

#### Constructors:

Scrollbar() throws HeadlessException	It creates a vertical scroll bar.
Scrollbar(int style) throws HeadlessException	It allows to specify the orientation of the scroll bar. If <i>style</i> is Scrollbar.VERTICAL, a vertical scroll bar is created. If <i>style</i> is Scrollbar.HORIZONTAL, the scroll bar is horizontal.
Scrollbar(int style, int initialValue, int thumbSize, int min, int max) throws HeadlessException	It allows to specify the orientation like the above one and the initial value of the scroll bar is passed in <i>initialValue</i> . The number of units represented by the height of the thumb is passed in <i>thumbSize</i> . The

	minimum and maximum values for the scroll bar are specified by <i>min</i> and <i>max</i> .
--	--

### Methods:

void setValues(int initialValue, int thumbSize, int min, int max)	To set different parameters of a scroll bars.
int getValue()	To obtain the current value of the scroll bar. it returns the current setting.
void setValue(int <i>newValue</i> )	To set the current value. Here, <i>newValue</i> specifies the new value for the scroll bar.
int getMinimum()	To retrieve the minimum value of a scroll bar.
int getMaximum()	To retrieve the maximum value of a scroll bar.
void setUnitIncrement(int <i>newIncr</i> )	By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. This method is called to change this increment value.
void setBlockIncrement(int <i>newIncr</i> )	By default, page-up and page-down increments are 10. This method is called to change this value.
void setPreferredSize(Dimension <i>dim</i> )	To set the size of the scrollbars.

### Handling Scroll Bars

- ✓ To process scroll bar events, implement the **AdjustmentListener** interface.
- ✓ Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType()** method can be used to determine the type of the adjustment.
- ✓ The types of adjustment events are as follows:

BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

#### //Demonstration of Scrollbar class.

// The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed.

```
import java.awt.*;
import java.awt.event.*;
public class SBDemo extends Frame implements AdjustmentListener
{
    String msg = "";
    Scrollbar vertSB, horzSB;
    public SBDemo()
    {
        setLayout(new FlowLayout());

        vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, 200);
        vertSB.setPreferredSize(new Dimension(20, 100));
```

```

        horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 100);
        horzSB.setPreferredSize(new Dimension(100, 20));

        //add the scroll bars to the frame
        add(vertSB);
        add(horzSB);

        //add AdjustmentListeners to the scroll bars
        vertSB.addAdjustmentListener(this);
        horzSB.addAdjustmentListener(this);

        //add MouseMotionListener
        addMouseMotionListener(new MouseAdapter(){
            public void mouseDragged(MouseEvent me)
            {
                int x = me.getX();
                int y = me.getY();
                vertSB.setValue(y);
                horzSB.setValue(x);
                repaint();
            }
        });

        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public void adjustmentValueChanged(AdjustmentEvent ae)
    {
        repaint();
    }

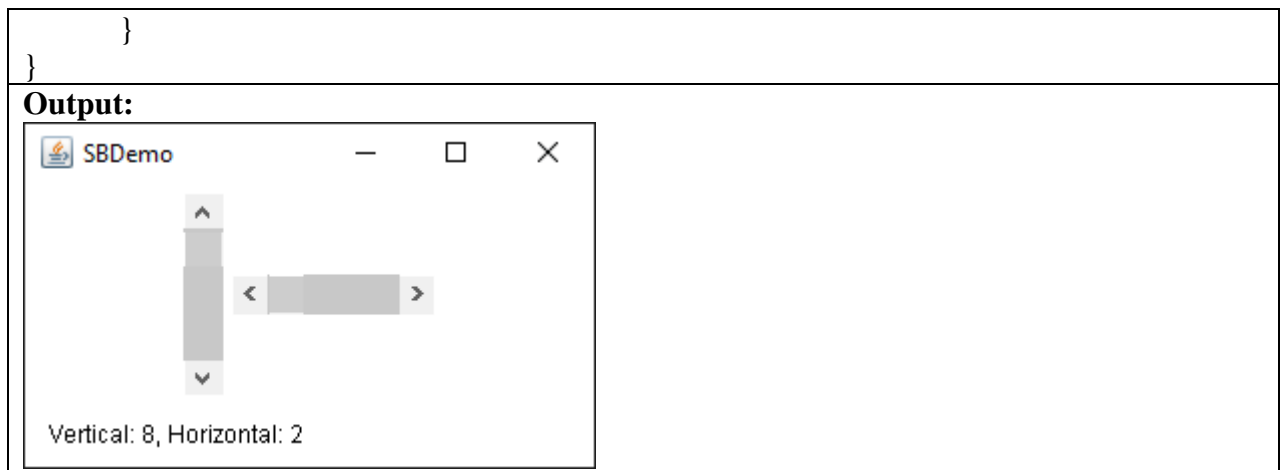
    public void paint(Graphics g)
    {
        msg = "Vertical: " + vertSB.getValue();
        msg += ", Horizontal: " + horzSB.getValue();
        g.drawString(msg, 20, 160);

        g.drawString("*", horzSB.getValue(), vertSB.getValue());
    }

    public static void main(String[] args)
    {
        SBDemo appwin = new SBDemo();

        appwin.setSize(new Dimension(300, 180));
        appwin.setTitle("SBDemo");
        appwin.setVisible(true);
    }

```



### java.awt.Canvas class:

- ✓ The Canvas control represents a blank rectangular area where the application can draw or trap input events from the user.
- ✓ It inherits the Component class.
- ✓ Canvas class declaration:

public class <b>Canvas</b> extends <b>Component</b> implements <b>Accessible</b>
--

### Constructors:

Canvas()	Constructs a new Canvas.
Canvas(GraphicsConfiguration config)	Constructs a new Canvas given a GraphicsConfiguration object.

### //Demonstrating Canvas Class

```

import java.awt.*;
public class CanvasExample
{
    public CanvasExample()
    {
        Frame f= new Frame("Canvas Example");
        f.add(new MyCanvas());
        f.setLayout(null);
        f.setSize(400, 400);
        f.setVisible(true);
    }

    public static void main(String args[])
    {
        new CanvasExample();
    }
}

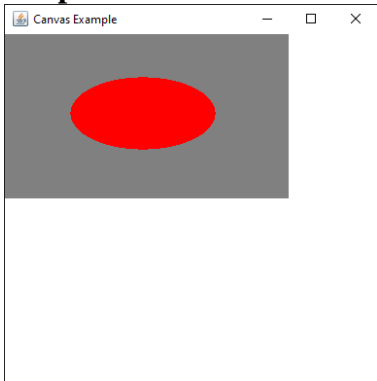
class MyCanvas extends Canvas
{
    public MyCanvas()
    {

```

```
        setBackground (Color.GRAY);
        setSize(300, 200);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        g.fillOval(75, 75, 150, 75);
    }
}
```

**Output:**



## Event Handling in Java:

- ✓ Event handling is fundamental to Java programming because it is integral to the creation of many kinds of applications.
- ✓ For example: Any program that uses a graphical user interface, such as a Java application written for Windows, is event driven.
- ✓ Thus, you cannot write these types of programs without a solid command of event handling.
- ✓ Events are supported by a number of packages, including **java.util**, **java.awt**, and **java.awt.event**.
- ✓ Beginning with JDK 9, **java.awt** and **java.awt.event** are part of the **java.desktop** module, and **java.util** is part of the **java.base** module.
- ✓ Most events to which the program will respond are generated when the user interacts with a GUI-based program.
- ✓ They are passed to your program in a variety of ways, with the specific method dependent upon the actual event.
- ✓ There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.

### Two Event Handling Mechanisms

- ✓ The way in which events are handled changed significantly between the original version of Java (1.0) and all subsequent versions of Java.
- ✓ The first mechanism is the 1.0 method of event handling is still supported, it is not recommended for new programs. Many of the methods that support the old 1.0 event model have been deprecated.
- ✓ The second mechanism is the modern approach. It is the way that events should be handled by all new programs and thus is the method employed by modern programs.

## The Delegation Event Model:

- ✓ The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events.
- ✓ Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- ✓ The advantage of this design is that the 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.
- ✓ 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.

## Advantage over Classical Approach:

- ✓ This is a more efficient way to handle events than the design used by the original Java 1.0 approach.
- ✓ Previously, an event was propagated up the containment hierarchy until it was handled by a component.
- ✓ This required components to receive events that they did not process, and it wasted valuable time.
- ✓ The delegation event model eliminates this overhead.

## EVENTS

- ✓ In the delegation model, an *event* is an object that describes a state change in a source.
- ✓ An event can be generated as a consequence of a person interacting with the elements in a graphical user interface.  
**For Example:** An event may be generated when the user interacts with the GUI like: pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
- ✓ Events may also occur that are not directly caused by interactions with a user interface.  
**For Example:** An event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

## EVENT SOURCES

- ✓ A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.
- ✓ A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- ✓ Each type of event has its own registration method. Here is the general form:

<pre>public void addTypeListener (TypeListener el)</pre>
Here, <i>Type</i> is the name of the event, and <i>el</i> is a reference to the event listener.

- ✓ **For example:**
  - I. The method that registers a keyboard event listener is called **addKeyListener()**.
  - II. The method that registers a mouse motion listener is called **addMouseMotionListener()**.
- ✓ When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.
- ✓ Some sources may allow only one listener to register. The general form of such a method is this:

<code>public void addTypeListener(TypeListener el)</code> throws <code>java.util.TooManyListenersException</code>
---

Here, <i>Type</i> is the name of the event, and <i>el</i> is a reference to the event listener.
---

- ✓ When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.
- ✓ A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

<code>public void removeTypeListener(TypeListener el)</code>
--

Here, <i>Type</i> is the name of the event, and <i>el</i> is a reference to the event listener.
---

**For example:** To remove a keyboard listener, you would call **removeKeyListener( )**.

## EVENT LISTENERS

- ✓ A *listener* is an object that is notified when an event occurs.
- ✓ It has two major requirements:
  - First, it must have been registered with one or more sources to receive notifications about specific types of events.
  - Second, it must implement methods to receive and process these notifications. In other words, the listener must supply the event handlers.
- ✓ The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event**.
- ✓ **For example:** The **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may handle one or both of these events if it provides an implementation of this interface.
- ✓ When a program receives an event, it must process it immediately, and then return. Failure to do this can cause the program to appear sluggish or even non-responsive.

## EVENT CLASSES

- ✓ The classes that represent events are at the core of Java's event handling mechanism.
- ✓ Java defines several types of events. The most widely used events are those defined by the AWT and those defined by Swing.
- ✓ **EventObject** is the root of the Java event class hierarchy, which is in **java.util** package.
- ✓ It is the superclass for all events.

### java.util.EventObject:

- ✓ **EventObject** is the root of the Java event class hierarchy, which is in **java.util** package.
- ✓ It is the superclass for all events.
- ✓ EventObject class declaration

<code>public class EventObject extends Object implements Serializable</code>
--

- ✓ Its only constructor is shown here:

### Constructors:

<code>EventObject(Object src)</code>
--------------------------------------

Here, <i>src</i> is the object that generates this event.
---

**Methods:**

<b>Object getSource()</b>	Returns the source of the event.
<b>String toString()</b>	Returns the string equivalent of the event.

**java.awt.AWTEvent class:**

- ✓ The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**.
- ✓ It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.
- ✓ AWTEvent class declaration:

```
public abstract class AWTEvent extends EventObject
```

**Constructors:**

AWTEvent(Event event)	Constructs an AWTEvent object from the parameters of a 1.0-style event.
AWTEvent(Object source, int id)	Constructs an AWTEvent object with the specified source object and type.

**Methods:**

int getID( )	Used to determine the type of the event.
--------------	--

**To summarize:**

- ☐ **EventObject** is a superclass of all events.
- ☐ **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.



- ✓ The package `java.awt.event` defines many types of events that are generated by various user interface elements.
- ✓ The following table shows several commonly used event classes and provides a brief description of when they are generated.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

### Commonly Used Event Classes in `java.awt.event`

#### `java.awt.ActionEvent` class:

- ✓ An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- ✓ The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT\_MASK**, **CTRL\_MASK**, **META\_MASK**, and **SHIFT\_MASK**. In addition, there is an integer constant, **ACTION\_PERFORMED**, which can be used to identify action events.
- ✓ **ActionEvent** class declaration:

```
public class ActionEvent extends AWTEvent
```

- ✓ **ActionEvent** has these three constructors:

#### Constructors:

<code>ActionEvent(Object src, int type, String cmd)</code>	Constructs an ActionEvent object.
<code>ActionEvent(Object src, int type, String cmd, int modifiers)</code>	Constructs an ActionEvent object with modifier keys.
<code>ActionEvent(Object src, int type, String cmd, long when, int modifiers)</code>	Constructs an ActionEvent object with the specified modifier keys and timestamp.

- ✓ Here, *src* is a reference to the object that generated this event.
- ✓ The type of the event is specified by *type*, and its command string is *cmd*.
- ✓ The argument *modifiers* indicates which modifier keys (alt, ctrl, meta, and/or shift) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

#### Methods:

String getActionCommand()	To obtain the command name for the invoking <b>ActionEvent</b> object. <b>For Example:</b> When a button is pressed, an action event is generated that has a command name equal to the label on that button.
int getModifiers()	Returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated.
long getWhen()	Returns the time at which the event took place. This is called the event's <i>timestamp</i> .

#### java.awt.AdjustmentEvent class:

- ✓ An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events.
- ✓ The **AdjustmentEvent** class defines integer constants that can be used to identify them.
- ✓ The constants and their meanings are shown here:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

- ✓ In addition, there is an integer constant, **ADJUSTMENT\_VALUE\_CHANGED** that indicates that a change has occurred.
- ✓ The AdjustmentEvent class declaration:

```
public class AdjustmentEvent extends AWTEvent
```

- ✓ The AdjustmentEvent class has two constructors:

#### Constructors:

AdjustmentEvent(Adjustable <i>src</i> , int <i>id</i> , int <i>type</i> , int <i>val</i> )	Constructs an AdjustmentEvent object with the specified Adjustable <i>src</i> , event <i>type</i> , adjustment <i>type</i> , and <i>val</i> .
<u>AdjustmentEvent</u> (Adjustable <i>src</i> , int <i>id</i> , int <i>type</i> , int <i>val</i> , boolean <i>isAdjusting</i> )	Constructs an AdjustmentEvent object with the specified Adjustable <i>src</i> , event <i>type</i> , adjustment <i>type</i> , and <i>val</i> .
<ul style="list-style-type: none"> <li>✓ Here, <i>src</i> is a reference to the object that generated this event.</li> <li>✓ The <i>id</i> specifies the event.</li> <li>✓ The type of the adjustment is specified by <i>type</i>, and its associated value is <i>val</i>.</li> </ul>	

**Methods:**

Adjustable getAdjustable()	Returns the object that generated the event.
int getAdjustmentType()	To obtain the type of the adjustment. It returns one of the constants defined by <b>AdjustmentEvent</b> .
int getValue()	To obtain the amount of the adjustment. <b>For example:</b> When a scroll bar is manipulated, this method returns the value represented by that change.

**java.awt.ComponentEvent class:**

- ✓ A **ComponentEvent** is generated when the size, position, or visibility of a component is changed.
- ✓ There are four types of component events.
- ✓ The **ComponentEvent** class defines integer constants that can be used to identify them.
- ✓ The constants and their meanings are shown here:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

- ✓ The **ComponentEvent** class declaration:

```
public class ComponentEvent extends AWTEvent
```

- ✓ **ComponentEvent** has this constructor:

ComponentEvent(Component <i>src</i> , int <i>type</i> )	Constructs a ComponentEvent object. Here, <i>src</i> is a reference to the object that generated this event. The type of the event is specified by <i>type</i> .
---	--

**Methods:**

Component getComponent()	Returns the component that generated the event.
--------------------------	---

**java.awt.ContainerEvent class:**

- ✓ A **ContainerEvent** is generated when a component is added to or removed from a container.
- ✓ There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT\_ADDED** and **COMPONENT\_REMOVED**. They indicate that a component has been added to or removed from the container.
- ✓ The **ContainerEvent** class declaration:

```
public class ContainerEvent extends ComponentEvent
```

- ✓ **ContainerEvent** is a subclass of **ComponentEvent** and has this constructor:

ContainerEvent(Component <i>src</i> , int <i>type</i> , Component <i>comp</i> )	Constructs a ContainerEvent object. Here, <i>src</i> is a reference to the container that generated this event. The type of the event is specified by <i>type</i> , and the component that has been added to or removed from the container is <i>comp</i> .
---	---

**Methods:**

Container getContainer()	To obtain a reference to the container that generated this event.
Component getChild()	Returns a reference to the component that was added to or removed from the container.

**java.awt.FocusEvent class:**

- ✓ A **FocusEvent** is generated when a component gains or loses input focus.
- ✓ These events are identified by the integer constants **FOCUS\_GAINED** and **FOCUS\_LOST**.
- ✓ The **FocusEvent** declaration:

```
public class FocusEvent extends ComponentEvent
```

- ✓ **FocusEvent** is a subclass of **ComponentEvent** and has these constructors:

FocusEvent(Component <i>src</i> , int <i>type</i> )	Constructs a FocusEvent object and identifies it as a permanent change in focus.
FocusEvent(Component <i>src</i> , int <i>type</i> , boolean <i>temporaryFlag</i> )	Constructs a FocusEvent object and identifies whether or not the change is temporary.
FocusEvent(Component <i>src</i> , int <i>type</i> , boolean <i>temporaryFlag</i> , Component <i>other</i> )	Constructs a FocusEvent object with the specified temporary state and opposite Component.
<ul style="list-style-type: none"> <li>✓ Here, <i>src</i> is a reference to the component that generated this event.</li> <li>✓ The type of the event is specified by <i>type</i>.</li> <li>✓ The argument <i>temporaryFlag</i> is set to <b>true</b> if the focus event is temporary.</li> <li>✓ Otherwise, it is set to <b>false</b>. (A temporary focus event occurs as a result of another user interface operation.</li> <li>✓ For example: Assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)</li> </ul>	

**Methods:**

Component getOppositeComponent()	To determine the <i>other</i> component. The opposite component is returned.
boolean isTemporary()	Returns <b>true</b> if the change is temporary. Otherwise, it returns <b>false</b> .

**java.awt.InputEvent class:**

- ✓ The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events.
- ✓ Its subclasses are **KeyEvent** and **MouseEvent**.
- ✓ The InputEvent class declaration:

```
public abstract class InputEvent extends ComponentEvent
```

- ✓ **InputEvent** defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event.

- ✓ The **InputEvent** class defined the following eight values to represent the modifiers, and these modifiers may still be found in older legacy code:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

- ✓ Because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

### Methods:

boolean isAltDown() boolean isAltGraphDown() boolean isControlDown() boolean isMetaDown() boolean isShiftDown()	To test if a modifier was pressed at the time an event is generated.
int getModifiers()	To obtain a value that contains all of the original modifier flags.
int getModifiersEx()	To obtain the extended modifiers.

### java.awt.ItemEvent class:

- ✓ An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- ✓ The **ItemEvent** declaration:

```
public class ItemEvent extends AWTEvent
```

- ✓ There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

- ✓ In addition, **ItemEvent** defines the integer constant, **ITEM\_STATE\_CHANGED** that signifies a change of state.
- ✓ **ItemEvent** has this constructor:

ItemEvent(ItemSelectable <i>src</i> , int <i>type</i> , Object <i>entry</i> , int <i>state</i> )	Constructs an ItemEvent object. Here, <i>src</i> is a reference to the component that generated this event. For example: This might be a list or choice element. The type of the event is specified by <i>type</i> . The specific item that generated the item event is passed in <i>entry</i> . The current state of that item is in <i>state</i> .
--	--

**Methods:**

Object getItem()	To obtain a reference to the item that changed.
ItemSelectable getItemSelectable()	To obtain a reference to the <b>ItemSelectable</b> object that generated an event.
int getStateChange()	Returns the state change (that is, <b>SELECTED</b> or <b>DESELECTED</b> ) for the event.

**java.awt.KeyEvent class:**

- ✓ A **KeyEvent** is generated when keyboard input occurs.
- ✓ There are three types of key events, which are identified by these integer constants: **KEY\_PRESSED**, **KEY\_RELEASED**, and **KEY\_TYPED**.
- ✓ The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated.
- ✓ Not all keypresses result in characters. **For example:** Pressing shift does not generate a character.
- ✓ There are many other integer constants that are defined by KeyEvent. **For example:** VK\_0 through VK\_9 and VK\_A through VK\_Z define the ASCII equivalents of the numbers and letters.
- ✓ Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

- ✓ The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.
- ✓ The **KeyEvent** class declaration:

```
public class KeyEvent extends InputEvent
```

- ✓ **KeyEvent** is a subclass of **InputEvent**. Here is one of its constructors:

KeyEvent(Component <i>src</i> , int <i>type</i> , long <i>when</i> , int <i>modifiers</i> , int <i>code</i> , char <i>ch</i> )	Here, <i>src</i> is a reference to the component that generated the event. The type of the event is specified by <i>type</i> . The system time at which the key was pressed is passed in <i>when</i> . The <i>modifiers</i> argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as <b>VK_UP</b> , <b>VK_A</b> , and so forth, is passed in <i>code</i> . The character equivalent (if one exists) is passed in <i>ch</i> . If no valid character exists, then <i>ch</i> contains <b>CHAR_UNDEFINED</b> . For <b>KEY_TYPED</b> events, <i>code</i> will contain <b>VK_UNDEFINED</b> .
--	---

**Methods:**

char getKeyChar()	Returns the character that was entered. If no valid character is available, then it returns <b>CHAR_UNDEFINED</b> .
int getKeyCode()	Returns the key code. When a <b>KEY_TYPED</b> event occurs, it returns <b>VK_UNDEFINED</b> .



## java.awt.MouseEvent class:

- ✓ There are eight types of mouse events.
- ✓ The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

- ✓ The **MouseEvent** class declaration:

```
public class MouseEvent extends InputEvent
```

- ✓ **MouseEvent** is a subclass of **InputEvent**. Here is one of its constructors:

MouseEvent(Component <i>src</i> , int <i>type</i> , long <i>when</i> , int <i>modifiers</i> , int <i>x</i> , int <i>y</i> , int <i>clicks</i> , boolean <i>triggersPopup</i> )	Here, <i>src</i> is a reference to the component that generated the event. The type of the event is specified by <i>type</i> . The system time at which the mouse event occurred is passed in <i>when</i> . The <i>modifiers</i> argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in <i>x</i> and <i>y</i> . The click count is passed in <i>clicks</i> . The <i>triggersPopup</i> flag indicates if this event causes a pop-up menu to appear on this platform.
--	--

### Methods:

int getX()	Return the X and Y coordinates of the mouse within the component when the event occurred.				
int getY()					
Point getPoint()	To obtain the coordinates of the mouse. It returns a <b>Point</b> object that contains the X,Y coordinates in its integer members: <b>x</b> and <b>y</b> .				
void translatePoint(int x, int y)	To change the location of the event. Here, the arguments x and y are added to the coordinates of the event.				
int getClickCount()	To obtains the number of mouse clicks for the current event.				
boolean isPopupTrigger()	To test if the current event causes a pop-up menu to appear on the platform.				
int getButton()	It returns a value that represents the button that caused the event. For most cases, the return value will be one of the following constants defined by <b>MouseEvent</b> :				
<table><tr><td>NOBUTTON</td><td>BUTTON1</td><td>BUTTON2</td><td>BUTTON3</td></tr></table> <p>The <b>NOBUTTON</b> value indicates that no button was pressed or released.</p>		NOBUTTON	BUTTON1	BUTTON2	BUTTON3
NOBUTTON	BUTTON1	BUTTON2	BUTTON3		

Point getLocationOnScreen()	To obtain the coordinates of the mouse relative to the screen rather than the component. Returns a <b>Point</b> object that contains both the X and Y coordinate.
int getXOnScreen()	To obtain the coordinates of the mouse relative to the screen rather than the component. The other two methods return the indicated coordinate.
int getYOnScreen()	

### java.awt.MouseWheelEvent class:

- ✓ The **MouseWheelEvent** class encapsulates a mouse wheel event. If a mouse has a wheel, it is typically located between the left and right buttons. Mouse wheels are used for scrolling.
- ✓ **MouseWheelEvent** defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

- ✓ **MouseWheelEvent** It is a subclass of **MouseEvent**. It's declaration:

```
public class MouseWheelEvent extends MouseEvent
```

- ✓ Here is one of the constructors defined by **MouseWheelEvent**:

<p>MouseWheelEvent(Component <i>src</i>, int <i>type</i>, long <i>when</i>, int <i>modifiers</i>, int <i>x</i>, int <i>y</i>, int <i>clicks</i>, boolean <i>triggersPopup</i>, int <i>scrollHow</i>, int <i>amount</i>, int <i>count</i>)</p>	<p>Constructs a MouseWheelEvent object with the specified <i>src</i> <i>modifiers</i>, <i>type</i>, <i>when</i>, <i>coordinates</i>, <i>scrollHow</i>, <i>scroll amount</i>, and wheel rotation <i>count</i>.</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Here, <i>src</i> is a reference to the object that generated the event.</li> <li><input type="checkbox"/> The type of the event is specified by <i>type</i>.</li> <li><input type="checkbox"/> The system time at which the mouse event occurred is passed in <i>when</i>.</li> <li><input type="checkbox"/> The <i>modifiers</i> argument indicates which modifiers were pressed when the event occurred.</li> <li><input type="checkbox"/> The coordinates of the mouse are passed in <i>x</i> and <i>y</i>.</li> <li><input type="checkbox"/> The number of clicks is passed in <i>clicks</i>.</li> <li><input type="checkbox"/> The <i>triggersPopup</i> flag indicates if this event causes a popup menu to appear on this platform.</li> <li><input type="checkbox"/> The <i>scrollHow</i> value must be either <b>WHEEL_UNIT_SCROLL</b> or <b>HEEL_BLOCK_SCROLL</b>.</li> <li><input type="checkbox"/> The number of units to scroll is passed in <i>amount</i>.</li> <li><input type="checkbox"/> The <i>count</i> parameter indicates the number of rotational units that the wheel moved.</li> </ul>
---	--

### Methods:

int getWheelRotation()	To obtain the number of rotational units. It returns the number of rotational units. If the value is positive, the wheel moved
------------------------	--



	counterclockwise. If the value is negative, the wheel moved clockwise.
double getPreciseWheelRotation()	Returns the number of "clicks" the mouse wheel was rotated, as a double. It supports high-resolution wheels
int getScrollType()	Returns the type of scrolling that should take place in response to this event. It returns either <b>WHEEL_UNIT_SCROLL</b> or <b>WHEEL_BLOCK_SCROLL</b> .
int getScrollAmount()	Returns the number of units that should be scrolled per click of mouse wheel rotation. To obtain the number of units to scroll, if the scroll type is <b>WHEEL_UNIT_SCROLL</b> .

### java.awt.TextEvent class:

- ✓ Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.
- ✓ **TextEvent** defines the integer constant **TEXT\_VALUE\_CHANGED**.
- ✓ The **TextEvent** class declaration:

```
public class TextEvent extends AWTEvent
```

- ✓ The one constructor for this class is shown here:

TextEvent(Object <i>src</i> , int <i>type</i> )	Constructs a TextEvent object. Here, <i>src</i> is a reference to the object that generated this event. The type of the event is specified by <i>type</i> .
---	---

### java.awt.WindowEvent class:

- ✓ There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them.
- ✓ The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

- ✓ The **WindowEvent** class declaration:

```
public class WindowEvent extends ComponentEvent
```

- ✓ **WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors:

WindowEvent(Window <i>src</i> , int <i>type</i> )	Constructs a WindowEvent object.
---	----------------------------------

	Here, <i>src</i> is a reference to the object that generated this event. The type of the event is <i>type</i> .
WindowEvent(Window <i>src</i> , int <i>type</i> , Window <i>other</i> )	Constructs a WindowEvent object with the specified opposite Window.
WindowEvent(Window <i>src</i> , int <i>type</i> , int <i>fromState</i> , int <i>toState</i> )	Constructs a WindowEvent object with the specified previous and new window states.
WindowEvent(Window <i>src</i> , int <i>type</i> , Window <i>other</i> , int <i>fromState</i> , int <i>toState</i> )	Constructs a WindowEvent object.
<input type="checkbox"/> Here, <i>other</i> specifies the opposite window when a focus or activation event occurs. <input type="checkbox"/> The <i>fromState</i> specifies the prior state of the window. <input type="checkbox"/> The <i>toState</i> specifies the new state that the window will have when a window state change occurs.	

### Methods:

Window getWindow()	Returns the <b>Window</b> object that generated the event.
Window getOppositeWindow()	Return the opposite window (when a focus or activation event has occurred).
int getOldState()	Return the previous window state.
int getNewState()	Return the current window state.

## SOURCES OF EVENTS

- ✓ The following table lists some of the user interface components that can generate the events described in the previous section.
- ✓ In addition to these graphical user interface elements, any class derived from **Component**, such as **Frame**, can generate events.
- ✓ **For example:** The program can receive key and mouse events from an instance of **Frame**.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	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.
Scroll bar	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 LISTENER INTERFACES

- ✓ As explained, the delegation event model has two parts: sources and listeners.
- ✓ Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.
- ✓ When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.
- ✓ The following table lists several commonly used listener interfaces and provides a brief description of the methods that they define.

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.

### java.awt.ActionListener interface:

- ✓ This interface defines the **actionPerformed()** method that is invoked when an action event occurs.
- ✓ The **ActionListener** interface declaration:

```
public interface ActionListener extends EventListener
```

- ✓ Its general form of actionPerformed() is shown here:

#### Abstract Methods:

void actionPerformed(ActionEvent ae)	Invoked when an action occurs.
--------------------------------------	--------------------------------

### java.awt.AdjustmentListener interface:

- ✓ The **AdjustmentListener** interface declaration:

```
public interface AdjustmentListener extends EventListener
```

- ✓ This interface defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs.
- ✓ Its general form is shown here:

**Abstract Methods:**

void adjustmentValueChanged(AdjustmentEvent ae)	Invoked when the value of the adjustable has changed.
---	---

**java.awt.ComponentListener interface:**

- ✓ The **ComponentListener** interface declaration:

public interface <b>ComponentListener</b> extends <u><b>EventListener</b></u>
---

- ✓ This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden.
- ✓ Their general forms are shown here:

**Abstract Methods:**

void componentResized(ComponentEvent ce)	Invoked when the component's size changes.
void componentMoved(ComponentEvent ce)	Invoked when the component's position changes.
void componentShown(ComponentEvent ce)	Invoked when the component has been made visible.
void componentHidden(ComponentEvent ce)	Invoked when the component has been made invisible.

**java.awt.ContainerListener interface:**

- ✓ The **ContainerListener** interface declaration:

public interface <b>ContainerListener</b> extends <u><b>EventListener</b></u>
---

- ✓ This interface contains two methods. When a component is added to a container, **componentAdded()** is invoked. When a component is removed from a container, **componentRemoved()** is invoked.
- ✓ Their general forms are shown here:

**Abstract Methods:**

void componentAdded(ContainerEvent ce)	Invoked when a component has been added to the container.
void componentRemoved(ContainerEvent ce)	Invoked when a component has been removed from the container.

**java.awt.FocusListener interface:**

- ✓ The **FocusListener** interface declaration:

public interface <b>FocusListener</b> extends <u><b>EventListener</b></u>
---

- ✓ This interface defines two methods. When a component obtains keyboard focus, **focusGained()** is invoked. When a component loses keyboard focus, **focusLost()** is called.
- ✓ Their general forms are shown here:

**Abstract Methods:**

void focusGained(FocusEvent <i>fe</i> )	Invoked when a component gains the keyboard focus.
void focusLost(FocusEvent <i>fe</i> )	Invoked when a component loses the keyboard focus.

**java.awt.ItemListener interface:**

- ✓ The **ItemListener** interface declaration:

```
public interface ItemListener extends EventListener
```

- ✓ This interface defines the `itemStateChanged()` method that is invoked when the state of an item changes.
- ✓ Its general form is shown here:

**Abstract Methods:**

void itemStateChanged(ItemEvent <i>ie</i> )	Invoked when an item has been selected or deselected by the user.
---	---

**java.awt.KeyListener interface:**

- ✓ The interface **KeyListener** declaration:

```
public interface KeyListener extends EventListener
```

- ✓ This interface defines three methods. The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively.
- ✓ The **keyTyped()** method is invoked when a character has been entered.
- ✓ **For Example:**
  - If a user presses and releases the 'a' key, three events are generated in sequence: key pressed, typed, and released.
  - If a user presses and releases the home key, two key events are generated in sequence: key pressed and released.
- ✓ The general forms of these methods are shown here:

**Abstract Methods:**

void keyPressed(KeyEvent <i>ke</i> )	Invoked when a key has been pressed.
void keyReleased(KeyEvent <i>ke</i> )	Invoked when a key has been released.
void keyTyped(KeyEvent <i>ke</i> )	Invoked when a key has been typed.

**java.awt.MouseListener interface:**

- ✓ The **MouseListener** interface declaration:

```
public interface MouseListener extends EventListener
```

- ✓ This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked()** is invoked.
- ✓ When the mouse enters a component, the **mouseEntered()** method is called. When it leaves, **mouseExited()** is called.
- ✓ The **mousePressed()** and **mouseReleased()** methods are invoked when the mouse is pressed and released, respectively.

- ✓ The general forms of these methods are shown here:

#### Abstract Methods:

void mouseClicked(MouseEvent <i>me</i> )	Invoked when the mouse button has been clicked (pressed and released) on a component.
void mouseEntered(MouseEvent <i>me</i> )	Invoked when the mouse enters a component.
void mouseExited(MouseEvent <i>me</i> )	Invoked when the mouse exits a component.
void mousePressed(MouseEvent <i>me</i> )	Invoked when a mouse button has been pressed on a component.
void mouseReleased(MouseEvent <i>me</i> )	Invoked when a mouse button has been released on a component.

#### java.awt.MouseMotionListener interface:

- ✓ The **MouseMotionListener** interface declaration:

```
public interface MouseMotionListener extends EventListener
```

- ✓ This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged.
- ✓ The **mouseMoved( )** method is called multiple times as the mouse is moved.
- ✓ Their general forms are shown here:

#### Abstract Methods:

void mouseDragged(MouseEvent <i>me</i> )	Invoked when a mouse button is pressed on a component and then dragged.
void mouseMoved(MouseEvent <i>me</i> )	Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

#### java.awt.MouseWheelListener interface:

- ✓ The **MouseWheelListener** interface declaration:

```
public interface MouseWheelListener extends EventListener
```

- ✓ This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved.
- ✓ Its general form is shown here:

#### Abstract Methods:

void mouseWheelMoved(MouseWheelEvent <i>mwe</i> )	Invoked when the mouse wheel is rotated.
---	--

#### java.awt.TextListener interface:

- ✓ The **TextListener** interface declaration:

```
public interface TextListener extends EventListener
```

- ✓ This interface defines the **textValueChanged( )** method that is invoked when a change occurs in a text area or text field.
- ✓ Its general form is shown here:



**Abstract Methods:**

void textValueChanged(TextEvent te)	Invoked when the value of the text has changed.
-------------------------------------	---

**java.awt.WindowFocusListener interface:**

- ✓ The **WindowFocusListener** interface declaration:

public interface <b>WindowFocusListener</b> extends <u><b>EventListener</b></u>
---

- ✓ This interface defines two methods: **windowGainedFocus()** and **windowLostFocus()**.
- ✓ These are called when a window gains or loses input focus.
- ✓ Their general forms are shown here:

**Abstract Methods:**

void windowGainedFocus(WindowEvent we)	Invoked when the Window is set to be the focused Window, which means that the Window, or one of its subcomponents, will receive keyboard events.
void windowLostFocus(WindowEvent we)	Invoked when the Window is no longer the focused Window, which means that keyboard events will no longer be delivered to the Window or any of its subcomponents.

**java.awt.WindowListener interface:**

- ✓ The **WindowListener** interface declaration:

public interface <b>WindowListener</b> extends <u><b>EventListener</b></u>
--

- ✓ This interface defines seven methods.
- ✓ The **windowActivated()** and **windowDeactivated()** methods are invoked when a window is activated or deactivated, respectively.
- ✓ If a window is iconified, the **windowIconified()** method is called.
- ✓ When a window is deiconified, the **windowDeiconified()** method is called.
- ✓ When a window is opened or closed, the **windowOpened()** or **windowClosed()** methods are called, respectively.
- ✓ The **windowClosing()** method is called when a window is being closed.
- ✓ The general forms of these methods are:

**Abstract Methods:**

void windowActivated(WindowEvent we)	Invoked when the Window is set to be the active Window.
void windowClosed(WindowEvent we)	Invoked when a window has been closed as the result of calling dispose on the window.
void windowClosing(WindowEvent we)	Invoked when the user attempts to close the window from the window's system menu.
void windowDeactivated(WindowEvent we)	Invoked when a Window is no longer the active Window.
void windowDeiconified(WindowEvent we)	Invoked when a window is changed from a minimized to a normal state.

void windowIconified(WindowEvent we)	Invoked when a window is changed from a normal to a minimized state.
void windowOpened(WindowEvent we)	Invoked the first time a window is made visible.

## USING THE DELEGATION EVENT MODEL

- ✓ After learning the theory behind the delegation event model, it is time to see it in practice.
- ✓ Using the delegation event model is quite easy.
- ✓ Just follow these two steps:
  - I. Implement the appropriate interface in the listener so that it can receive the type of event desired.
  - II. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.
- ✓ Remember that a source may generate several types of events. Each event must be registered separately.
- ✓ An object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.
- ✓ In all cases, an event handler must return quickly. An event handler must not retain control for an extended period of time.
- ✓ To see how the delegation model works in practice, look at examples that handle two common event generators: the mouse and keyboard.

## Some Key AWT GUI Concepts:

- ✓ It is important to point out that Java supports three GUI frameworks: the AWT, Swing, and JavaFX.
- ✓ The AWT was Java's first GUI framework, and for very limited GUI programs, it is the easiest to use.
- ✓ Swing, which is built on the foundation of the AWT, was Java's second GUI framework and still it is the most popular and widely used framework.
- ✓ JavaFX is the latest GUI framework, and it incorporates many advances in GUI design.
- ✓ To demonstrate the fundamentals of event handling, AWT-based GUI programs are an appropriate choice and are used here.

### How it works:

There are four key AWT features used by the following programs.

- **First**, all should create a top-level window by extending the **Frame** class. **Frame** defines a "normal" window.  
**For example:** It has minimize, maximize, and close boxes. It can be resized, covered, and redisplayed.
- **Second**, all should override the **paint()** method to display output in the window. This method is called by the run-time system to display output in the window.  
**For example:** It is called when a window is first shown and after a window has been hidden and then uncovered.
- **Third**, when the program needs output displayed, it does not call **paint()** directly. Instead, it calls **repaint()**. The **repaint()** tells the AWT to call **paint()**.
- **Finally**, when the top-level **Frame** window for an application is closed—for example, by clicking its close box—the program must explicitly exit, often through a call to **System.exit()**. Clicking the close box, by itself, does not cause the program to terminate. Therefore, it is necessary for an AWT-based GUI program to handle a window-close event.



## HANDLING MOUSE EVENTS

- ✓ To handle mouse events, the program must implement the **MouseListener** and the **MouseMotionListener** interfaces. (If it needs the program may also implement **MouseWheelListener**.)
- ✓ The following program demonstrates the process. It displays the current coordinates of the mouse in the program's window.
- ✓ Each time a button is pressed, the phrase "Button Down" is displayed at the location of the mouse pointer.
- ✓ Each time the button is released, the phrase "Button Released" is shown.
- ✓ If a button is clicked, a message stating this fact is displayed at the current mouse location.
- ✓ As the mouse enters or exits the window, a message is displayed that indicates what happened.
- ✓ When dragging the mouse, a \* is shown, which tracks with the mouse pointer as it is dragged.
- ✓ Notice that the two variables, **mouseX** and **mouseY**, they store the location of the mouse when a mouse pressed, released, or dragged event occurs.
- ✓ These coordinates are then used by **paint( )** to display output at the point of these occurrences.

### //Demonstrate several mouse event handlers.

```
import java.awt.*;
import java.awt.event.*;
public class MouseEventsDemo extends Frame implements MouseListener, MouseMotionListener
{
    String msg = "";
    int mouseX = 0, mouseY = 0;

    public MouseEventsDemo()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
        addWindowListener(new MyWindowAdapter());
    }

    //Handle mouse clicked.
    public void mouseClicked(MouseEvent me)
    {
        msg = msg+"--click received";
        repaint();
    }

    //Handle mouse entered.
    public void mouseEntered(MouseEvent me)
    {
        mouseX = 100;
        mouseY = 100;
        msg = "Mouse entered.";
        repaint();
    }

    //Handle mouse exited.
    public void mouseExited(MouseEvent me)
    {
        mouseX = 100;
        mouseY = 100;
```

```

        msg = "Mouse exited.";
        repaint();
    }

    //Handle button pressed.
    public void mousePressed(MouseEvent me)
    {
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Button down.";
        repaint();
    }

    //Handle button released.
    public void mouseReleased(MouseEvent me)
    {
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Button released.";
        repaint();
    }

    //Handle mouse dragged.
    public void mouseDragged(MouseEvent me)
    {
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "*" + " mouse at " + mouseX + ", " + mouseY;
        repaint();
    }

    //Handle mouse moved.
    public void mouseMoved(MouseEvent me)
    {
        msg = "Moving mouse at " + me.getX() + ", " + me.getY();
        repaint();
    }

    public void paint(Graphics g)
    {
        g.drawString(msg, mouseX, mouseY);
    }

    public static void main(String args[])
    {
        MouseEventsDemo appwin = new MouseEventsDemo();

        appwin.setSize(new Dimension(300, 300));
        appwin.setTitle("Mouse Events Demo");
        appwin.setVisible(true);
    }

```

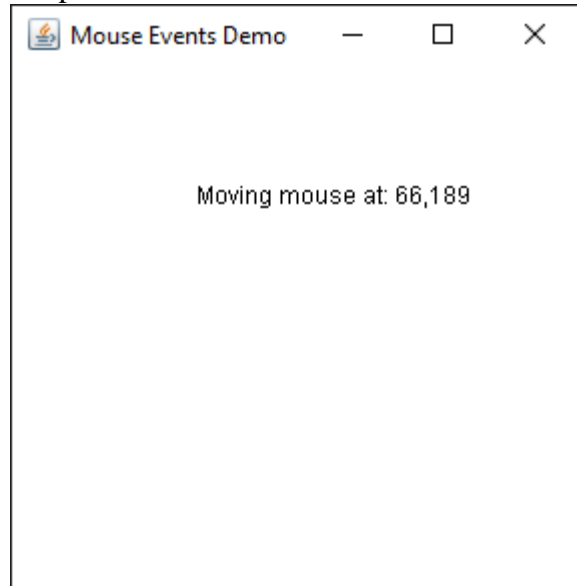
```

}

class MyWindowAdapter extends WindowAdapter
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
}

```

Output:



**Let's look closely at this example:**

- ✓ First, notice that **MouseEventsDemo** extends **Frame**. Thus, it forms the top-level window for the application.
- ✓ Next, notice that it implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process various types of mouse events.
- ✓ Also, Notice that **MouseEventsDemo** is both the source and the listener for these events. This works because **Frame** supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods. Being both the source and the listener for events is not uncommon for simple GUI programs.
- ✓ Inside the **MouseEventsDemo** constructor, the program registers itself as a listener for mouse events. This is done by calling **addMouseListener( )** and **addMouseMotionListener( )**. They are shown here:
  - void addMouseListener(MouseListener *ml*)
  - void addMouseMotionListener(MouseMotionListener *mml*)

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events.
- ✓ In this program, the same object is used for both. **MouseEventsDemo** then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces.
- ✓ These are the event handlers for the various mouse events. Each method handles its event and then returns.
- ✓ Notice that the **MouseEventsDemo** constructor also adds a **WindowListener**. This is needed to enable the program to respond to a window close event when the user clicks the close box. This listener uses an *adapter class* to implement the **WindowListener** interface.
- ✓ Adapter classes supply empty implementations of a listener interface, enabling us to override only the method or methods in which we are interested.

- ✓ In this case, the **windowClosing()** method is overridden. This method is called by the AWT when the window is closed. Here, it calls **System.exit()** to end the program.
- ✓ Now notice the mouse event handlers. Each time a mouse event occurs, **msg** is assigned a string that describes what happened and then **repaint()** is called.
- ✓ In this case, **repaint()** ultimately causes the AWT to call **paint()** to display output.
- ✓ Notice that **paint()** has a parameter of type **Graphics**. This class describes the *graphics context* of the program. It is required for output.
- ✓ The program uses the **drawString()** method provided by **Graphics** to actually display a string in the window at the specified X, Y location. The form used in the program is shown here:
  - `void drawString(String message, int x, int y)`  
 Here, *message* is the string to be output beginning at x, y. In a Java window, the upper-left corner is location 0, 0.
- ✓ As mentioned, **mouseX** and **mouseY** keep track of the location of the mouse. These values are passed to **drawString()** as the location at which output is displayed.
- ✓ Finally, the program is started by creating a **MouseEventsDemo** instance and then setting the size of the window, its title, and making the window visible.

## HANDLING KEYBOARD EVENTS

- ✓ To handle keyboard events, use the same general architecture as that shown in the mouse event example in the preceding section.
- ✓ The difference is that, here the **KeyListener** interface will be implemented.
- ✓ Before looking at the example, it is useful to review how key events are generated.
- ✓ When a key is pressed, a **KEY\_PRESSED** event is generated. This results in a call to the **keyPressed()** event handler.
- ✓ When the key is released, a **KEY\_RELEASED** event is generated and the **keyReleased()** handler is executed.
- ✓ If a character is generated by the keystroke, then a **KEY\_TYPED** event is sent and the **keyTyped()** handler is invoked.
- ✓ Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events.
- ✓ However, if the program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed()** handler.
- ✓ The following program demonstrates keyboard input. It echoes keystrokes to the window and shows the pressed/released status of each key.

**//Demonstrate the key event handlers.**

```
import java.awt.*;
import java.awt.event.*;
public class SimpleKey extends Frame implements KeyListener
{
    String msg = "";
    String keyState = "";

    public SimpleKey()
    {
        addKeyListener(this);
        addWindowListener(new MyWindowAdapter());
    }
}
```

```

//Handle a key press.
public void keyPressed(KeyEvent ke)
{
    keyState = "Key down.";
    repaint();
}

//Handle a key release.
public void keyReleased(KeyEvent ke)
{
    keyState = "Key up.";
    repaint();
}

//Handle key typed.
public void keyTyped(KeyEvent ke)
{
    msg+=ke.getKeyChar();
    repaint();
}

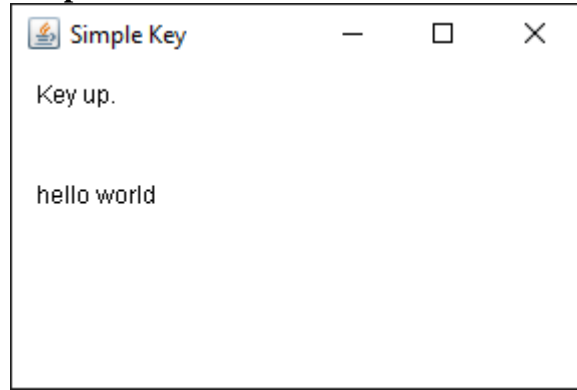
//Display keystrokes.
public void paint(Graphics g)
{
    g.drawString(msg, 20, 100);
    g.drawString(keyState, 20, 50);
}

public static void main(String[] args)
{
    SimpleKey appwin = new SimpleKey();
    appwin.setSize(new Dimension(300, 200));
    appwin.setTitle("Simple Key");
    appwin.setVisible(true);
}
}

class MyWindowAdapter extends WindowAdapter
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
}

```

### Output:



## ADAPTER CLASSES

- ✓ Java provides a special feature, called an *adapter class* that can simplify the creation of event handlers in certain situations.
- ✓ An adapter class provides an empty implementation of all methods in an event listener interface.
- ✓ Adapter classes are useful when only some of the events need to be received and handled by a particular event listener interface.
- ✓ Define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.
- ✓ For example: The **MouseMotionAdapter** class has two methods, **mouseDragged()** and **mouseMoved()**, which are the methods defined by the **MouseMotionListener** interface.
- ✓ If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged()**. The empty implementation of **mouseMoved()** would handle the mouse motion events for you.
- ✓ The following table lists several commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener, MouseMotionListener, and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

### Commonly Used Listener Interfaces Implemented by Adapter Classes

- ✓ The following program provides an example of an adapter. It uses **MouseAdapter** to respond to mouse click and mouse drag events.
- ✓ As shown in the above table, **MouseAdapter** implements all of the mouse listener interfaces. Thus, it can be used to handle all types of mouse events.
- ✓ The implementing class need to override only those methods that are used by your program.

- ✓ In the following example, **MyMouseAdapter** extends **MouseAdapter** and overrides the **mouseClicked()** and **mouseDragged()** methods. All other mouse events are silently ignored.
- ✓ Notice that the **MyMouseAdapter** constructor is passed a reference to the **AdapterDemo** instance. This reference is saved and then used to assign a string to **msg** and to invoke **repaint()** on the object that receives the event notification.
- ✓ As before, a **WindowAdapter** is used to handle a window closing event.

**//Demonstrate adapter class.**

```
import java.awt.*;
import java.awt.event.*;
public class AdapterDemo extends Frame
{
    String msg = "";
    public AdapterDemo()
    {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }

    //Display mouse information.
    public void paint(Graphics g)
    {
        g.drawString(msg, 20, 80);
    }

    public static void main(String[] args)
    {
        AdapterDemo appwin = new AdapterDemo();

        appwin.setSize(new Dimension(300, 200));
        appwin.setTitle("Adapter Demo");
        appwin.setVisible(true);
    }
}

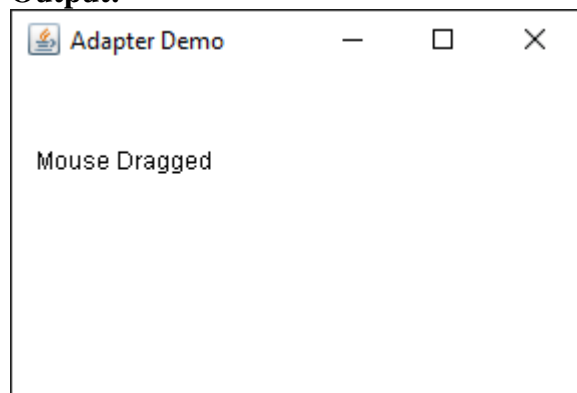
class MyMouseAdapter extends MouseAdapter
{
    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo ad)
    {
        this.adapterDemo = ad;
    }

    //Handle mouse clicked.
    public void mouseClicked(MouseEvent me)
    {
        adapterDemo.msg = "Mouse Clicked";
        adapterDemo.repaint();
    }
}
```

```
//Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
    adapterDemo.msg = "Mouse Dragged";
    adapterDemo.repaint();
}

class MyWindowAdapter extends WindowAdapter
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
}
```

#### Output:



## NESTED AND INNER CLASSES

- ✓ It is possible to define a class within another class; such classes are known as *nested classes*.
- ✓ The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A.
- ✓ A nested class has access to the members, including private members, of the class in which it is nested. The enclosing class does not have access to the members of the nested class.
- ✓ A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.
- ✓ There are two types of nested classes: *static* and *non-static*.
- ✓ A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
- ✓ The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.
- ✓ The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer\_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```
//Demonstrate an inner class.
```

```
class Outer
{
```



```

    int outer_x = 100;
    void test()
    {
        Inner in = new Inner();
        in.display();
    }

    class Inner
    {
        void display()
        {
            System.out.println("Display outer_x value:"+outer_x);
        }
    }
}

class InnerClassDemo
{
    public static void main(String[] args)
    {
        Outer out = new Outer();
        out.test();
    }
}

```

**Output:**

Display outer\_x value:100

- ✓ Inner classes can be used to simplify the code when using event adapter classes. To understand the benefit provided by inner classes, consider the program shown in the following listing.
- ✓ It *does not* use an inner class. Its goal is to display the string "Mouse Pressed" when the mouse is pressed.
- ✓ Similar to the approach used by the preceding example, a reference to the **MousePressedDemo** instance is passed to the **MyMouseAdapter** constructor and saved.
- ✓ This reference is used to assign a string to **msg** and invoke **repaint( )** on the object that received the event.

**//This program does NOT use an inner class.**

```

import java.awt.*;
import java.awt.event.*;
public class MousePressedDemo extends Frame
{
    String msg = "";
    public MousePressedDemo()
    {
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }

    public void paint(Graphics g)
    {
        g.drawString(msg, 20, 100);
    }
}

```

```

    public static void main(String[] args)
    {
        MousePressedDemo appwin = new MousePressedDemo();
        appwin.setSize(new Dimension(200, 150));
        appwin.setTitle("Mouse pressed Demo");
        appwin.setVisible(true);
    }
}

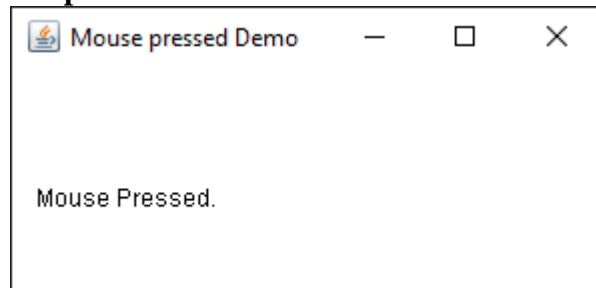
class MyMouseAdapter extends MouseAdapter
{
    MousePressedDemo mpd;
    public MyMouseAdapter(MousePressedDemo tmpd)
    {
        this.mpd = tmpd;
    }

    public void mousePressed(MouseEvent me)
    {
        mpd.msg = "Mouse Pressed.";
        mpd.repaint();
    }
}

class MyWindowAdapter extends WindowAdapter
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
}

```

**Output:**



- ✓ The following listing shows how the preceding program can be improved by using an inner class.

**//Event handling adapter classes program using inner classes.**

```

import java.awt.*;
import java.awt.event.*;
public class InnerClassDemo2 extends Frame
{
    String msg = "";
    public InnerClassDemo2()
    {
        addMouseListener(new MyMouseAdapter());
    }
}

```

```

        addWindowListener(new MyWindowAdapter());
    }

    class MyMouseAdapter extends MouseAdapter
    {
        public void mousePressed(MouseEvent me)
        {
            msg = "Mouse Pressed.";
            repaint();
        }
    }

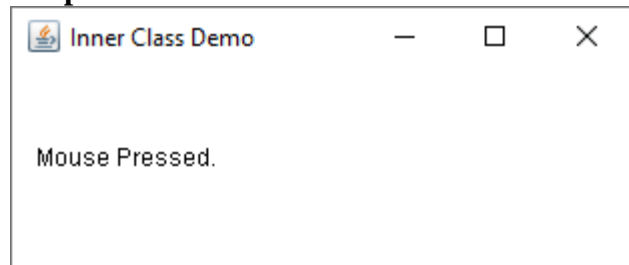
    class MyWindowAdapter extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    }

    public void paint(Graphics g)
    {
        g.drawString(msg, 20, 80);
    }

    public static void main(String[] args)
    {
        InnerClassDemo2 appwin = new InnerClassDemo2();
        appwin.setSize(new Dimension(200, 150));
        appwin.setTitle("Inner Class Demo");
        appwin.setVisible(true);
    }
}

```

#### Output:



## ANONYMOUS INNER CLASSES

- ✓ An *anonymous* inner class is one that is not assigned a name.
- ✓ This section illustrates how an anonymous inner class can facilitate the writing of event handlers.
- ✓ Consider the program shown in the following listing. As before, its goal is to display the string "Mouse Pressed" when the mouse is pressed.

**//Anonymous inner class demo.**

import java.awt.\*;

```

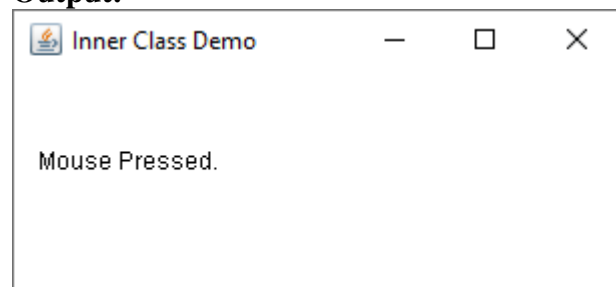
import java.awt.event.*;
public class AnonymousInnerClassDemo extends Frame
{
    String msg = "";
    public AnonymousInnerClassDemo()
    {
        addMouseListener(new MouseAdapter(){
            public void mousePressed(MouseEvent me)
            {
                msg = "Mouse Pressed.";
                repaint();
            }
        });
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }

    public void paint(Graphics g)
    {
        g.drawString(msg, 20, 80);
    }

    public static void main(String[] args)
    {
        InnerClassDemo2 appwin = new InnerClassDemo2();
        appwin.setSize(new Dimension(200, 150));
        appwin.setTitle("Inner Class Demo");
        appwin.setVisible(true);
    }
}

```

#### Output:



## UNDERSTANDING LAYOUT MANAGERS

- ✓ We can position components by using the layout managers.
- ✓ A layout manager automatically arranges your controls within a window by using some type of algorithm. While it is possible to lay out Java controls by hand, but it is not recommended for two main reasons.
- ✓ First, it is very tedious to manually lay out a large number of components.

- ✓ Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.
- ✓ Each **Container** object has a layout manager associated with it.

void setLayout(LayoutManager layoutObj)	<p>To set the layout manager.</p> <p>Here, <i>layoutObj</i> is a reference to the desired layout manager.</p> <p>To disable the layout manager and position components manually, pass <b>null</b> for <i>layoutObj</i>. To do this, determine the shape and position of each component manually, using the <b>setBounds( )</b> method defined by <b>Component</b>.</p>
---	--

- ✓ A layout manager is an instance of any class that implements the **LayoutManager** interface.

### java.awt.LayoutManager interface:

- ✓ The LayoutManager interface declaration:

public interface <b>LayoutManager</b>
---------------------------------------

### Abstract Methods:

Method	Description
void addLayoutComponent(String name, Component comp)	If the layout manager uses a per-component string, adds the component comp to the layout, associating it with the string specified by name.
void <u>layoutContainer</u> (Container parent)	Lays out the specified container.
Dimension minimumLayoutSize(Container parent)	Calculates the minimum size dimensions for the specified container, given the components it contains.
Dimension preferredLayoutSize(Container parent)	Calculates the preferred size dimensions for the specified container, given the components it contains.
void removeLayoutComponent(Component comp)	Removes the specified component from the layout.

- ✓ Each component that is being managed by a layout manager contains the **getPreferredSize( )** and **getMinimumSize( )** methods. These return the preferred and minimum size required to display each component.
- ✓ The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy.
- ✓ Java has several predefined LayoutManager classes, several of which are described next. Any of these classes can be used, that best fits the application.

### java.awt.FlowLayout class:

- ✓ **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.

- ✓ Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line.
- ✓ A small space is left between each component, above and below, as well as left and right.
- ✓ The **FlowLayout** class declaration:

```
public class FlowLayout extends Object implements LayoutManager, Serializable
```

- ✓ Here are the constructors for **FlowLayout**:

FlowLayout( )	It creates the default layout, which centers components and leaves five pixels of space between each component.
FlowLayout(int <i>how</i> )	To specify how each line is aligned. Valid values for <i>how</i> are as follows: <b>FlowLayout.LEFT</b> , <b>FlowLayout.CENTER</b> , <b>FlowLayout.RIGHT</b> , <b>FlowLayout.LEADING</b> , <b>FlowLayout.TRAILING</b> . These values specify left, center, right, leading edge, and trailing edge alignment, respectively.
FlowLayout(int <i>how</i> , int <i>horz</i> , int <i>vert</i> )	It allows you to specify the horizontal and vertical space left between components in <i>horz</i> and <i>vert</i> , respectively.

```
//Demonstration of FlowLayout class
```

Consider the program of Checkbox in preceding topics. Just change the values for *how* parameter as **FlowLayout.LEFT**, **FlowLayout.CENTER**, **FlowLayout.RIGHT**, **FlowLayout.LEADING**, **FlowLayout.TRAILING** layout manager setting to see the difference.

**Example:**

```
setLayout(new FlowLayout(FlowLayout.LEFT);
setLayout(new FlowLayout(FlowLayout.LEFT);
setLayout(new FlowLayout(FlowLayout.CENTER);
setLayout(new FlowLayout(FlowLayout.RIGHT);
setLayout(new FlowLayout(FlowLayout.LEADING);
setLayout(new FlowLayout(FlowLayout.TRAILING);
```

### **java.awt.BorderLayout class:**

- ✓ 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**.
- ✓ The **BorderLayout** class declaration:

```
public class BorderLayout extends Object implements LayoutManager2, Serializable
```

- ✓ Here are the constructors defined by **BorderLayout**:

BorderLayout( )	It creates a default border layout.
BorderLayout(int <i>horz</i> , int <i>vert</i> )	It allows to specify the horizontal and vertical space left between components in <i>horz</i> and <i>vert</i> , respectively.

- ✓ **BorderLayout** defines the following commonly used constants that specify the regions:

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

### Methods:

void add(Component <i>compRef</i> , Object <i>region</i> )	<ul style="list-style-type: none"> <li>✓ To add components to the layout manager.</li> <li>✓ While adding the components the above constants are used in the add() method.</li> <li>✓ Here, <i>compRef</i> is a reference to the component to be added, and <i>region</i> specifies where the component will be added.</li> </ul>
--	---

//Demonstrate the **BorderLayout** with a component in each layout area:

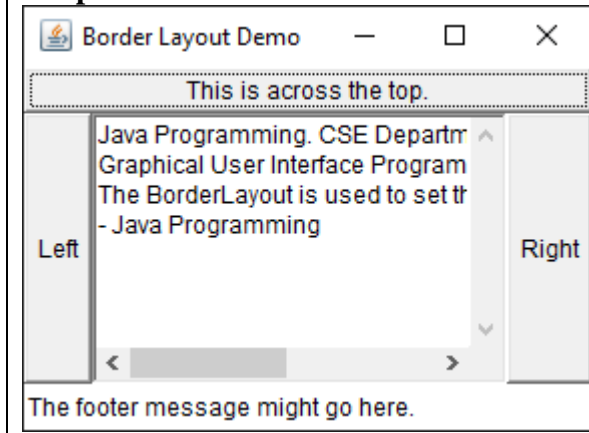
```
import java.awt.*;
import java.awt.event.*;
public class BorderLayoutDemo extends Frame
{
    public BorderLayoutDemo()
    {
        //BorderLayout is used by default so no need to set any layout.
        add(new Button("This is across the top."), BorderLayout.NORTH);
        add(new Label("The footer message might go here."), BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "Java Programming. CSE Department\n"+
            "Graphical User Interface Programming\n"+
            "The BorderLayout is used to set the default layout for the program\n"+
            "- Java Programming\n";
        add(new TextArea(msg), BorderLayout.CENTER);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public static void main(String[] args)
    {
        BorderLayoutDemo appwin = new BorderLayoutDemo();

        appwin.setSize(new Dimension(300, 220));
        appwin.setTitle("Border Layout Demo");
        appwin.setVisible(true);
    }
}
```

**Output:****USING INSETS**

- ✓ To leave a small amount of space between the container that holds the components and the window that contains it override the **getInsets()** method that is defined by **Container**.
- ✓ This method returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed.
- ✓ These values are used by the layout manager to inset the components when it lays out the window.
- ✓ The **Insets** class declaration:

```
public class Insets extends Object implements Cloneable, Serializable
```

- ✓ The constructor for **Insets** is shown here:

```
Insets(int top, int left, int bottom, int right)
```

The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.

**Methods:**

```
Insets getInsets()
```

When this method gets overridden, it return a new **Insets** object that contains the desired inset spacing.

//**BorderLayout** example modified so that it insets its components. The background color has been set to cyan to help make the insets more visible.

```
import java.awt.*;
import java.awt.event.*;
public class InsetsDemo extends Frame
{
    public InsetsDemo()
    {
        //BorderLayout is used by default so no need to set any layout.
        setBackground(Color.cyan);
        setLayout(new BorderLayout());
        add(new Button("This is across the top."), BorderLayout.NORTH);
        add(new Label("The footer message might go here."), BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);
    }
}
```



```

String msg = "Java Programming. CSE Department\n"+
             "Graphical User Interface Programming\n"+
             "The BorderLayout is used to set the default layout for the program\n"+
             "- Java Programming\n";
add(new TextArea(msg), BorderLayout.CENTER);

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
});

}

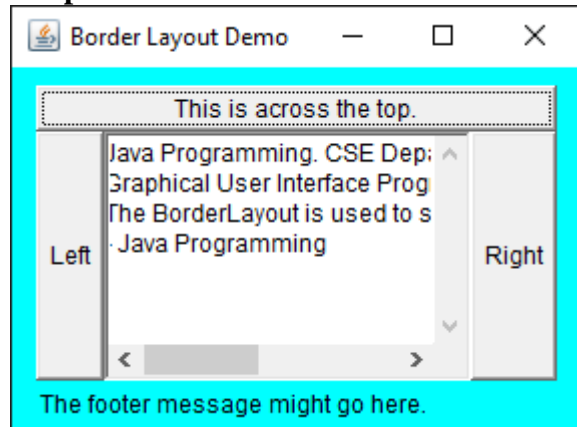
public Insets getInsets()
{
    return new Insets(40, 20, 10, 20);
}

public static void main(String[] args)
{
    InsetsDemo appwin = new InsetsDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("Border Layout Demo");
    appwin.setVisible(true);
}
}

```

#### Output:



#### java.awt.GridLayout class:

- ✓ **GridLayout** lays out components in a two-dimensional grid.
- ✓ When an object of **GridLayout** gets instantiated, it defines the number of rows and columns.
- ✓ The **GridLayout** class declaration:

```
public class GridLayout extends Object implements LayoutManager, Serializable
```

- ✓ The constructors supported by **GridLayout** are shown here:

<code>GridLayout( )</code>	It creates a single-column grid layout.
<code>GridLayout(int <i>numRows</i>, int <i>numColumns</i>)</code>	It creates a grid layout with the specified number of rows and columns.
<code>GridLayout(int <i>numRows</i>, int <i>numColumns</i>, int <i>horz</i>, int <i>vert</i>)</code>	The third form allows to specify the horizontal and vertical space left between components in <i>horz</i> and <i>vert</i> , respectively. Either <i>numRows</i> or <i>numColumns</i> can be zero. Specifying <i>numRows</i> as zero allows for unlimited-length columns. Specifying <i>numColumns</i> as zero allows for unlimited-length rows.

**//Demonstration a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:**

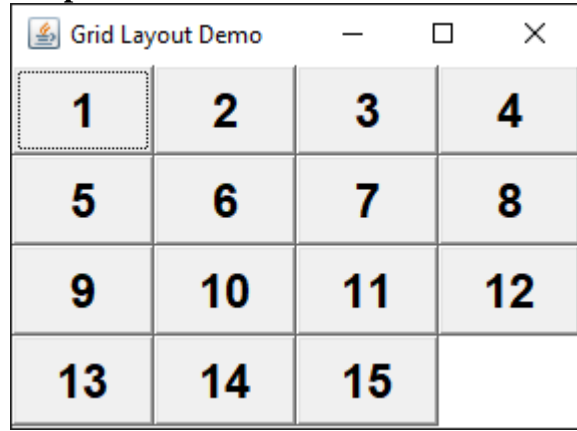
```
import java.awt.*;
import java.awt.event.*;
public class GridLayoutDemo extends Frame
{
    static final int n = 4;
    public GridLayoutDemo()
    {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));

        for(int i=0; i<n; i++)
        {
            for(int j=0; j<n; j++)
            {
                int k = i*n+j;
                if(k > 0)
                    add(new Button(""+k));
            }
        }

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public static void main(String[] args)
    {
        GridLayoutDemo appwin = new GridLayoutDemo();

        appwin.setSize(new Dimension(300, 220));
        appwin.setTitle("Grid Layout Demo");
        appwin.setVisible(true);
    }
}
```

**Output:****java.awt.CardLayout class:**

- ✓ The **CardLayout** class is unique among the other layout managers in that it stores several different layouts.
- ✓ Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.
- ✓ This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.
- ✓ It can be used to prepare the other layouts and have them hidden, ready to be activated when needed.
- ✓ The **CardLayout** class declaration:

```
public class CardLayout extends Object implements LayoutManager2, Serializable
```

- ✓ **CardLayout** provides these two constructors:

CardLayout( )	It creates a default card layout.
CardLayout(int <i>horz</i> , int <i>vert</i> )	It allows to specify the horizontal and vertical space left between components in <i>horz</i> and <i>vert</i> , respectively.

- ✓ Use of a card layout requires a bit more work than the other layouts.
- ✓ The cards are typically held in an object of type Panel.
- ✓ This panel must have CardLayout selected as its layout manager.
- ✓ The cards that form the deck are also typically objects of type Panel. There is need to create a panel that contains the deck and a panel for each card in the deck.
- ✓ Next, add to the appropriate panel the components that form each card. Then add these panels to the panel for which CardLayout is the layout manager.
- ✓ Finally, add this panel to the window.
- ✓ Once these steps are complete, provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

**Methods:**

void add(Component <i>panelRef</i> , Object <i>name</i> )	To add the card panels to the panel by name. Here, <i>name</i> is a string that specifies the name of the card whose panel is specified by <i>panelRef</i> .
void first(Container <i>deck</i> )	It causes the first card in the deck to be shown.
void last(Container <i>deck</i> )	To show the last card.

<code>void next(Container <i>deck</i>)</code>	To show the next card.
<code>void previous(Container <i>deck</i>)</code>	To show the previous card.
<code>void show(Container <i>deck</i>, String <i>cardName</i>)</code>	To displays the card whose name is passed in <i>cardName</i> .
Here, <i>deck</i> is a reference to the container (usually a panel) that holds the cards, and <i>cardName</i> is the name of a card.	

### //Demonstration of CardLayout.

The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Mac OS, Android, and Solaris are displayed in the other card.

```
import java.awt.*;
import java.awt.event.*;
public class CardLayoutDemo extends Frame
{
    Checkbox windows10, windows8, windows7, android, solaris, mac;
    Panel osCards;
    CardLayout cardLO;
    Button win, other;
    public CardLayoutDemo()
    {
        //Use a flow layout for the main frame.
        setLayout(new FlowLayout());

        win = new Button("Windows");
        other = new Button("Others");
        add(win);
        add(other);

        //set osCards panel to use CardLayout.
        cardLO = new CardLayout();
        osCards = new Panel();
        osCards.setLayout(cardLO);

        windows7 = new Checkbox("Windows 7", true);
        windows8 = new Checkbox("Windows 8");
        windows10 = new Checkbox("Windows 10");
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        //Add Windows check boxes to a panel.
        Panel winPan = new Panel();
        winPan.add(windows7);
        winPan.add(windows8);
        winPan.add(windows10);

        //Add other OS check boxes to a panel.
        Panel otherPan = new Panel();
        otherPan.add(android);
        otherPan.add(solaris);
    }
}
```

```

otherPan.add(mac);

//Add panels to card deck panel.
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Others");

//Add cards to main frame panel.
add(osCards);

//Use Lamda expressions to handle button events.
win.addActionListener((ae)->cardLO.show(osCards, "Windows"));
other.addActionListener((ae)->cardLO.show(osCards, "Others"));

addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent me)
    {
        cardLO.next(osCards);
    }
});

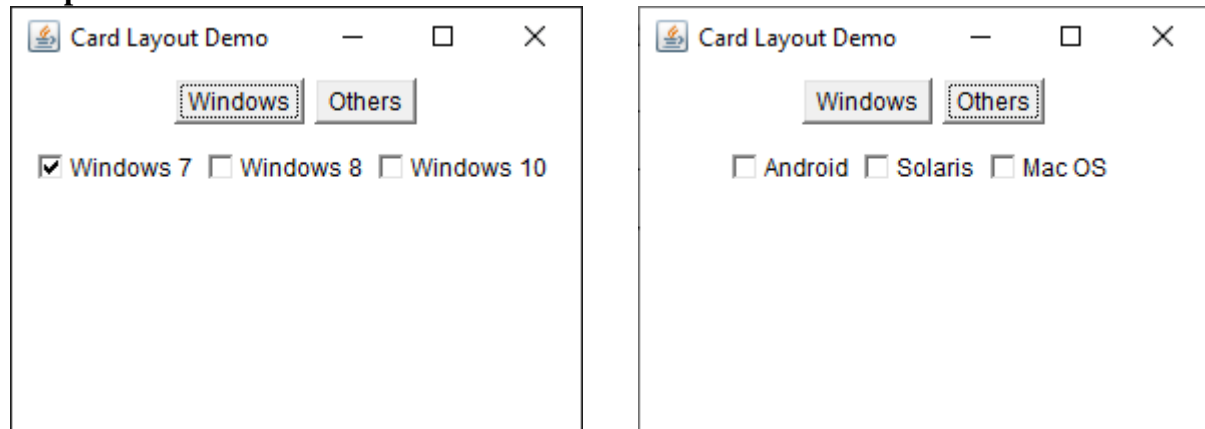
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
});
}

public static void main(String[] args)
{
    CardLayoutDemo appwin = new CardLayoutDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("Card Layout Demo");
    appwin.setVisible(true);
}
}

```

### Output:



## java.awt.GridBagLayout class:

- ✓ What makes the grid bag useful is that, it allows to specify the relative placement of components by specifying their positions within cells inside a grid.
- ✓ The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.
- ✓ This is why the layout is called a *grid bag*. It's a collection of small grids joined together.
- ✓ The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**.
- ✓ Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.
- ✓ The general procedure for using a grid bag is:
  - I. First create a new **GridBagLayout** object and to make it the current layout manager.
  - II. Then, set the constraints that apply to each component that will be added to the grid bag.
  - III. Finally, add the components to the layout manager.
- ✓ Although **GridBagLayout** is a bit more complicated than the other layout managers, it is quite easy to use once understood how it works.
- ✓ The **GridBagLayout** class declaration:

public class <b>GridBagLayout</b> extends <b>Object</b> implements <b>LayoutManager2</b> , <b>Serializable</b>
--

- ✓ **GridBagLayout** defines only one constructor, which is shown here:

GridBagLayout()	Creates a grid bag layout manager.
-----------------	------------------------------------

### Methods:

void setConstraints(Component <i>comp</i> , GridBagConstraints <i>cons</i> )	Sets the constraints for the specified component in this layout. Here, <i>comp</i> is the component for which the constraints specified by <i>cons</i> apply. This method sets the constraints that apply to each component in the grid bag.
---	--

- ✓ The key to successfully using GridBagLayout is the proper setting of the constraints, which are stored in a **GridBagConstraints** object.
- ✓ **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component.
- ✓ These are shown in the following table. Several are described in greater detail in the following discussion.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is <b>GridBagConstraints.CENTER</b> .
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are <b>GridBagConstraints.NONE</b> (the default), <b>GridBagConstraints.HORIZONTAL</b> , <b>GridBagConstraints.VERTICAL</b> , <b>GridBagConstraints.BOTH</b> .
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is <b>GridBagConstraints.RELATIVE</b> .
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is <b>GridBagConstraints.RELATIVE</b> .
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

- ✓ **GridBagConstraints** also defines several static fields that contain standard constraint values, such as **GridBagConstraints.CENTER** and **GridBagConstraints.VERTICAL**.
- ✓ When a component is smaller than its cell, we can use the **anchor** field to specify where within the cell the component's top-left corner will be located.
- ✓ There are three types of values that you can give to anchor. The first are absolute:

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

- ✓ The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages. The relative values are shown here:

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START



- ✓ The third type of values that can be given to **anchor** allows you to position components relative to the baseline of the row. These values are shown here:

GridBagConstraints.BASELINE	GridBagConstraints.BASELINE_LEADING
GridBagConstraints.BASELINE_TRAILING	GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE_LEADING	GridBagConstraints.ABOVE_BASELINE_TRAILING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_BASELINE_LEADING
GridBagConstraints.BELOW_BASELINE_TRAILING	

- ✓ The **weightx** and **weighty** fields are both quite important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero.
- ✓ When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns.
- ✓ The best way to understand how these values work is to experiment with them a bit.
- ✓ The gridwidth variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use **GridBagConstraints.REMAINDER**.
- ✓ To specify that a component use the next-to-last cell in a row, use **GridBagConstraints.RELATIVE**.
- ✓ The gridheight constraint works the same way, but in the vertical direction.
- ✓ To specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to ipadx. To pad vertically, assign a value to ipady.

//Demonstrate **GridBagLayout**.

Here is an example that uses **GridBagLayout** to demonstrate several of the points just discussed:

```
import java.awt.*;
import java.awt.event.*;
public class GridBagDemo extends Frame implements ItemListener
{
    String msg = "";
    Checkbox windows, android, solaris, mac;
    public GridBagDemo()
    {
        //Use a GridBagLayout.
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);

        //Define check boxes.
        windows = new Checkbox("Windows", true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        //Define the grid bag.

        //Use default row weight of 0 for first row.
        gbc.weightx = 1.0; //use a column weight of 1
        gbc.ipadx = 200; //pad by 200 units
```



```

gbc.insets = new Insets(0, 6, 0, 0);

gbc.anchor = GridBagConstraints.NORTHEAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(windows, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(android, gbc);

//Give second row a weight of 1.
gbc.weighty = 1.0;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(solaris, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(mac, gbc);

//Add the components.
add(windows);
add(android);
add(solaris);
add(mac);

//Register to receive item events.
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
});
}

public void itemStateChanged(ItemEvent ie)
{
    repaint();
}

public void paint(Graphics g)
{
    msg = "Current State:";
    g.drawString(msg, 20, 100);
    msg = " Windows: "+ windows.getState();
    g.drawString(msg, 30, 120);
    msg = " Android: "+ android.getState();
    g.drawString(msg, 30, 140);
}

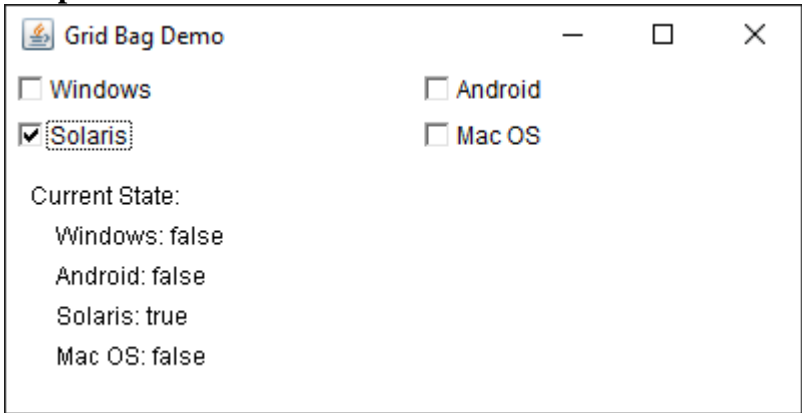
```

```
        msg = " Solaris: "+ solaris.getState();
        g.drawString(msg, 30, 160);
        msg = " Mac OS: "+ mac.getState();
        g.drawString(msg, 30, 180);
    }

    public static void main(String[] args)
    {
        GridBagDemo appwin = new GridBagDemo();

        appwin.setSize(new Dimension(250, 200));
        appwin.setTitle("Grid Bag Demo");
        appwin.setVisible(true);
    }
}
```

**Output:**



## INTRODUCING SWING

- ✓ The AWT is still a crucial part of Java, its component set is no longer widely used to create graphical user interfaces.
- ✓ Today, most programmers use Swing or JavaFX for this purpose.
- ✓ Swing is a framework that provides more powerful and flexible GUI components than does the AWT. As a result, it is the GUI that has been widely used by Java programmers for more than a decade.
- ✓ It is important to understand that the number of classes and interfaces in the Swing packages is quite large, and they can't all be covered here. (In fact, full coverage of Swing requires an entire book of its own.)

### The Origins of Swing

- ✓ 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.
- ✓ The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
- ✓ One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or *peers*.
- ✓ This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as *heavyweight*.
- ✓ The use of native peers led to several problems.

- A. First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere.
- B. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed.
- C. Third, the use of heavyweight components caused some frustrating restrictions.

**For example:** a heavyweight component was always opaque.

- ✓ Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed.
- ✓ The solution was Swing. 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. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

### Swing Is Built on the AWT

- ✓ Before moving on, it is necessary to make one important point: although Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it. Instead, Swing is built on the foundation of the AWT.
- ✓ This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT.
- ✓ Therefore, a basic understanding of the AWT and of event handling is required to use Swing.

### Two Key Swing Features

- ✓ Swing was created to address the limitations present in the AWT. It does this through two key features:
  - 1. Lightweight components
  - 2. Pluggable look and feel.
- ✓ Together they provide an elegant, yet easy-to-use solution to the problems of the AWT.
- ✓ More than anything else, it is these two features that define the essence of Swing.

#### 1. Swing Components Are Lightweight:

- ☐ With very few exceptions, 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.
- ☐ Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.
- ☐ As a result, each component will work in a consistent manner across all platforms.

#### 2. Swing Supports a Pluggable Look and Feel:

- ☐ Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing.
- ☐ This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.
- ☐ Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects.
- ☐ In other words, 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.

- ❑ Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply “plugged in.” Once this is done, all components are automatically rendered using that style.
- ❑ Pluggable look-and-feels offer several important advantages.
  - ✚ It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform.  
**For example:** If you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.
  - ✚ It is also possible to design a custom look and feel.
  - ✚ Finally, the look and feel can be changed dynamically at run time.

## Difference between AWT and Swing

- ✓ There are many differences between java awt and swing that are given below.

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

## The MVC Connection

- ✓ In general, a visual component is a composite of three distinct aspects:
  - The way that the component looks when rendered on the screen
  - The way that the component reacts to the user
  - The state information associated with the component
- ✓ No matter what architecture is used to implement a component, it must implicitly contain these three parts.
- ✓ Over the years, one component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.
- ✓ The MVC architecture is successful because each piece of the design corresponds to an aspect of a component.
- ✓ In MVC terminology:
  - A. The 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.
  - B. The 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.
  - C. The 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.  
**For instance:** Different view implementations can render the same component in different ways without affecting the model or the controller.
- ✓ Although the 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 the Separable Model architecture.
- ✓ Although Swing's component architecture is based on MVC, it does not use a classical implementation of it.
- ✓ Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program.
- ✓ It is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.
- ✓ To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate.
  - ✚ Models are defined by interfaces.  
**For example:** The model for a button is defined by the **ButtonModel** interface.
  - ✚ UI delegates are classes that inherit **ComponentUI**.  
**For example:** The UI delegate for a button is **ButtonUI**. Normally, the programs will not interact directly with the UI delegate.

## Components and Containers

- ✓ A Swing GUI consists of two key items: *components* and *containers*. This distinction is mostly conceptual because all containers are also components.
- ✓ The difference between the two is found in their intended purpose:
  - A *component* is an independent visual control, such as a push button or slider.
  - A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. To display a component, it must be held within a container.
- ✓ Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers.
- ✓ This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

## Components:

- ✓ In general, Swing components are derived from the **JComponent** class.
- ✓ **JComponent** provides the functionality that is common to all components.
- ✓ **For example:** **JComponent** supports the pluggable look and feel.
- ✓ **JComponent** inherits the AWT classes **Container** and **Component**.
- ✓ Thus, 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**.
- ✓ The following table shows the class names for Swing components (including those used as containers).

JApplet (Deprecated)	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		

- ✓ Notice that all component classes begin with the letter **J**.  
**For example:** The class for a label is **JLabel**; the class for a push button is **JButton**; and the class for a scroll bar is **JScrollBar**.

### Containers:

- ✓ Swing defines two types of containers.
- ✓ The first type are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**.
  - These containers do not inherit **JComponent**.
  - They inherit the AWT classes **Component** and **Container**.
  - Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.
  - As the name implies, a top-level container must be at the top of a containment hierarchy.
  - A top-level container is not contained within any other container.
  - Every containment hierarchy must begin with a top-level container.
  - One of the most commonly used top-level container for applications is **JFrame**.
  - The one used for applets is **JApplet**. Beginning with JDK 9 applets have been deprecated and are not recommended for new code.
- ✓ The second type of containers supported by Swing are lightweight containers. An example of a lightweight container is **JPanel**, which is a general-purpose container.
  - Lightweight containers *do* inherit **JComponent**.
  - Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. They can be used (such as **JPanel**) to create subgroups of related controls that are contained within an outer container.

### The Top-Level Container Panes

- ✓ Each top-level container defines a set of *panes*. At the top of the hierarchy is an instance of **JRootPane**. **JRootPane** is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar.
- ✓ The panes that comprise the root pane are called the *glass pane*, the *content pane*, and the *layered pane*.
- ✓ The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of **JPanel**. The glass pane enables us to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component.



- ✓ The layered pane is an instance of **JLayeredPane**. The layered pane allows components to be given a depth value. This value determines which component overlays another. (Thus, the layered pane lets you specify a Z-order for a component, although this is not something that you will usually need to do.)
- ✓ The layered pane holds the content pane and the (optional) menu bar.
- ✓ Although the glass pane and the layered panes are integral to the operation of a top-level container and serve important purposes, much of what they provide occurs behind the scene.
- ✓ The pane with which the application will interact the most is the content pane, because this is the pane to which the program will add visual components.
- ✓ For Example: when a component such as a button is added to a top-level container, it will be added to the content pane. By default, the content pane is an opaque instance of **JPanel**.

## The Swing Packages

- ✓ Swing is a very large subsystem and makes use of many packages.
- ✓ The following are the packages defined by Swing.

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

- ✓ Beginning the JDK 9, the Swing packages are part of the **java.desktop** module. The main package is **javax.swing**.
- ✓ This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

## A Simple Swing Application

- ✓ Swing programs differ from both the console-based programs and the AWT-based programs.  
**For example:** They use a different set of components and a different container hierarchy than the AWT.
- ✓ The best way to understand the structure of a Swing program is to work through an example.
- ✓ Before we begin, it is necessary to point out that there are two types of Java programs in which Swing was typically used.
  - The first is a desktop application, and that is the type of Swing program described here.
  - The second is the applet. Because applets are now deprecated and not recommended for new code, they are not discussed.
- ✓ The following program shows one way to write a Swing application. In the process, it demonstrates several key features of Swing.
- ✓ It uses two Swing components: **JFrame** and **JLabel**.
- ✓ **JFrame** is the top-level container that is commonly used for Swing applications.
- ✓ **JLabel** is the Swing component that creates a label, which is a component that displays information.
- ✓ The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output.
- ✓ The program uses a **JFrame** container to hold an instance of a **JLabel**. The label displays a short text message.

```
//Demonstration of a simple swing application.
```

```
import javax.swing.*;
```

```

class SwingDemo
{
    SwingDemo()
    {
        //Create a JFrame container.
        JFrame jf = new JFrame("A Simple Swing Application");

        //Give the frame an initial size.
        jf.setSize(275, 100);

        //Terminate the program when the user closes the application.
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a text-based label.
        JLabel jl = new JLabel("Swing means powerful GUI's");

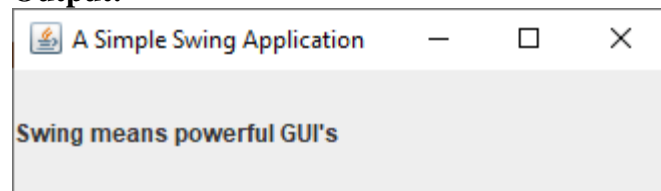
        //Add the label to the content pane.
        jf.add(jl);

        //Display the frame.
        jf.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run()
            {
                new SwingDemo();
            }
        });
    }
}

```

#### Output:



### Exploring Swing

- ✓ The discussion of Swing by presenting an overview of several Swing components, such as buttons, check boxes, is required to understand the working of swings.
- ✓ The Swing components provide rich functionality and allow a high level of customization.
- ✓ Because of space limitations, it is not possible to describe all of their features and attributes.
- ✓ The Swing component classes are shown here:



JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	TextField	JToggleButton	JTree

- ✓ These components all are lightweight, which means that they all are derived from **JComponent**.
- ✓ Also discussed is the **ButtonGroup** class, which encapsulates a mutually exclusive set of Swing buttons, and **ImageIcon**, which encapsulates a graphics image.
- ✓ Both are defined by Swing and packaged in **javax.swing**.

## JLabel and ImageIcon

### javax.swing.JLabel Class:

- ✓ **JLabel** is Swing's easiest-to-use component.
- ✓ It creates a label.
- ✓ **JLabel** can be used to display text and/or an icon.
- ✓ It is a passive component in that it does not respond to user input.
- ✓ **JLabel** class declaration:

public class <b>JLabel</b> extends <b>JComponent</b> implements <b>SwingConstants</b> , <b>Accessible</b>
---

- ✓ **JLabel** defines several constructors. Here are three of them:

JLabel(Icon <i>icon</i> )	Creates a JLabel instance with the specified image.
JLabel(String <i>str</i> )	Creates a JLabel instance with the specified <i>str</i> .
JLabel(String <i>str</i> , Icon <i>icon</i> , int <i>align</i> )	Creates a JLabel instance with the specified <i>str</i> , <i>icon</i> , and horizontal alignment <i>align</i> .
<ul style="list-style-type: none"> <li>✓ Here, <i>str</i> and <i>icon</i> are the text and icon used for the label.</li> <li>✓ The <i>align</i> argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label.</li> <li>✓ It must be one of the following values: <b>LEFT</b>, <b>RIGHT</b>, <b>CENTER</b>, <b>LEADING</b>, or <b>TRAILING</b>.</li> <li>✓ These constants are defined in the <b>SwingConstants</b> interface, along with several others used by the Swing classes.</li> </ul>	

### javax.swing.ImageIcon class:

- ✓ The easiest way to obtain an icon is to use the **ImageIcon** class.
- ✓ **ImageIcon** implements **Icon** and encapsulates an image.
- ✓ Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor.
- ✓ There are several ways to provide the image, including reading it from a file or downloading it from a URL.
- ✓ Here is the **ImageIcon** constructor used by the example in this section:

ImageIcon(String <i>filename</i> )	It obtains the image in the file named <i>filename</i> .
------------------------------------	--

### Methods:

Icon getIcon()	To obtain the icon associated with the label.
String getText()	To obtain the text associated with the label.
void setIcon(Icon <i>icon</i> )	To set the icon associated with a label.
void setText(String <i>str</i> )	To set the text associated with a label.

//Demonstration of JLabel and ImageIcon classes.

- ☐ The following program illustrates how to create and display a label containing both an icon and a string. It begins by creating an **ImageIcon** object for the file **flower.png**, which depicts a flower.
- ☐ This is used as the second argument to the **JLabel** constructor.
- ☐ The first and last arguments for the **JLabel** constructor are the label text and the alignment.
- ☐ Finally, the label is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
public class JLabelDemo
{
    JLabelDemo()
    {
        //Setup JFrame.
        JFrame jf = new JFrame("JLabel Demo");
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(260, 200);

        //Create an icon.
        ImageIcon ii = new ImageIcon("flower.png");

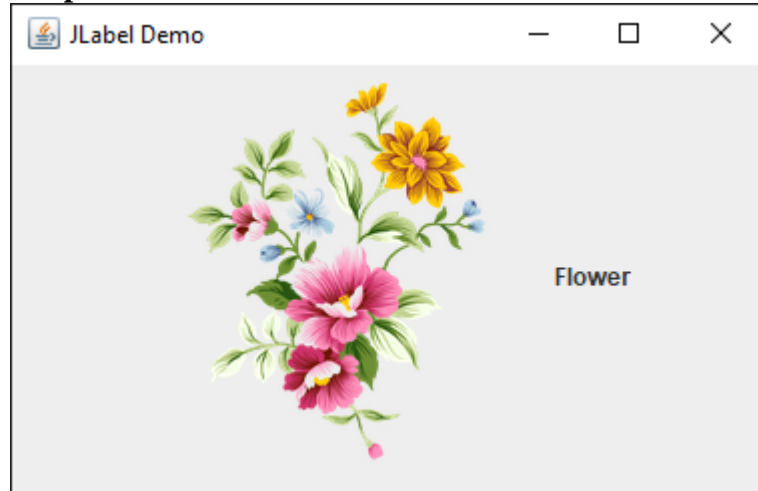
        //create a label.
        JLabel jl = new JLabel("Flower",ii,JLabel.CENTER);

        //Add the label to the content pane.
        jf.add(jl);

        //Display the frame.
        jf.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run()
            {
                new JLabelDemo();
            }
        });
    }
}
```

### Output:



### javax.swing.JTextField class

- ✓ **JTextField** is the simplest Swing text component.
- ✓ It is also probably the most widely used text component.
- ✓ **JTextField** allows you to edit one line of text.
- ✓ It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components.
- ✓ **JTextField** uses the **Document** interface for its model.
- ✓ The **JTextField** class declaration:

```
public class JTextField extends JTextComponent implements SwingConstants
```

- ✓ Three of **JTextField**'s constructors are shown here:

<code>JTextField(int cols)</code>	Constructs a new empty TextField with the specified number of <i>cols</i> .
<code>JTextField(String str, int cols)</code>	Constructs a new TextField initialized with the specified <i>str</i> and <i>cols</i> .
<code>JTextField(String str)</code>	Constructs a new TextField initialized with the specified <i>str</i> .
<ul style="list-style-type: none"><li>✓ Here, <i>str</i> is the string to be initially presented, and <i>cols</i> is the number of columns in the text field.</li><li>✓ If no string is specified, the text field is initially empty.</li><li>✓ If the number of columns is not specified, the text field is sized to fit the specified string.</li></ul>	

- ✓ **JTextField** generates events in response to user interaction.
- ✓ **For example:** An **ActionEvent** is fired when the user presses enter. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in `javax.swing.event`.) Other events are also possible.
- ✓ In many cases, the program will not need to handle these events. Instead, it will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call `getText()`.

#### //Demonstrate **JTextField** class.

- ☐ The following example illustrates **JTextField**.
- ☐ It creates a **JTextField** and adds it to the content pane.
- ☐ When the user presses enter, an action event is generated. This is handled by displaying the text in a label.

```
import java.awt.*;  
import java.awt.event.*;
```

```

import javax.swing.*;
public class JtextFieldDemo
{
    JtextFieldDemo()
    {
        //Setup JFrame.
        JFrame jf = new JFrame("JtextField Demo");
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(260, 120);

        //Add a text field to the content pane.
        JtextField jtf = new JtextField(15);
        jf.add(jtf);

        //Add a label.
        JLabel jl = new JLabel();
        jf.add(jl);

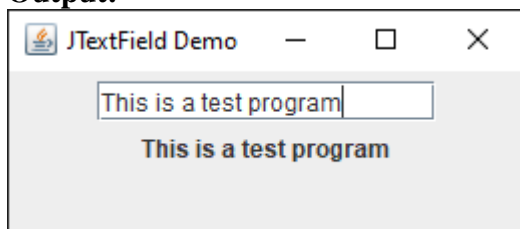
        //Handle action events.
        jtf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae)
            {
                jl.setText(jtf.getText());
            }
        });

        //Display the frame.
        jf.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run()
            {
                new JtextFieldDemo();
            }
        });
    }
}

```

#### Output:



## The Swing Buttons

- ✓ Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**.
- ✓ All are subclasses of the **AbstractButton** class, which extends **JComponent**.
- ✓ Thus, all buttons share a set of common properties.

#### **javax.swing.AbstractButton class:**

- ✓ The **AbstractButton** declaration:

public abstract class <b>AbstractButton</b> extends <b>JComponent</b> implements <b>ItemSelectable</b> , <b>SwingConstants</b>
--

- ✓ **AbstractButton** contains many methods that allows to control the behavior of buttons.  
For example: It can be used to define different icons that are displayed for the button when it is disabled, pressed, or selected.
- ✓ Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button.

#### **Methods:**

void setDisabledIcon(Icon <i>di</i> )	Sets the disabled icon for the button.
void setPressedIcon(Icon <i>pi</i> )	Sets the pressed icon for the button.
void setSelectedIcon(Icon <i>si</i> )	Sets the selected icon for the button.
void setRolloverIcon(Icon <i>ri</i> )	Sets the rollover icon for the button.
String getText()	Returns the button's text.
void setText(String <i>str</i> )	Sets the button's text.
✓ Here, <i>di</i> , <i>pi</i> , <i>si</i> , and <i>ri</i> are the icons to be used for the indicated purpose.	
✓ Here, <i>str</i> is the text to be associated with the button.	

- ✓ The model used by all buttons is defined by the **ButtonModel** interface. A button generates an action event when it is pressed. Other events are possible.

#### **javax.swing.JButton Class:**

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

public class <b>JButton</b> extends <b>AbstractButton</b> implements <b>Accessible</b>
--

- ✓ Three of its constructors are shown here:

JButton(Icon <i>icon</i> )	Creates a button with an <i>icon</i> .
JButton(String <i>str</i> )	Creates a button with <i>str</i> .
JButton(String <i>str</i> , Icon <i>icon</i> )	Creates a button with initial <i>str</i> and an <i>icon</i> .
Here, <i>str</i> and <i>icon</i> are the string and icon used for the button.	

- ✓ When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed( )** method of the registered **ActionListener**, the *action command* string associated with the button can be obtained. By default, this is the string displayed inside the button.
- ✓ To set the action command **setActionCommand( )** can be used on the button, and to obtain the action command call **getActionCommand( )** on the event object.
- ✓ The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives an easy way to determine which button was pressed.

//Demonstrating <b>JButton</b> class.
---------------------------------------

The following program demonstrates an icon-based button. It displays four push buttons and a label. Each button displays an icon that represents a timepiece. When a button is pressed, the name of that timepiece is displayed in the label.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JButtonDemo implements ActionListener
{
    JLabel jl;
    JButtonDemo()
    {
        //Setup JFrame.
        JFrame jf = new JFrame("JButton Demo");
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(300, 300);

        //Add a buttons to the content pane.
        ImageIcon brain = new ImageIcon("JButton/brain.png");
        JButton jb = new JButton(brain);
        jb.setActionCommand("Brain");
        jb.addActionListener(this);
        jf.add(jb);

        ImageIcon flower = new ImageIcon("JButton/flower.png");
        jb = new JButton(flower);
        jb.setActionCommand("Flower");
        jb.addActionListener(this);
        jf.add(jb);

        ImageIcon key = new ImageIcon("JButton/key.png");
        jb = new JButton(key);
        jb.setActionCommand("Key");
        jb.addActionListener(this);
        jf.add(jb);

        ImageIcon student = new ImageIcon("JButton/student.png");
        jb = new JButton(student);
        jb.setActionCommand("Student");
        jb.addActionListener(this);
        jf.add(jb);

        //Create and add label to the content pane.
        jl = new JLabel("Choose an icon...");
        jf.add(jl);

        //Display the frame.
        jf.setVisible(true);
    }

    //Handle button events.
```

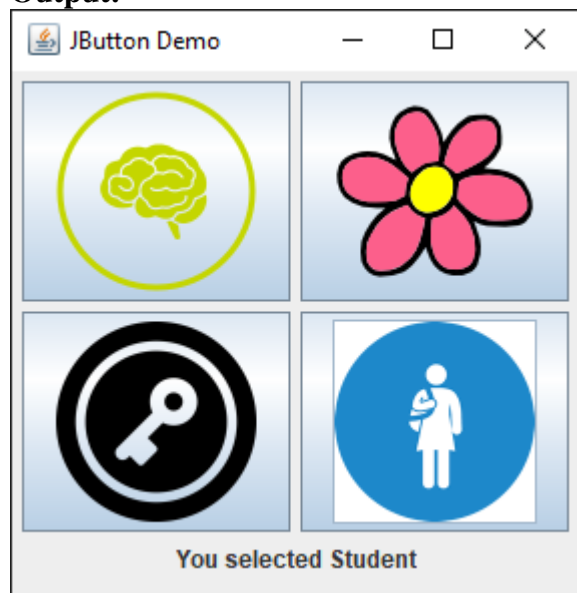
```

public void actionPerformed(ActionEvent ae)
{
    jl.setText("You selected "+ae.getActionCommand());
}

public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable() {
        public void run()
        {
            new JButtonDemo();
        }
    });
}
}

```

### Output:



### javax.swing.JToggleButton class:

- ✓ A useful variation on the push button is called a *toggle button*.
- ✓ A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released.
- ✓ That is, when a toggle button is pressed, it stays pressed rather than popping back up as a regular push button does. When it is pressed a second time, it releases (pops up).
- ✓ Therefore, each time a toggle button is pushed, it toggles between its two states.
- ✓ Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**.
- ✓ The **JToggleButton** class declaration:

```
public class JToggleButton extends AbstractButton implements Accessible
```

- ✓ In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**.
- ✓ Thus, **JToggleButton** defines the basic functionality of all twostate components.
- ✓ **JToggleButton** defines several constructors:

JToggleButton(String <i>str</i> )	This creates a toggle button that contains the text passed in <i>str</i> . By default, the button is in the off position.
-----------------------------------	---

- ✓ Like **JButton**, **JToggleButton** generates an action event each time it is pressed. Unlike **JButton**, **JToggleButton** also generates an item event.
- ✓ This event is used by those components that support the concept of selection. When a **JToggleButton** is pressed in, it is selected.
- ✓ When it is popped out, it is deselected. To handle item events, you must implement the **ItemListener** interface.
- ✓ Each time an item event is generated, it is passed to the **itemStateChanged()** method defined by **ItemListener**.
- ✓ Inside **itemStateChanged()**, the **getItem()** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event. A reference to the button class is returned, cast this reference to **JToggleButton**.
- ✓ The easiest way to determine a toggle button's state is by calling the **isSelected()** method (inherited from **AbstractButton**) on the button that generated the event. It is shown here:

boolean isSelected()	It returns true if the button is selected and false otherwise.
----------------------	--

//Demonstrating **JToggleButton** class.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls **isSelected()** to determine the button's state.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JToggleButtonDemo
{
    public JToggleButtonDemo()
    {
        //Setup the JFrame.
        JFrame jf = new JFrame("JToggleButton Demo");
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(300, 100);

        //Create a label.
        JLabel jl = new JLabel("Button is off");

        //Make a toggle button.
        JToggleButton jtb = new JToggleButton("On/Off");

        //Add an item listener for the toggle button.
        jtb.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent ie) {
                if(jtb.isSelected())
                    jl.setText("Button is on.");
                else
                    jl.setText("Button is off.");
            }
        })
    }
}
```



```

    });

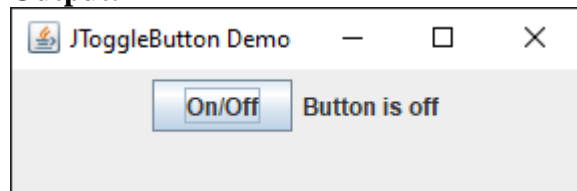
    //Add the toggle button and label to the content pane.
    jf.add(jtb);
    jf.add(jl);

    //Display the frame.
    jf.setVisible(true);
}

public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable() {
        public void run()
        {
            new JToggleButtonDemo();
        }
    });
}
}

```

#### Output:



### javax.swing.JCheckBox class:

- ✓ The **JCheckBox** class provides the functionality of a check box.
- ✓ Its immediate superclass is **JToggleButton**, which provides support for two-state buttons.
- ✓ The **JCheckBox** class declaration:

```
public class JCheckBox extends JToggleButton implements Accessible
```

- ✓ **JCheckBox** defines several constructors. The one used here is:

JCheckBox(String <i>str</i> )	It creates a check box that has the text specified by <i>str</i> as a label.
-------------------------------	--

- ✓ When the user selects or deselects a check box, an **ItemEvent** is generated. To obtain a reference to the **CheckBox** that generated the event, call **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.

//Demonstrating **JCheckBox** class.

- ✓ The following example illustrates check boxes.
- ✓ It displays four check boxes and a label.
- ✓ When the user clicks a check box, an **ItemEvent** is generated.
- ✓ Inside the **itemStateChanged()** method, **getItem()** is called to obtain a reference to the **JCheckBox** object that generated the event.
- ✓ Next, a call to **isSelected()** determines if the box was selected or cleared.

✓ The **getText()** method gets the text for that check box and uses it to set the text inside the label.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JCheckBoxDemo implements ItemListener
{
    JLabel jl;
    public JCheckBoxDemo()
    {
        //Setup the JFrame.
        JFrame jf = new JFrame("JCheckBox Demo");
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(250, 100);

        //Add checkboxes to the content pane.
        JCheckBox cb = new JCheckBox("C");
        cb.addItemListener(this);
        jf.add(cb);

        cb = new JCheckBox("C++");
        cb.addItemListener(this);
        jf.add(cb);

        cb = new JCheckBox("Java");
        cb.addItemListener(this);
        jf.add(cb);

        cb = new JCheckBox("Perl");
        cb.addItemListener(this);
        jf.add(cb);

        //Create a label.
        jl = new JLabel("Select languages");
        jf.add(jl);

        //Display the frame.
        jf.setVisible(true);
    }

    //Handle item events for the check boxes.
    public void itemStateChanged(ItemEvent ie)
    {
        JCheckBox cb = (JCheckBox)ie.getItem();

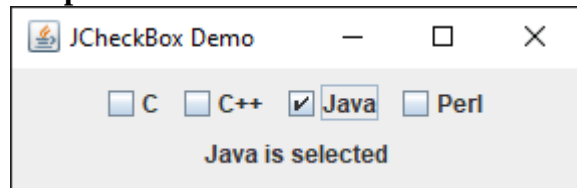
        if(cb.isSelected())
            jl.setText(cb.getText() + " is selected");
        else
            jl.setText(cb.getText() + " is cleared");
    }
}
```

```

public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable() {
        public void run()
        {
            new JCheckBoxDemo();
        }
    });
}
}

```

#### Output:



### javax.swing.JRadioButton class:

- ✓ Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time.
- ✓ They are supported by the **JRadioButton** class, which extends **JToggleButton**.
- ✓ The **JRadioButton** declaration is:

```
public class JRadioButton extends JToggleButton implements Accessible
```

- ✓ **JRadioButton** provides several constructors. The one used in the example is shown here:

JRadioButton(String <i>str</i> )	Here, <i>str</i> is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.
----------------------------------	---

- ✓ In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time.  
**For example:** If a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.
- ✓ A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton ab)	To add elements to the button group. Here, ab is a reference to the button to be added to the group.
-----------------------------	--

- ✓ A **JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the **ActionListener** interface. The only method defined by **ActionListener** is **actionPerformed()**.
- ✓ A number of different ways can be used to determine which button was selected.
  - First, by calling **getActionCommand()** its action command can be checked. By default, the action command is the same as the button label, but the action command can be set to something else by calling **setActionCommand()** on the radio button.
  - Second, the **getSource()** on the **ActionEvent** object can be called to check that reference against the buttons.

- Third, the **isSelected()** method can be called on each button to check each radio button and to find out which one is currently selected.
  - Finally, each button could use its own action event handler implemented as either an anonymous inner class or a lambda expression.
- ✓ Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

//Demonstrating **JRadioButton** class.

- ✓ The following example illustrates how to use radio buttons.
- ✓ Three radio buttons are created. The buttons are then added to a button group.
- ✓ As explained, this is necessary to cause their mutually exclusive behavior.
- ✓ Pressing a radio button generates an action event, which is handled by **actionPerformed()**.
- ✓ Within that handler, the **getActionCommand()** method gets the text that is associated with the radio button and uses it to set the text within a label.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JRadioButtonDemo implements ActionListener
{
    JLabel jl;
    public JRadioButtonDemo()
    {
        //Setup the JFrame.
        JFrame jf = new JFrame("JRadioButton Demo");
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(300, 100);

        //Create Radio buttons and add them to the content pane.
        JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        jf.add(b1);

        JRadioButton b2 = new JRadioButton("B");
        b2.addActionListener(this);
        jf.add(b2);

        JRadioButton b3 = new JRadioButton("C");
        b3.addActionListener(this);
        jf.add(b3);

        //Define a button group.
        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);

        //Create a label and add it to the content pane.
        jl = new JLabel("Select One");
        jf.add(jl);
    }
}
```

```

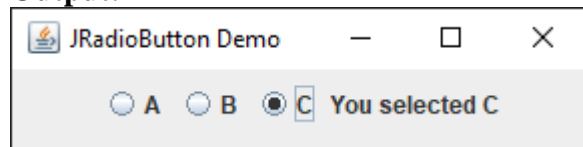
        //Display the frame.
        jf.setVisible(true);
    }

    //Handle button selection.
    public void actionPerformed(ActionEvent ae)
    {
        jl.setText("You selected "+ae.getActionCommand());
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run()
            {
                new JRadioButtonDemo();
            }
        });
    }
}

```

#### Output:



### javax.swing.JScrollPane class:

- ✓ **JScrollPane** is a lightweight container that automatically handles the scrolling of another component.
- ✓ The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**.
- ✓ In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane.
- ✓ Because **JScrollPane** automates scrolling, it usually eliminates the need to manage individual scroll bars.
- ✓ The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport.
- ✓ In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed.  
**For example:** If the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.
- ✓ The **JScrollPane** class declaration:

```
public class JScrollPane extends JComponent implements ScrollPaneConstants, Accessible
```

- ✓ **JScrollPane** defines several constructors. The one used in this chapter is shown here:

```
JScrollPane(Component comp) | The component to be scrolled is specified by comp.
```

- ✓ Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.
- ✓ Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

//Demonstrating **JScrollPane** class.

- ✓ The following example illustrates a scroll pane.
- ✓ First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns.
- ✓ This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically.
- ✓ Scroll bars can be used to scroll the buttons into view.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JScrollPaneDemo
{
    public JScrollPaneDemo()
    {
        //Setup the JFrame.
        JFrame jf = new JFrame("JScrollPane Demo");
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(400, 400);

        //Create a panel and add 400 buttons to it.
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20, 20));

        int b=0;
        for(int i=0; i<20; i++)
        {
            for (int j=0; j<20; j++)
            {
                jp.add(new JButton("Button "+b));
                ++b;
            }
        }

        //Create and add scroll pane to the content pane.
        JScrollPane jsp = new JScrollPane(jp);
        jf.add(jsp, BorderLayout.CENTER);

        //Display the frame.
        jf.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run()
            {
                new JScrollPaneDemo();
            }
        });
    }
}
```

```
});  
}  
}
```

Output

JScrollPane Demo				
2	Button 63	Button 64	Button 65	Button 66
12	Button 83	Button 84	Button 85	Button 86
22	Button 103	Button 104	Button 105	Button 106
32	Button 123	Button 124	Button 125	Button 126
42	Button 143	Button 144	Button 145	Button 146
52	Button 163	Button 164	Button 165	Button 166
62	Button 183	Button 184	Button 185	Button 186
72	Button 203	Button 204	Button 205	Button 206
82	Button 223	Button 224	Button 225	Button 226
92	Button 243	Button 244	Button 245	Button 246
102	Button 263	Button 264	Button 265	Button 266
112	Button 283	Button 284	Button 285	Button 286
122	Button 303	Button 304	Button 305	Button 306
132	Button 323	Button 324	Button 325	Button 326