**UNIT IV EVENT HANDLING**

**Event:**

An event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a GUI. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

**Foreground Events -** Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

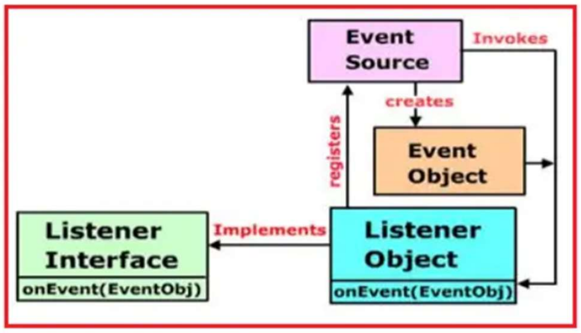
**Background Events -** Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

**Event Handling:** is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs.

**Event Handling Mechanisms:**

1. Hierarchical Event Handling Model
2. Delegation Event Model

**THE DELEGATION EVENT MODEL:**



The modern approach to handling events is predicated on the delegation event model, which defines standard and consistent mechanisms to get and process events. Its concept is sort of simple: a source generates an occasion and sends it to at least one or more listeners. In this scheme, the listener simply waits until it receives an occasion. Once an occasion 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.

**Components of Event Handling**
**Events**

In the delegation model, an event is an object that describes a state change in a source.

Among other causes, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface.

Some of the activities that cause events to be generated are 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.

**Event Sources**

A source is an object that generates an event. Generally sources are components. 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:

**public void addTypeListener (TypeListener el )**

Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ).

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: public void removeTypeListener(TypeListener el )

**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.

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 receive and process one or both of these events if it provides an implementation of this interface.

**Sources of Events:**

| **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; |
| 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 Classes and Listener Interfaces:**

The classes that represent events are at the core of Java's event handling mechanism. Widely used Event classes are those defined by AWT and those defined by Swing.

The java.awt.event package provides many event classes and Listener interfaces for event handling. At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the super class for all events. It's one constructor is shown here:

EventObject(Object src) - Here, src is the object that generates this event.

The package java.awt.event defines many types of events that are generated by various user interface elements

| EVENT CLASS | DESCRIPTION | LISTENER INTERFACE |
|---|---|---|
| ActionEvent | Generated when a button is pressed, a list item is double-clicked, or a menu item is selected. | ActionListener |
| AdjustmentEvent | Generated when a scroll bar is manipulated. | AdjustmentListener |
| ComponentEvent | Generated when a component is hidden, moved, resized, or becomes visible. | ComponentListener |
| ContainerEvent | Generated when a component is added to or removed from a container. | ContainerListener |
| FocusEvent | Generated when a component gains or losses keyboard focus. | FocusListener |
| ItemEvent | Generated when a check box or list item is clicked | ItemListener |
| KeyEvent | Generated when input is received from the keyboard. | KeyListener |
| MouseEvent | Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component. | MouseListener and MouseMotionListener |
| TextEvent | Generated when the value of a text area or text field is changed. | TextListener |
| WindowEvent | Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. | WindowListener |

**The 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 (Ex. Escape), and SHIFT_MASK.

ActionEvent has these three constructors:

- ActionEvent(Object src, int type, String cmd)
- ActionEvent(Object src, int type, String cmd, int modifiers)
- ActionEvent(Object src, int type, String cmd, long when, int modifiers)

**The AdjustmentEvent Class:**

An AdjustmentEvent is generated by a scroll bar. There are five types of adjustment events.

| | |
|---|---|
| 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. |

**The 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:

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

ComponentEvent is the superclass either directly or indirectly of ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent, among others.

**The 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 constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED.

**The 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 InputEvent Class:**

The abstract class InputEvent is a subclass of ComponentEvent and is the super class for component input events. Its subclasses are KeyEvent and MouseEvent.

**The 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.

**The 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_RELEASED | The mouse was released. |
| MOUSE_WHEEL | The mouse wheel was moved. |

**The 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 WindowEvent Class:**

The WindowEvent class defines integer constants that can be used to identify different types of events:

| | |
|---|---|
| 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 was iconified. |
| WINDOW_ICONIFIED | The window gained input focus. |
| WINDOW_LOST_FOCUS | The window lost input focus. |
| WINDOW_OPENED | The window was opened. |

**EventListener Interfaces:**

An event listener registers with an event source to receive notifications about the events of a particular type. Various event listener interfaces defined in the java.awt.event package are given below:

| **INTERFACE** | **DESCRIPTION** |
|---|---|
| ActionListener | Defines the actionPerformed() method to receive and process action events.<br><br>void actionPerformed(ActionEvent ae) |
| MouseListener | Defines five methods to receive mouse events, such as when a mouse is clicked, pressed, released, enters, or exits a component void mouseClicked(MouseEvent me)<br><br>void mouseEntered(MouseEvent me) void mouseExited(MouseEvent me) void mousePressed(MouseEvent me) void mouseReleased(MouseEvent me) |
| MouseMotionListener | Defines two methods to receive events, such as when a mouse is dragged or moved.<br><br>void mouseDragged(MouseEvent me) void mouseMoved(MouseEvent me) |
| AdjustmentListner | Defines the adjustmentValueChanged() method to receive and process the adjustment events.<br><br>void adjustmentValueChanged(AdjustmentEvent ae) |

| | |
|---|---|
| TextListener | Defines the textValueChanged() method to receive and process an event when the text value changes.<br><br>void textValueChanged(TextEvent te) |
| WindowListener | Defines seven window methods to receive events.<br><br>void windowActivated(WindowEvent we) void windowClosed(WindowEvent we) void windowClosing(WindowEvent we)<br><br>void windowDeactivated(WindowEvent we) void windowDeiconified(WindowEvent we) void windowIconified(WindowEvent we) void windowOpened(WindowEvent we) |
| ItemListener | Defines the itemStateChanged() method when an item has been<br><br>void itemStateChanged(ItemEvent ie) |
| WindowFocusListener | 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:<br><br>void windowGainedFocus(WindowEvent we) void windowLostFocus(WindowEvent we) |
| ComponentListener | This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:<br><br>void componentResized(ComponentEvent ce) void componentMoved(ComponentEvent ce)<br><br>void componentShown(ComponentEvent ce) void componentHidden(ComponentEvent ce) |

**The ActionListener Interface**

This interface defines the actionPerformed( ) method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent ae)

**The AdjustmentListener Interface**

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

void adjustmentValueChanged(AdjustmentEvent ae)

**The ComponentListener Interface**

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

void componentResized(ComponentEvent ce)

void componentMoved(ComponentEvent ce)

void componentShown(ComponentEvent ce)

void componentHidden(ComponentEvent ce)

**The ContainerListener Interface**

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:

void componentAdded(ContainerEvent ce)

void componentRemoved(ContainerEvent ce)

**The FocusListener Interface**

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:

void focusGained(FocusEvent fe) void focusLost(FocusEvent fe)

**The ItemListener Interface**

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

void itemStateChanged(ItemEvent ie)

**The KeyListener Interface**

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:

void keyPressed(KeyEvent ke)

void keyReleased(KeyEvent ke)

void keyTyped(KeyEvent ke)

**The MouseListener Interface**

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:

void mouseClicked(MouseEvent me)

void mouseEntered(MouseEvent me)

void mouseExited(MouseEvent me)

void mousePressed(MouseEvent me)

void mouseReleased(MouseEvent me)

**The MouseMotionListener Interface**

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:

void mouseDragged(MouseEvent me)

void mouseMoved(MouseEvent me)

**The MouseWheelListener Interface**

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

void mouseWheelMoved(MouseWheelEvent mwe)

**The TextListener Interface**

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

void textChanged(TextEvent te)

**The WindowFocusListener Interface**

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:

void windowGainedFocus(WindowEvent we)

void windowLostFocus(WindowEvent we)

**The WindowListener Interface**

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

void windowActivated(WindowEvent we)

void windowClosed(WindowEvent we)

void windowClosing(WindowEvent we)

void windowDeactivated(WindowEvent we)

void windowDeiconified(WindowEvent we)

void windowIconified(WindowEvent we)

void windowOpened(WindowEvent we)

**Handling Mouse Events**

To handle mouse events, you must implement the MouseListener and the MouseMotionListener interfaces

Mouse event occurs when a mouse related activity is performed on a component such as clicking, dragging, pressing, moving or releasing a mouse etc. Objects representing mouse events are created from MouseEvent class.

The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. The two variables, mouseX and mouseY, 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.

**Methods of MouseListener interface:**

```
public abstract void mouseClicked(MouseEvent e);

public abstract void mouseEntered(MouseEvent e);

public abstract void mouseExited(MouseEvent e);

public abstract void mousePressed(MouseEvent e);

public abstract void mouseReleased(MouseEvent e);
```

**Methods of MouseMotionListener interface:**

```
public abstract void mouseDragged(MouseEvent e);

public abstract void mouseMoved(MouseEvent e);
```

**Handling Mouse Events Example Program:**

```
import java.awt.*;

import javax.swing.*;

import java.awt.event.*;

public class MyMouse extends JFrame implements MouseListener

{

 JLabel label;

 MyMouse(){

  addMouseListener(this);

  label = new JLabel();

  label.setBounds(90,80,130,20);

  label.setFont(new Font("Serif", Font.BOLD, 20));
  add(label);
  setSize(250,250);
  setLayout(null);
  setVisible(true);
 }
 public void mouseClicked(MouseEvent e) {
  label.setText("Clicked");
}
 public void mouseEntered(MouseEvent e) {
  label.setText("Entered");
 }
 public void mouseExited(MouseEvent e) {
  label.setText("Exited");
 }
 public void mousePressed(MouseEvent e) {
  label.setText("Pressed");
 }
 public void mouseReleased(MouseEvent e) {
  label.setText("Released");
 }
 public static void main(String[] args) {
  new MyMouse();
 }}
```

**Handling Key Board Events:**

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package.

It is a subclass of the abstract InputEvent class and is generated when the user presses or releases a key

is that you will be implementing the KeyListener interface.

**Methods of KeyListener interface**

```
public abstract void keyPressed(KeyEvent e);

public abstract void keyReleased(KeyEvent e);

 public abstract void keyTyped(KeyEvent e);
```

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.

**Example:**

```
import java.awt.*;
 import java.awt.event.*;
class MyFrame extends Frame implements KeyListener
{
String keystate="Hello PVPSIT"; String msg="";
MyFrame()
{
addKeyListener(this); addWindowListener(new MyWindow() );
}
public void keyPressed(KeyEvent ke)
{
keystate="Key Pressed"; msg+=ke.getKeyText( ke.getKeyCode()); repaint();
}
public void keyTyped(KeyEvent ke)
{
keystate="Key Typed"; repaint(); msg=msg+ke.getKeyChar();
}
public void keyReleased(KeyEvent ke)
{
keystate="Key Released"; repaint();
}
public void paint(Graphics g)
{
g.drawString(keystate,100,50); g.drawString(msg,100,100);
}
}
class KeyEventsExample
{
public static void main(String arg[])
{
MyFrame f=new MyFrame(); f.setTitle("Key Events Example"); f.setSize(500,300);
f.setVisible(true);
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent we)
{
System.exit(0);
}
});
}}
```

## SUMMARY ABOUT EVENT HANDLING

Event handling has three main components:

**Events :** An event is a change in state of an object.

**Events Source :** Event source is an object that generates an event.

**Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

**Action:** What user does is known as action. Example, a click over button. Here, click is the action performed by the user.

**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 you want to receive and process only some of the events that are handled by a particular event listener interface.

**For example**

| **MouseListener** | **MouseAdapter** |
|---|---|
| void mouseClicked(MouseEvent me) | void mouseClicked(MouseEvent me){ } |
| void mouseEntered(MouseEvent me) | void mouseEntered(MouseEvent me) { } |
| void mouseExited(MouseEvent me) | void mouseExited(MouseEvent me) { } |
| void mousePressed(MouseEvent me) | void mousePressed(MouseEvent me) { } |
| void mouseReleased(MouseEvent me) | void mouseReleased(MouseEvent me) { } |

| **Adapter Class** | **Listener Interface** |
|---|---|
| ComponentAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| FocusAdapter | FocusListener |
| KeyAdapter | KeyListener |
| MouseAdapter | MouseListener |
| MouseMotionAdapter | MouseMotionListener |
| WindowAdapter | WindowListener |

**Example Program**

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

class WindowEx1 extends Frame

{

String msg;

WindowEx1(){

addWindowListener(new WA());

}

}

class WA extends WindowAdapter

{

public void windowClosing(WindowEvent we)

{
```

```
System.exit(0);
}}
class WindowAdapterEx
{
public static void main(String[] args) { WindowEx1 w=new WindowEx1(); w.setSize(400,400);
w.setLayout(null);
w.setVisible(true);
}}
```

**JAVA ABSTRACT WINDOW TOOLKIT(AWT)**

Java AWT is an API that contains large number of classes and methods to create and manage graphical user interface ( GUI ) applications.

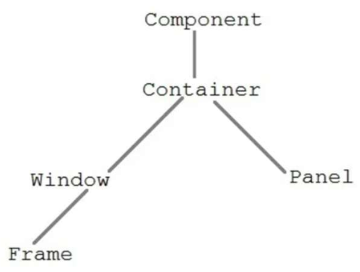
Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java. Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.

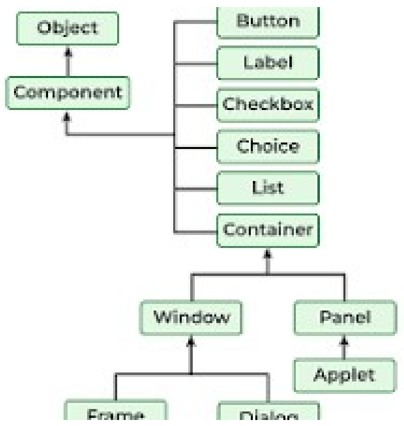
AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

**Java AWT Hierarchy**

The hierarchy of Java AWT classes are given below, all the classes are available in java.awt package.





**Component class**

Component class is at the top of AWT hierarchy. It is an abstract class that encapsulates all the attributes of visual component. A component object is responsible for remembering the current foreground and background colors and the currently selected text font.

**Container**

Container The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

**Window**

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

**Panel**

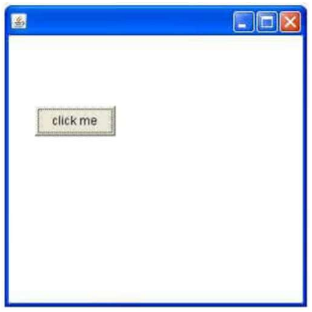
The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

**Frame**

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

**AWT Example**

```
import java.awt.*;

class First extends Frame

{

First()

{

Button b=new Button("click me");

b.setBounds(30,100,80,30);      // setting button position

add(b);                         //adding button into frame

setSize(300,300);               //frame size 300 width and 300 height

setLayout(null);                //no layout manager

setVisible(true);               //now frame will be visible, by default not visible

}

public static void main(String args[])

{

First f=new First();

}}
```



**Examples of GUI based Applications**

- ➢ Automated Teller Machine (ATM)
- ➢ Airline Ticketing System
- ➢ Information Kiosks at railway stations
- ➢ Mobile Applications
- ➢ Navigation Systems

**DIFFERENCES BETWEEN AWT AND SWINGS**

| AWT | Swing |
|---|---|
| AWT components are heavyweight components | Swing components are lightweight components |
| AWT doesn't support pluggable look and feel | Swing supports pluggable look and feel |
| AWT programs are not portable | Swing programs are portable |
| AWT is old framework for creating GUIs | Swing is new framework for creating GUIs |
| AWT components require java.awt package | Swing components require javax.swing package |
| AWT supports limited number of GUI controls | Swing provides advanced GUI controls like Jtable, JTabbedPane etc |
| More code is needed to implement AWT controls functionality | Less code is needed to implement swing controls functionality |
| AWT doesn't follow MVC | Swing follows MVC |

**Graphical User Interface (GUI) COMPONENTS:**

**Label**

The object of the Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by a programmer but a user cannot edit it directly.

It is called a passive control as it does not create any event when it is accessed. To create a label, we need to create the object of Label class.

**Label Class Declaration**

public class Label extends Component implements Accessible

Label Fields

The java.awt.Component class has following fields:

static int LEFT: It specifies that the label should be left justified.

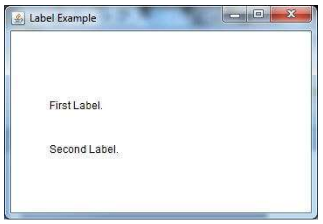
static int RIGHT: It specifies that the label should be right justified.

static int CENTER: It specifies that the label should be placed in center.

**Example**

```
import java.awt.*;
  public class LabelExample {
public static void main(String args[]){
      // creating the object of Frame class and Label class
    Frame f = new Frame ("Label example");
    Label l1, l2;
      // initializing the labels
    l1 = new Label ("First Label.");
    l2 = new Label ("Second Label.");
      // set the location of label
    l1.setBounds(50, 100, 100, 30);
    l2.setBounds(50, 150, 100, 30);
      // adding labels to the frame
    f.add(l1);
    f.add(l2);
      // setting size, layout and visibility of frame
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}    }
```

Output:



First Label.

Second Label.

**Button**

A button is basically a control component with a label that generates an event when pushed. The Button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

When we press a button and release it, AWT sends an instance of ActionEvent to that button by calling processEvent on the button. The processEvent method of the button receives the all the events, then it passes an action event by calling its own method processActionEvent. This method passes the action event on to action listeners that are interested in the action events generated by the button.

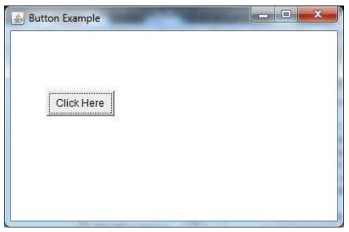
**Class Declaration**

public class Button extends Component implements Accessible

Example

```
import java.awt.*;

public class ButtonExample {

public static void main (String[] args) {

   // create instance of frame with the label

  Frame f = new Frame("Button Example");

   // create instance of button with label

  Button b = new Button("Click Here");

   // set the position for the button in frame

  b.setBounds(50,100,80,30);

   // add button to the frame

  f.add(b);

  // set size, layout and visibility of frame

  f.setSize(400,400);

  f.setLayout(null);

  f.setVisible(true);

}   }
```

**Output:**



**Canvas**

The Canvas class controls and 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

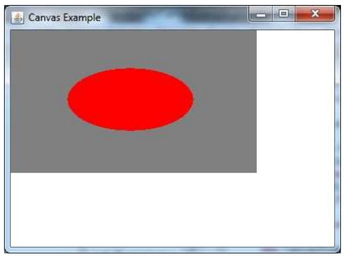
public class Canvas extends Component implements Accessible

**Example**

```
// importing awt class

import java.awt.*;

// class to construct a frame and containing main method

public class CanvasExample

{

  // class constructor

  public CanvasExample()

  {

    // creating a frame

    Frame f = new Frame("Canvas Example");

    // adding canvas to frame

    f.add(new MyCanvas());

    // setting layout, size and visibility of frame

    f.setLayout(null);

    f.setSize(400, 400);

    f.setVisible(true);

  }

  // main method

  public static void main(String args[])

  {

    new CanvasExample();

  }    }

  // class which inherits the Canvas class

// to create Canvas

class MyCanvas extends Canvas

{           // class constructor

        public MyCanvas() {
```

```
        setBackground (Color.GRAY);
        setSize(300, 200);
      }
       // paint() method to draw inside the canvas
  public void paint(Graphics g)
  {
      // adding specifications
    g.setColor(Color.red);
    g.fillOval(75, 75, 150, 75);
  }
}
```

**Output:**



**Scrollbar**

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

It can be added to top-level container like Frame or a component like Panel. The Scrollbar class extends the Component class.

**Class Declaration**

public class Scrollbar extends Component implements Adjustable, Accessible

**Scrollbar Class Fields**

static int HORIZONTAL - It is a constant to indicate a horizontal scroll bar.

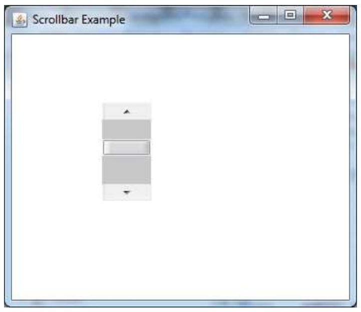
static int VERTICAL - It is a constant to indicate a vertical scroll bar.

**Example**

```
// importing awt package
import java.awt.*;
  public class ScrollbarExample1 {
```

```
// class constructor
ScrollbarExample1() {
     // creating a frame
       Frame f = new Frame("Scrollbar Example");
    // creating a scroll bar
       Scrollbar s = new Scrollbar();
     // setting the position of scroll bar
       s.setBounds (100, 100, 50, 100);
     // adding scroll bar to the frame
       f.add(s);
     // setting size, layout and visibility of frame
       f.setSize(400, 400);
       f.setLayout(null);
       f.setVisible(true);
}
 // main method
public static void main(String args[]) {
    new ScrollbarExample1();
}
}
```

**Output:**



**Text Field**

The object of a Text Field class is a text component that allows a user to enter a single line text and edit it. It inherits Text Component class, which further inherits Component class.

When we enter a key in the text field (like key pressed, key released or key typed), the event is sent to Text Field. Then the Key Event is passed to the registered Key Listener. It can also be done using Action Event; if the Action Event is enabled on the text field, then the Action Event may be fired by pressing return key. The event is handled by the Action Listener interface.

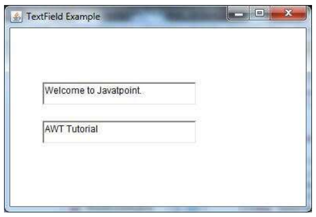
**Class Declaration**

public class TextField extends TextComponent

**Example**

```
// importing AWT class
import java.awt.*;
public class TextFieldExample1 {
  // main method
  public static void main(String args[]) {
  // creating a frame
  Frame f = new Frame("TextField Example");
    // creating objects of textfield
  TextField t1, t2;
  // instantiating the textfield objects
  // setting the location of those objects in the frame
  t1 = new TextField("Welcome to Javatpoint.");
  t1.setBounds(50, 100, 200, 30);
  t2 = new TextField("AWT Tutorial");
  t2.setBounds(50, 150, 200, 30);
  // adding the components to frame
  f.add(t1);
  f.add(t2);
  // setting size, layout and visibility of frame
  f.setSize(400,400);
  f.setLayout(null);
  f.setVisible(true);
}
}
```

**Output:**



**TextArea**

The object of a TextArea class is a multiline region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

The text area allows us to type as much text as we want. When the text in the text area becomes larger than the viewable area, the scroll bar appears automatically which helps us to scroll the text up and down, or right and left.

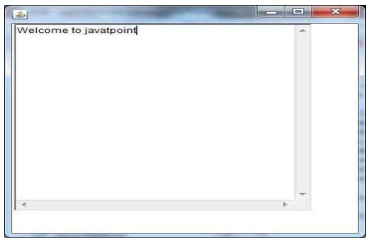
**Class Declaration**

public class TextArea extends TextComponent

**Example**

```
//importing AWT class

import java.awt.*;

public class TextAreaExample

{

// constructor to initialize

    TextAreaExample() {

// creating a frame

      Frame f = new Frame();

// creating a text area

         TextArea area = new TextArea("Welcome to javatpoint");

// setting location of text area in frame

      area.setBounds(10, 30, 300, 300);

// adding text area to frame

      f.add(area);

// setting size, layout and visibility of frame

      f.setSize(400, 400);

      f.setLayout(null);

      f.setVisible(true);

    }

// main method

public static void main(String args[])

{

  new TextAreaExample();

}
}
```

**Output:**



```
Welcome to javatpoint
```

**Checkbox**

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

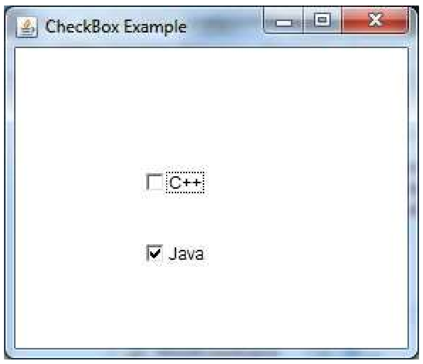
**Class Declaration**

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

**Example**

```
// importing AWT class
import java.awt.*;
public class CheckboxExample1
{
// constructor to initialize
    CheckboxExample1() {
// creating the frame with the title
      Frame f = new Frame("Checkbox Example");
// creating the checkboxes
        Checkbox checkbox1 = new Checkbox("C++");
        checkbox1.setBounds(100, 100,  50, 50);
        Checkbox checkbox2 = new Checkbox("Java", true);
// setting location of checkbox in frame
checkbox2.setBounds(100, 150,  50, 50);
// adding checkboxes to frame
        f.add(checkbox1);
        f.add(checkbox2);
  // setting size, layout and visibility of frame
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
// main method
public static void main (String args[])
{
    new CheckboxExample1();
}    }
```

**Output:**



**CheckboxGroup**

The object of CheckboxGroup class is used to group together a set of Checkbox. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the object class.

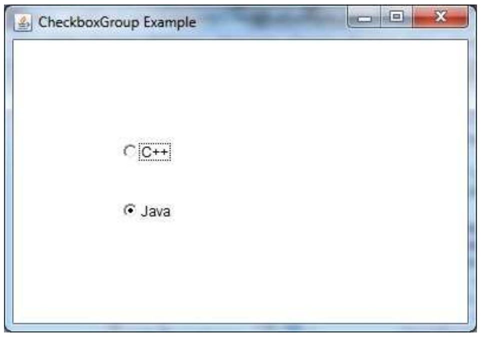
**Class Declaration**

public class CheckboxGroup extends Object implements Serializable

**Example**

```
import java.awt.*;
public class CheckboxGroupExample
{
       CheckboxGroupExample(){
       Frame f= new Frame("CheckboxGroup Example");
        CheckboxGroup cbg = new CheckboxGroup();
        Checkbox checkBox1 = new Checkbox("C++", cbg, false);
        checkBox1.setBounds(100,100, 50,50);
        Checkbox checkBox2 = new Checkbox("Java", cbg, true);
        checkBox2.setBounds(100,150, 50,50);
        f.add(checkBox1);
        f.add(checkBox2);

        f.setSize(400,400);

        f.setLayout(null);

        f.setVisible(true);

     }

public static void main(String args[])

{

   new CheckboxGroupExample();

}    }
```

**Output:**



**Choice**

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

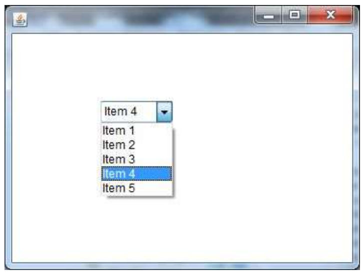
**Class Declaration**

public class Choice extends Component implements ItemSelectable, Accessible

**Example**

```
// importing awt class

import java.awt.*;

public class ChoiceExample1 {

          // class constructor

        ChoiceExample1() {

          // creating a frame

        Frame f = new Frame();

          // creating a choice component

        Choice c = new Choice();

        // setting the bounds of choice menu
        c.setBounds(100, 100, 75, 75);
          // adding items to the choice menu
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
          // adding choice menu to frame
        f.add(c);
          // setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
     }
  // main method
public static void main(String args[])
{
   new ChoiceExample1();
}
}
```

**Output:**



**List**

The object of List class represents a list of text items. With the help of the List class, user can choose either one item or multiple items. It inherits the Component class.

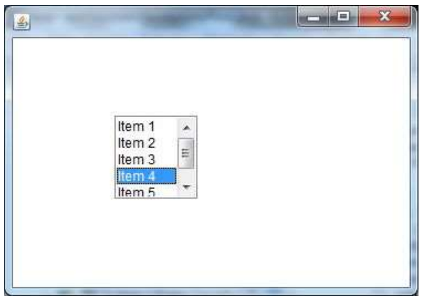
**class Declaration**

public class List extends Component implements ItemSelectable, Accessible

**Example**

```
// importing awt class
import java.awt.*;
  public class ListExample1
{
    // class constructor
     ListExample1() {
     // creating the frame
        Frame f = new Frame();
       // creating the list of 5 rows
        List l1 = new List(5);
          // setting the position of list component
        l1.setBounds(100, 100, 75, 75);
          // adding list items into the list
        l1.add("Item 1");
        l1.add("Item 2");
        l1.add("Item 3");
        l1.add("Item 4");
        l1.add("Item 5");
          // adding the list to frame
        f.add(l1);
          // setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
     }

// main method
public static void main(String args[])
{
   new ListExample1();
}
}
```

**Output:**



**Panel**

The Panel is a simplest container class. It provides space in which an application can attach any other component. It inherits the Container class.

It doesn't have title bar.

**class declaration**

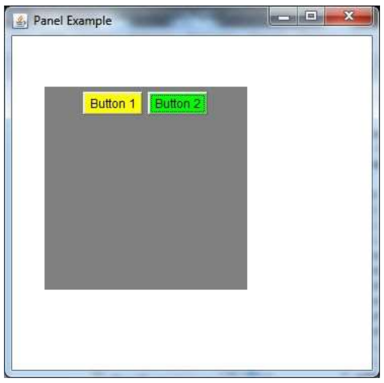
public class Panel extends Container implements Accessible

**Example**

```
import java.awt.*;

public class PanelExample {

    PanelExample()

      {

      Frame f= new Frame("Panel Example");

      Panel panel=new Panel();

      panel.setBounds(40,80,200,200);

      panel.setBackground(Color.gray);

      Button b1=new Button("Button 1");

      b1.setBounds(50,100,80,30);

      b1.setBackground(Color.yellow);

      Button b2=new Button("Button 2");

      b2.setBounds(100,100,80,30);

      b2.setBackground(Color.green);

      panel.add(b1); panel.add(b2);

      f.add(panel);

      f.setSize(400,400);

      f.setLayout(null);
```

```
        f.setVisible(true);
        }
        public static void main(String args[])
        {
        new PanelExample();
        }
}
```

**Output:**



**Dialog**

The Dialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Window class.

Unlike Frame, it doesn't have maximize and minimize buttons.

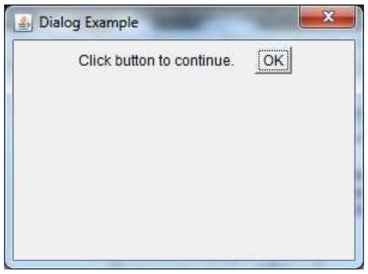
**Dialog class declaration**

public class Dialog extends Window

**Example**

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
    private static Dialog d;
    DialogExample() {
        Frame f= new Frame();
        d = new Dialog(f , "Dialog Example", true);
        d.setLayout( new FlowLayout() );
        Button b = new Button ("OK");
        b.addActionListener ( new ActionListener()
        {
            public void actionPerformed( ActionEvent e )
            {
                DialogExample.d.setVisible(false);
            }
        });
        d.add( new Label ("Click button to continue."));
        d.add(b);
        d.setSize(300,300);
        d.setVisible(true);
    }
    public static void main(String args[])
    {
        new DialogExample();
    }
}
```

**Output:**



**LAYOUT MANAGERS**

The Layout managers allow us to control the arrangement of visual components in GUI forms by controlling the size and position of components inside containers. There is a layout manager associated with each Container object. A layout manager is an instance of a class that implements the LayoutManager interface.

Layout managers are available for general use in several AWT and Swing classes:

- BorderLayout
- BoxLayout
- CardLayout
- FlowLayout
- GridBagLayout
- GridLayout

**Border Layout**

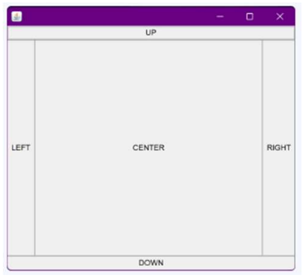
This arrangement will show the components along the container's border. The component can be presented in five different places on this layout: North, South, East, West, and Center are the locations. The center is the default region.

There are 5 types of constructor in Border Layout. They are as following:

1. public static final int NORTH
2. public static final int SOUTH
3. public static final int EAST
4. public static final int WEST
5. public static final int CENTER

**Example:**

```
import java.awt.*;
public class BorderLayoutDemo
{
  public static void main (String[]args)
  {
    Frame f1 = new Frame (); // creating frame
    f1.setSize (250, 250);
    //  creating buttons
    Button b1 = new Button ("UP");
    Button b2 = new Button ("RIGHT");
    Button b3 = new Button ("LEFT");
    Button b4 = new Button ("DOWN");
    Button b5 = new Button ("CENTER");
    f1.add (b1, BorderLayout.NORTH); //placing b1 in north direction
    f1.add (b2, BorderLayout.EAST); //placing b2 in east direction
    f1.add (b3, BorderLayout.WEST); //placing b3 in west direction
    f1.add (b4, BorderLayout.SOUTH); //placing b4 in south direction
    f1.add (b5); // b5 will be placed in center by default
    f1.setVisible (true);
  }}
```



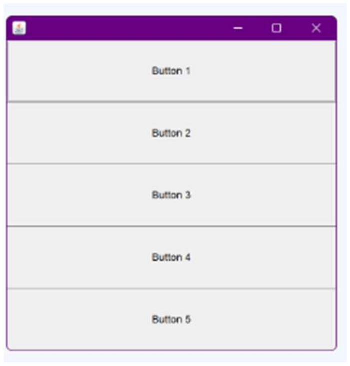
**Box Layout**

The BoxLayout places elements in a container either vertically in a single column or horizontally in a single row. The components are assembled either top to bottom or left to right. The height of each component will be the same and equal to the largest size component if the components are horizontally aligned. The width of each component will be the same and equivalent to the greatest width component if the components are vertically aligned.

X_Axis, Y_Axis, Line_Axis, and Page_Axis are the four constants that comprise the class. The horizontal BoxLayout that the constant X_AXIS generates arranges each component from left to right. All components are arranged vertically in a BoxLayout using the constant Y_AXIS, from top to bottom.

**Example:**

```
import java.awt.*;

import javax.swing.*;

   public class BoxLayoutDemo extends Frame {

    private static final long serialVersionUID = 1L;

    Button buttons[];

 public BoxLayoutDemo () {

   buttons = new Button [5];

   for (int i = 0;i<5;i++) {

      buttons[i] = new Button ("Button " + (i + 1));

      // adding the buttons

      add (buttons[i]);

    }

  // placing the buttons horizontally

setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));

/* replace above line with

setLayout (new BoxLayout(this, BoxLayout.X_AXIS));

To arrange the buttons vertically */

setSize(500,500);
setVisible(true);
}
public static void main(String args[]){
BoxLayoutDemo b=new BoxLayoutDemo();
}    }
```



**Card Layout**

The CardLayout arranges elements as a stack of cards. In a CardLayout, just the top card is visible. CardLayout only shows one component at a time.

The top of the deck will be reserved for the first component to be inserted into the container. The card components are presented horizontally or vertically, with a default gap of zero at the left, right, top, and bottom edges.

There are 2 types of constructor in the Card Layout. They are as following:

1. CardLayout()
2. CardLayout(inthgap, intvgap)

**Example:**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardDemo1 extends JFrame implements ActionListener
{
CardLayoutc_Card;
JButton box1,box2,box3;
Container d;
        CardDemo1()
{
                d=getContentPane();
                c_Card=new CardLayout(40,30);
```

```
                    d.setLayout(c_Card);
                     box1=new JButton("studytonight_1");
                     box2=new JButton("studytonight_2");
                    box3=new JButton("studytonight_3");
                    box1.addActionListener(this);
                    box2.addActionListener(this);
                    box3.addActionListener(this);
                    d.add("P",box1);
                    d.add("Q",box2);
                    d.add("R",box3);
  }
  public void actionPerformed(ActionEvent e)
{
                    c_Card.next(d);
        }
  public static void main(String[] args)
{
                    CardDemo1 obj =new CardDemo1();
                    obj.setSize(500,500);
                    obj.setVisible(true);
                    obj.setDefaultCloseOperation(EXIT_ON_CLOSE);
        }  }
```

**Flow Layout**

Like words on a page, it puts the elements in a container. It completely occupies the top line from top to bottom and left to right. If the container is not broad enough to display all the components, they are wrapped around the line in the order that they are inserted, with the first component appearing at the top left. Component spacing can be adjusted both vertically and horizontally. The components may be positioned to the left, center, or right.

The FlowLayout class defines the following five constants to represent the five various alignments:

LEFT

RIGHT

CENTER

LEADING

TRAILING

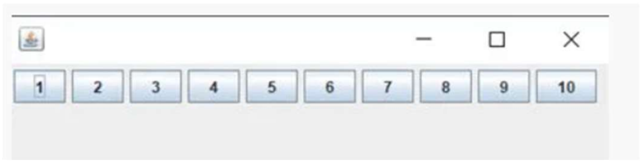
There are 3 types of constructor in the Flow Layout. They are as following:

1. FlowLayout()
2. FlowLayout(int align)
3. FlowLayout(int align, inthgap, intvgap)

**Example:**

```
import java.awt.*;
import javax.swing.*;
public class FlowDemo1{
JFrame frame1;
FlowDemo1(){
frame1=new JFrame();
JButton box1=new JButton("1");
JButton box2=new JButton("2");
JButton box3=new JButton("3");
JButton box4=new JButton("4");
JButton box5=new JButton("5");
JButton box6=new JButton("6");
JButton box7=new JButton("7");
JButton box8=new JButton("8");
JButton box9=new JButton("9");
JButton box10=new JButton("10");

  frame1.add(box1);
  frame1.add(box2);
  frame1.add(box3);
  frame1.add(box4);
  frame1.add(box5);
  frame1.add(box6);
  frame1.add(box7);
  frame1.add(box8);
  frame1.add(box9);
  frame1.add(box10);
  frame1.setLayout(new FlowLayout(FlowLayout.LEFT));
  frame1.setSize(400,400);
```

```
    frame1.setVisible(true);
}
public static void main(String[] args) {
    new FlowDemo1();
} }
```



**Gridbag Layout**

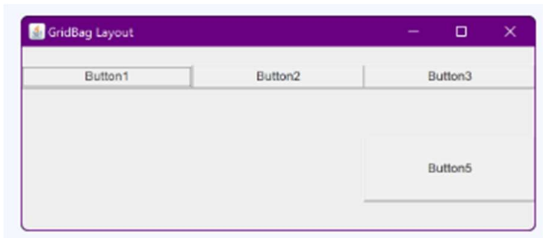
It is a strong layout that organizes all the components into a grid of cells and keeps the object's aspect ratio even when the container is modified. The size of the cells in this arrangement may vary. It allocates components with a constant horizontal and vertical spacing. Using this feature, we can set a default alignment for elements inside the columns or rows.

**Constructors:**

GridBagLayout(): A grid bag layout manager is made using the parameterless constructor.

**Example:**

```
import java.awt.*;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class GridbagDemo
{
    public static void main (String[]args)
    {
        JFrame f1 = new JFrame ("GridBag Layout");
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setSize (250, 250);
        GridBagLayout gb = new GridBagLayout ();
        f1.setLayout (gb);
        GridBagConstraints gc = new GridBagConstraints ();
        //adding buttons
        Button b1 = new Button ("Button1");
        Button b2 = new Button ("Button2");
        Button b3 = new Button ("Button3");
        gc.fill = GridBagConstraints.HORIZONTAL;
        gc.weightx = 0.5;
        gc.weighty = 0.5;
        gc.gridx = 0;
        gc.gridy = 0;
        f1.add (b1, gc);
        gc.gridx = 1;
        gc.gridy = 0;
        f1.add (b2, gc);
        gc.gridx = 2;
        gc.gridy = 0;
        f1.add (b3, gc);
        Button b4 = new Button ("Button4");
        gc.gridx = 0;
        gc.gridy = 1;
        gc.gridwidth = 3;
        gc.ipady = 40;
        Button b5 = new Button ("Button5");
        gc.gridx = 2;
        gc.gridy = 3;
        gc.insets = new Insets (10, 0, 10, 0);
        f1.add (b5, gc);
        f1.pack ();
        f1.setVisible (true);
    }}
```



**Grid Layout**

It arranges each element in a grid of cells that are all the same size, adding each element from left to right and top to bottom. Each grid area will be the same size, and each cell can only hold one component. All cells are automatically adjusted when the container is changed in size. As the elements are added, the order in which they should be placed in a cell is decided.

There are 3 types of constructor in Grid Layout. They are as following:

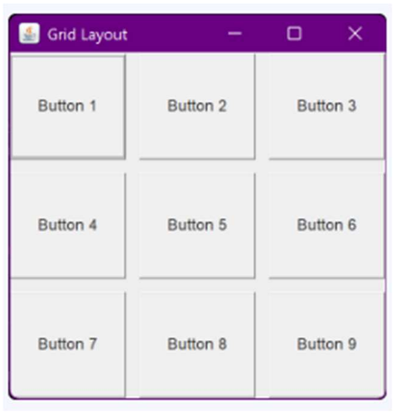
1. GridLayout()

2. GridLayout(int rows, int columns)

3. GridLayout(int rows, int columns, inthgap, int vgap)

**Example:**

```
import java.awt.Button;
import java.awt.GridLayout;
import javax.swing.JFrame;
public class GridLayoutDemo
{
JFrame frame;
Button buttons[];
// constructor
GridLayoutDemo()
{
frame = new JFrame(“Grid Layout”);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
buttons = new Button [9];
// adding the buttons.
for (int i = 0;i<9;i++) {
  buttons[i] = new Button ("Button " + (i + 1));
  frame.add (buttons[i]);
}
// since, we are using the non parameterized constructor, therefore;
// the number of columns is equal to the number of buttons we
// are adding to the frame. The row count remains one.
// setting the grid layout
// a 3 * 3 grid is created with the horizontal gap 10
// and vertical gap 10 ( all these customizations are optional)
//you can create grid layout simply using
// frame.setLayout(new GridLayout());
frame.setLayout(new GridLayout(3, 3, 10, 10));
frame.setSize(300, 300);
frame.setVisible(true);
}
public static void main(String argvs[])
{
new GridLayoutDemo();
}
}
```


Grid Layout
Button 1
Button 2
Button 3
Button 4
Button 5
Button 6
Button 7
Button 8
Button 9