The java.awt package − Layout

25) Write code to implement listener classes and methods and in listener methods extract information from the event to determine the affected component mouse position nature and time of the event. State the event classname for any specified event listener interface in the java.awt.event package.

ActionListener :
User clicks a button, presses Return while typing in a text field, or chooses a menu item

AdjustmentListener :
User moved the Scrollbar

AWTEventListener :
event recorders for automated testing, and facilities such as the Java Accessibility package.

ComponentListener :
Component becomes visible, moved, resized

ContainerListener :
component is added to or removed from the container. These events are for notification only −− no container listener need be present for components to be successfully added or removed.

FocusListener :
Component gets the keyboard focus

InputMethodListener:
Used with text editing components

ItemListener :
User selected or deselected a checkbox

KeyListener :
User pressed or released a key

MouseListener :
User presses a mouse button while the cursor is over a component

MouseMotionListener :
User moves the mouse over a component

TextListener :
User changes a component's text.

WindowListener :
User closes or opens a frame (main window)

Listener Interfaces:

Interface	*Interface Methods*	*Add Listener*

ActionListener	actionPerformed(ActionEvent)	addActionListener()
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	addAdjustmentListener()
AWTEventListener	eventDispatched(AWTEvent)	addAWTEventListener()
ComponentListener	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)	addComponentListener()
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)	addContainerListener()
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	addFocusListener()
InputMethodListener	caretPositionChanged(InputMethodEvent) inputMethodTextChanged(InputMethodEvent)	addInputMethodListener()
ItemListener	itemStateChanged(ItemEvent)	addItemListener()
KeyListener	keyTyped(KeyEvent) keyPressed(KeyEvent keyReleased(KeyEvent)	addKeyListener()
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	addMouseListener()
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)	addMouseMotionListener()
TextListener	textValueChanged(TextEvent)	addTextListener()
WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) |windowOpened(WindowEvent)	addWindowListener()
Usage: class AL implements ActionListener	public void actionPerformed(ActionEvent ae)	getContentPane().add(btn); btn.addActionListener(AL);

Example without using Adapter class and using inner classes :

Most of the listener interfaces contain more than one method. For example, the MouseListener interface contains five methods: mousePressed, mouseReleased, mouseEntered, mouseExited, and mouseClicked. Even if you want only the mouse click to be checked, if your class directly implements MouseListener, then you must implement all five MouseListener methods. Also, note in the program how to find the affected component, mouse, position, nature and time of the event.

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

public class mousevt extends JApplet{
JButton b;
public void init() {
b=new JButton("Click");

class alinn implements ActionListener {
public void actionPerformed(ActionEvent ae) {
String command = ae.getActionCommand();
Object source = ae.getSource();//finds the affected component
if (source == b) {
System.out.println("text on the button was : " + command);
b.setText("You clicked");
}
}
}

class mlinn implements MouseListener {
public void mousePressed(MouseEvent me){
int evtid=me.getID(); //gets the event ID
System.out.println("id is : " + evtid);
long evtwhen=me.getWhen(); //gets the time of the event
System.out.println("timestamp is : " + evtwhen);
int x = me.getX(); //gets the mouse's X position
int y = me.getY(); //gets the mouse's X position
System.out.println("x " + x + "y " + y);
System.out.println("you Pressed!");
}
public void mouseClicked(MouseEvent me){
System.out.println("you Clicked!");
}
public void mouseExited(MouseEvent me){
System.out.println("you Exited!");
}
public void mouseEntered(MouseEvent me){
System.out.println("you Entered!");
}
public void mouseReleased(MouseEvent me){
System.out.println("you Released!");
}

}

class flinn implements FocusListener {
public void focusGained(FocusEvent fe){
System.out.println("Gained Focus!");
}
public void focusLost(FocusEvent fe){
System.out.println("Lost Focus!");
}
```

```
}

class klinn implements KeyListener {
public void keyPressed(KeyEvent ke){
System.out.println("Key Pressed at : ");
int x = me.getX();
int y = me.getY();
System.out.println("x " + x + "y " + y);
}
public void keyTyped(KeyEvent ke){
System.out.println("Key Typed!");
}
public void keyReleased(KeyEvent ke){
System.out.println("Key Released!");
}
}

class mmlinn implements MouseMotionListener {
public void mouseDragged(MouseEvent me){
System.out.println("Mouse Dragged!");
}
public void mouseMoved(MouseEvent me){
System.out.println("Mouse Moved!");
}
}

getContentPane().add(b);
b.addActionListener(new alinn());
b.addMouseListener(new mlinn());
b.addFocusListener(new flinn());
b.addKeyListener(new klinn());
}
}
```

Example using Adapter class :

To avoid cluttering your code with empty method bodies, you have an adapter class for each listener interface which has more than one method.
An example of extending an adapter class instead of directly implementing a listener interface:

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

public class mouseEvt extends JApplet {
JButton b;
public void init() {
b=new JButton("Click");
getContentPane().add(b);
b.addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent e) {
System.out.println("Pressed");
```

```
}
} );
}
}
```

Remember :

Event.target Object Points to the component that generated the event
This indicates the component over which the event occurred or with which the event is associated. This object has been replaced by AWTEvent.getSource()

Event.when long Timestamp of the event. Replaced by InputEvent.getWhen().
getX() − int − X coordinate of mouse − MouseEvent
getY() − int − Y coordinate of mouse − MouseEvent
getId() − long − ID Number of the event − AWTEvent
getWhen() − long − Time stamp of the event − InputEvent

---

26) Write code using component container and layout manager classes of the java.awt package to present a GUI with specified appearance and resize the behavior and distinguish the responsibilities of layout managers from those of containers.

Components :

A Component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes.
Some of the methods in Components are :

getSize() :

- argument : Dimension
- returns size of a Component.
- The return type is Dimension, which has public data members height and width.

setForeground() and setBackground() :

- argument : instance of java.awt.Color
- sets the foreground and background of a component.
- If a component's foreground or background color is not explicitly set, the component uses the foreground and background color of its immediate container.
- for eg, if an applet's foreground is black and background is gray, when you add a component such as, a button to that applet without calling the setForeground() and setBackground() methods on the button, then the button's label will be black on gray.

setFont() :

- argument : instance of java.awt.Font
- Determines the Component's text font.
- If setFont() is not called explicitly on a component, it uses the font of its container.

setEnabled() :

- argument : boolean
- If the argument is tru the component has normal appearance.
- If false, the component will be grayed out and doesn't respond to user input.

setSize() and setBounds() :

- setSize() arguments : width and height and setBounds() arguments : four int arguments x,y widht and height

setVisible() :

- argument : boolean
- If the argument is true, the component will be visible
- If false, the component will not be seen on the screen.

Components :

Button :

- sends : ActionEvent

Canvas :

- Component that has no default appearance or behavior
- can subclass Canvas to create custom drawing regions, work areas, components and so on.
- The default size (preferred size) of canvas will be too small.
- So, you have to use a layout manager that will resize the canvas or you have to call setSize() on Canvas.
- sends : MouseMotion and KeyEvents

CheckBox :

- default is false
- sends : ItemEvents when they are selected

Choice :

- presents a pop−up menu of choices.
- The current choice is displayed as the title of the menu.
- sends : Item Events

FileDialog :

- The FileDialog class displays a dialog window from which the user can select a file.
- It is a modal dialog.

Label :

- A label does not react to input events. So, it cannot get the keyboard focus.
- Label can display text, image, or both.
- Default alignment is left.

List :

- Collection of text items, arranged vertically.
- If there are more items than it can display, it acquires vertical scroll bar.
- If number of visible rows are not specified the height will be determined by a layout manager.
- By default, supports single selection
- Selecting an item : sends ItemEvent
- double−clicking an item sends an ActionEvent.

ScrollPane :

- A specialized container that manages a viewport, optional vertical and horizontal scrollbars, and optional row and column heading viewports.
- can contain a single component

- sends MouseEvent and MouseMotionEvent

ScrollBar :

- The user positions the slider in the scrollbar to determine the contents of the viewing area.
- calculating the slider size : If the horizontal scrollbar's minimum value is 600 and maximum value is 700 then, 700−600 = 100 and 10/100=1/10th
- So, the slider would be one−tenth the width of the scrollbar.

TextField and TextArea :

- The number of columns is the width, in terms of columns of text as rendered in a particular font.
- A 25−column textarea with a tiny font will be very narrow, while a 5−column text area with a huge font will be extremely wide.
- The column width for a fixed−width font is determinable
- The column width for proportion font is the average of all the font's character widths.
- Text areas support scroll bars, if the user types beyond the right most column.
- Text fields can be scrolled by left and right arrow keys.
- both sends : Text Events.
- Additionally TextField generates ActionEvent when a Enter Key is pressed

Applet :

- Changing size of an applet is permitted or forbidden by the browser on which you run the applet. So appletviewer is the best thing to use.

Frame :

- When constructed, it has no size. So should call setSize() or setBounds(), and then setVisible(true)
- When you are done with a frame, you have to recycle its non−memory resources.
- For releasing the non−memory resource of a frame, you have to call its dispose() method.

Panel :

- Applets and Frames are top−level GUI components, where Panels are intermediate level.
- You can add all components of a GUI directly to an applet or a frame.
- You can provide additional level of grouping in panels by adding components to panels and adding panels to top−level applet or frame.
- You can add a panels to a Panel.

Dialog :

- is a pop−up window which accepts user input.
- It can be modal.
- It is the superclass of FileDialog class
- Default layout manager for this class is borderlayout.

Menu :

- a popup window containing MenuItems that is displayed when the user selects an item on the MenuBar.
- a Menu can also contain Separators.
- MenuBar may appear only in Frames and therefore, Menus also.
- To create a frame with menubar, a) create a menubar and attach it to the frame, b) create and populate the menu, c) attach the menu to the menubar
- menu can have 4 kinds of elements − MenuItem, CheckBoxMenuItem, Separator, Menu
- MenuItems sends ActionEvents
- CheckBoxMenuItem sends ItemEvents.

Layout Manager :

Every Java component has a preferred size. The preferred size express how big the component would like to be, barring conflict with a layout manager. Preferred size is generally the smallest size necessary to render the component in a visually meaningful way. Preferred size is platform dependent. The Layout manager overrules the component's preferred size.

Flow Layout Manager	Grid Layout Manager	Border Layout Manager
Default for Panels and Applets		Default for Frames
Arranges components in horizontal rows within every row, the components are evenly spaced, and cluster of components are centered You can specify the direction by passing arguments with FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT	subdivides its territory into a matrix of rows and columns. The number of rows and columns are specified as parameters	Divides its territory into 5 regions as North, South, East, West, Center. Each region may contain a single component
Always honors component's preferred size	Always ignores component's preferred size	something inbetween

setLayout(new FlowLayout(FlowLayout.RIGHT));	setLayout(new GridLayout(2,2));	TextField t = new extField();
		Button b = new Button();
		setLayout(new BorderLayout());
		add(b);
		add(t);

if the applet's size is small the layout manager will adjust components in the next row	if no arguments given, it takes the full frame and displays the components.All components within the frame are the same width and height.	The components gets resized according to the applet's size. The above example will show only one component. so, it should be,
		add(t, BorderLayout.NORTH);
		add(b, BorderLayout.SOUTH);

GridBagLayout : The GridBagLayout can be used to layout components in a grid. Unlike the GridLayout, the sizes of the grids do not need to be constant, and a component can occupy more (or less) than one row or column.

Some of the information that the GridBagLayout needs to know about an object are:

- row and column
- number of cells spanned
- placement within its space
- stretch and shrink values

- Each component is associated with one area which may be one or more grid rectangles.
- The number of rows and columns in the grid is determined by the largest values of gridx and gridy.
- The anchor attribute indicates where in the grid the component will appear in its area if it does not exactly fill its area.
- The fill attribute determines whether the components should be stretched to fill the entire area.
- The weight attributes determine the various sizes of the grid areas.

Recipe for making a GridBagLayout :

- Sketch out the component layout on a piece of paper.
- Divide the paper to rows and columns according to the number of components you need to place
- place small components in one cell and larger components spaning multiple cells
- Label the rows and columns with 0,1,2,3.....
- Now you can read the gridx, gridy, gridwidth and gridheight values
- For each component ask yourself whether it needs to fill its cell horizontally or vertically, if not, how do you want to align? tells you the fill and anchor parameters
- If you want a particular row or column to always stay at its default size, set weightx and weighty to 0 in all components that belong to that row or column, otherwise set it to 100.

These informations are stored in an object of type GridBagContstraints and is associated with a component using "setContraints(Component, GridBagContraints)" This causes the layout manager to make a copy of the constraints and associate them with the object. Therefore you only need one of these GridBagContraints objects. The following is a complete list of all of the constraints:

- anchor determines position in the display area
- fill determines if a component is stretched to fill the area
- gridheight and gridwidth determine the number of rows and columns in the component's area
- gridx and gridy determine the position of the component's area.
- insets determine a border around a component's area.
- ipadx and ipady allows the minimum or preferred size of a component to be adjusted.
- weightx and weighty determine the sizes of the rows and columns of the grids.

Weights :

The weights determine how the extra space is distributed after the components are laid out using their preferred sizes.
The algorithm for doing this is simple in the case in which all components have gridwidth and gridheight equal to 1. In this case the weight of a grid column or row is the maximal weight of any component in that column or row.
The extra space is divided based on these weights.
For example, if the total weight of all of the columns is 10, then a column with weight 4 would get 4/10 of the total extra space.
If there are components which span more than one row or column, the weights are first determined by those components spanning only one row and column.
Then each additional component is handled in the order it was added. If its weight is less than the total weights of the rows or columns it spans, it has no effect.
Otherwise, its extra weight is distributed based on the weights of those rows or columns it spans.
If the extra space is negative, the layout manager squeezes things and you have not control over how this is done.

Anchors

There are nine anchor types specifying 8 basic compass positions and the center position. NORTH, SOUTH, EAST, WEST, NORTHWEST, NORTHEAST, SOUTHWEST, SOUTHEAST, CENTER

Fill

The four fill types are NONE (the default), VERTICAL, HORIZONTAL, and BOTH. They affect components whose preferred size does not completely fill their grid area.

gridx, gridy

The gridx and gridy fields determine the positioning of the components. The default is to put the object in the position after the last one which was added.

Internal Padding

The fields ipadx and ipady can be used to modify the minimum or preferred size of a component. For a button, the preferred width is determined by the length of its label and the preferred height is determined by the font used for the label.

gridwidth and gridheight

The gridwidth and gridheight fields determine the number of cells that are included in a component's area.
The possible values are a (small) positive integer, or one of the special values REMAINDER or RELATIVE.
REMAINDER means the component will occupy all of the cells to the right or below it.
RELATIVE means the component will occupy all of the cells to the right or below it except for the last cell.
By default, a cell is placed right after the previous one.
By setting a width or height to REMAINDER, the component will occupy all cells to the right or below it.
By setting the width or height to Relative, the component will occupy all cells to the right or below, except for the last one.

Insets

The insets field has four subfields, top, bottom, left, and right.
Each controls the size of the border around the component. The defaults are all zero.
A negative inset allows a component to extend outside its area.

Procedure to create GridBagLayout :

- Create an Object of type GridBagLayout (without specifying the number of rows and columns)
- Set this GridBagLayout Object to be the layout manager for the component.
- Create an object of type GridBagConstraints (will specify how the components are laid out within the gridbag)
- For each component, fill in the GridBagConstraints object and call the setConstraints object to pass this information to GridBagLayout.
- Finally, add the components

eg,

```
GridBagLayout layout = new GridBagLayout();
panel.setLayout(layout);
GridBagConstraints constraints = new
GridBagConstraints();
//if set to 0 the area never grows
//beyond its initial size in that direction, if window is
resized.
constraints.weightx=100;
constraints.weighty=100;
//specifies the column and row positions of the upper left
//corner of the component to be added.
constraints.gridx=0;
constraints.gridy=0;
//determines how many columns and rows it occupies.
constraints.gridwidth=1;
```

```
constraints.gridheight=3;
List l=new List(4);
layout.setConstraints(l, constraints);
panel.add(l);
```

An alternate method for gridx, gridy, gridwidth, gridheight :

- Instead of setting the gridx and gridy you set as GridBagConstraints.RELATIVE
- Then add components to the gridbaglayout in a standardized order going from left to right in the first row, then moving along the next row and so on.
- GridBagConstraints.REMAINDER tells the component is the last one in its row. (instead of gridwidth and gridheight)only if the component extends to the last row or column REMAINER can be used.

Remember :

Layout managers and container size :

- The BorderLayout tells its enclosing container the size to allow for each control by invoking the preferredSize() methods of each control.
- GridBagLayout and GridLayout force the components to adjust their size according to the actual dimensions of the container.
- FlowLayout doesn't change the sizes of contained components at all

GridBag :

- If you don't want a component to stretch out and fill the entire area, you need to set fill
- If the component doesn't fill the entire area, you can specify where in the area you want it by setting the anchor field.
