**GUI programming with Swing:** Introduction, limitations of AWT, MVC Architecture, containers. Understanding Layout Managers: Flow, Border, Grid, Card, GridBag.

Introduction:

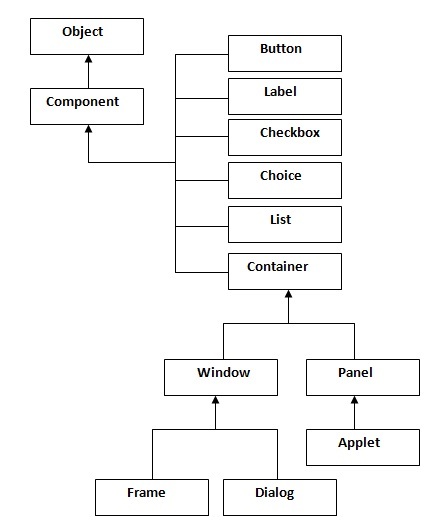
**Java AWT** (Abstract Windowing Toolkit) is an API to develop GUI or window-based application in java.

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 uses the resources of system.

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.



**AWT Components**

**Component class**

**Useful Methods of Component class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void **add(Component c)** | inserts a component on this component. |
| public void **setSize(int width,int height)** | sets the size (width and height) of the component. |
| public void **setLayout(LayoutManager m)** | defines the layout manager for the component. |
| public void **setVisible(boolean status)** | changes the visibility of the component, by default false. |

**1. Label :** it is a passive control because it does not create any event when accessed by the user. The label control is an object of Label. A label displays a single line of read-only text. However the text can be changed by the application programmer but cannot be changed by the end user in any way.

**Class constructors :**

|  |  |
| --- | --- |
| 1 | **Label()**Constructs an empty label. |
| 2 | **Label(String text)**Constructs a new label with the specified string of text, left justified. |
| 3 | **Label(String text, int alignment)**Constructs a new label that presents the specified string of text with the specified alignment. |

**Useful methods :**

**void setText(String text)**

Sets the text for this label to the specified text.

**int getAlignment()**

Gets the current alignment of this label.

**String getText()**

Gets the text of this label.

**2. Button** : Button is a control component that has a label and generates an event when pressed. When a button is pressed and released, AWT sends an instance of ActionEvent to the button.

**Class constructors :**

|  |  |
| --- | --- |
| 1 | **Button()** Constructs a button with an empty string for its label. |
| 2 | **Button(String text)** Constructs a new button with specified label. |

**Useful methods :**

**void addActionListener(ActionListener l)**

Adds the specified action listener to receive action events from this button.

**String getLabel()**

Gets the label of this button.

**void removeActionListener(ActionListener l)**

Removes the specified action listener so that it no longer receives action events from this button.

**void setLabel(String label)**

Sets the button's label to be the specified string.

**3. Checkbox**: Checkbox control is used to turn an option on(true) or off(false). There is label for each checkbox representing what the checkbox does.The state of a checkbox can be changed by clicking on it.

**Class constructors :**

|  |  |
| --- | --- |
| 1 | **Checkbox()** Creates a check box with an empty string for its label. |
| 2 | **Checkbox(String label)** Creates a check box with the specified label. |
| 3 | **Checkbox(String label, boolean state)**  Creates a check box with the specified label and sets the specified state. |
| 4 | **Checkbox(String label, boolean state, CheckboxGroup group)**  Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group. |
| 5 | **Checkbox(String label, CheckboxGroup group, boolean state)**  Creates a check box with the specified label, in the specified check box group, and set to the specified state. |

**Useful methods :**

**void addItemListener(ItemListener l)** Adds the specified item listener to receive item events from this check box.

**String getLabel()** Gets the label of this check box.

**boolean getState()** Determines whether this check box is in the **on** or **off** state.

**void setState(boolean state)** Sets the state of this check box to the specified state.

**void removeItemListener(ItemListener l)**

Removes the specified item listener so that the item listener no longer receives item events from this check box.

**4. List** :The List represents a list of text items. The list can be configured to that user can choose either one item or multiple items.

**Class constructors :**

|  |  |
| --- | --- |
| 1 | **List()**  Creates a new scrolling list. |
| 2 | **List(int rows)**  Creates a new scrolling list initialized with the specified number of visible lines. |
| 3 | **List(int rows, boolean multipleMode)**  Creates a new scrolling list initialized to display the specified number of rows. |

**Useful methods :**

**void add(String item)**

Adds the specified item to the end of scrolling list.

**void add(String item, int index)**

Adds the specified item to the the scrolling list at the position indicated by the index.

**void addActionListener(ActionListener l)**

Adds the specified action listener to receive action events from this list.

**void addItem(String item)**

Deprecated. replaced by add(String).

**void addItem(String item, int index)**

Deprecated. replaced by add(String, int).

**5. Text Field** The TextField component allows the user to edit single line of text.When the user types a key in the text field the event is sent to the TextField. The key event may be key pressed, Key released or key typed. The key event is passed to the registered KeyListener. It is also possible to for an ActionEvent if the ActionEvent is enabled on the textfield then ActionEvent may be fired by pressing the return key.

**Class constructors :**

|  |
| --- |
| **TextField()** Constructs a new text field.  **TextField(int columns)** Constructs a new empty text field with the specified number of columns.  **TextField(String text)** Constructs a new text field initialized with the specified text.  **TextField(String text, int columns)** Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns. |

**Useful methods :**

**void addActionListener(ActionListener l)**

Adds the specified action listener to receive action events from this text field.

**int getColumns()**

Gets the number of columns in this text field.

**char getEchoChar()**

Gets the character that is to be used for echoing.

**void setColumns(int columns)**

Sets the number of columns in this text field.

**void setEchoChar(char c)**

Sets the echo character for this text field.

**void setEchoCharacter(char c)**

Deprecated. As of JDK version 1.1, replaced by setEchoChar(char).

**void setText(String t)**

Sets the text that is presented by this text component to be the specified text.

**6. TextArea :** The TextArea control in AWT provide us multiline editor area. The user can type here as much as he wants. When the text in the text area become larger than the viewable area the scroll bar is automatically appears which help us to scroll the text up & down and right & left.

## Field (constants of TextArea class)

Following are the fields for **java.awt.TextArea** class:

**static int SCROLLBARS\_BOTH**-- Create and display both vertical and horizontal scrollbars.

**static int SCROLLBARS\_HORIZONTAL\_ONLY**-- Create and display horizontal scrollbar only.

**static int SCROLLBARS\_NONE**-- Do not create or display any scrollbars for the text area.

**static int SCROLLBARS\_VERTICAL\_ONLY** -- Create and display vertical scrollbar only.

**Class constructors :**

|  |
| --- |
| **TextArea()** Constructs a new text area with the empty string as text.  **TextArea(int rows, int columns)** Constructs a new text area with the specified number of rows and columns and the empty string as text.  **TextArea(String text)** Constructs a new text area with the specified text.  **TextArea(String text, int rows, int columns)** Constructs a new text area with the specified text, and with the specified number of rows and columns.  **TextArea(String text, int rows, int columns, int scrollbars)**  Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified. |

**Useful methods :**

**void append(String str)** Appends the given text to the text area's current text.

**int getColumns()** Returns the number of columns in this text area.

**int getRows()** Returns the number of rows in the text area.

**int getScrollbarVisibility()** Returns an enumerated value that indicates which scroll bars the text area uses.

**void insert(String str, int pos)** Inserts the specified text at the specified position in this text area.

**void setColumns(int columns)** Sets the number of columns for this text area.

**void setRows(int rows)** Sets the number of rows for this text area.

**7. Choice :** Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.

**Class constructors :**

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

**Useful methods :**

**void add(String item)** Adds an item to this Choice menu.

**void addItemListener(ItemListener l)**

Adds the specified item listener to receive item events from this Choice menu.

**String getItem(int index)** Gets the string at the specified index in this Choice menu.

**int getItemCount()** Returns the number of items in this Choice menu.

**int getSelectedIndex()** Returns the index of the currently selected item.

**String getSelectedItem()** Gets a representation of the current choice as a string.

**void insert(String item, int index)** Inserts the item into this choice at the specified position.

**void remove(int position)** Removes an item from the choice menu at the specified position.

**void remove(String item)** Removes the first occurrence of item from the Choice menu.

**void removeAll()** Removes all items from the choice menu.

**void removeItemListener(ItemListener l)** Removes the specified item listener so that it no longer receives item events from this Choice menu.

**Containers :**

Containers are integral part of AWT GUI components. A container provides a space where a component can be located. A Container in AWT is a component and it has the capacity to add other components to itself. Following are noticable points to be considered.

* Sub classes of Container are called as Containters. For example Panel, Frame and Window.
* A default layout is present in each container which can be overridden using setLayout method.

**Container** : The class **Container** is the super class for the containers of AWT. Container object can contain other AWT components.

**Class constructors :**

**Container()** This creates a new Container.

**Useful methods :**

**Component add(Component comp)** Appends the specified component to the end of this container.

**Component add(Component comp, int index)**

Adds the specified component to this container at the given position.

**void doLayout()** Causes this container to lay out its components.

**float getAlignmentX()** Returns the alignment along the x axis.

**float getAlignmentY()** Returns the alignment along the y axis.

**Component getComponent(int n)** Gets the nth component in this container.

**Component getComponentAt(int x, int y)** Locates the component that contains the x,y position.

**int getComponentCount()** Gets the number of components in this panel.

**LayoutManager getLayout()** Gets the layout manager for this container.

**void paint(Graphics g)** Paints the container.

**void remove(Component comp)** Removes the specified component from this container.

**void remove(int index)** Removes the component, specified by index, from this container.

**void removeAll()** Removes all the components from this container.

**void setFont(Font f)** Sets the font of this container.

**void setLayout(LayoutManager mgr)** Sets the layout manager for this container.

1. **Panel :**

The class **Panel** is the simplest container class. It provides space in which an application can attach any other component, including other panels. It uses FlowLayout as default layout manager.

**Class constructors :**

|  |
| --- |
| **Panel()** Creates a new panel using the default layout manager.  **Panel(LayoutManager layout)** Creates a new panel with the specified layout manager. |

1. **Frame :**

The class **Frame** is a top level window with border and title. It uses BorderLayout as default layout manager.

**Class constructors :**

|  |
| --- |
| **Frame()** Constructs a new instance of Frame that is initially invisible.  **Frame(GraphicsConfiguration gc)** Constructs a new, initially invisible Frame with the specified GraphicsConfiguration.  **Frame(String title)** Constructs a new, initially invisible Frame object with the specified title.  **Frame(String title, GraphicsConfiguration gc)** Constructs a new, initially invisible Frame object with the specified title and a GraphicsConfiguration. |

**Useful methods :**

**Image getIconImage()** Returns the image to be displayed as the icon for this frame.

**Rectangle getMaximizedBounds()** Gets maximized bounds for this frame.

**String getTitle()** Gets the title of the frame.

**boolean isResizable()** Indicates whether this frame is resizable by the user.

**void remove(MenuComponent m)** Removes the specified menu bar from this frame.

1. **Window** : The class **Window** is a top level window with no border and no menubar. It uses BorderLayout as default layout manager.

**Class constructors :**

|  |
| --- |
| **Window(Frame owner)** Constructs a new, initially invisible window with the specified Frame as its owner.  **Window(Window owner)** Constructs a new, initially invisible window with the specified Window as its owner.  **Window(Window owner, GraphicsConfiguration gc)** Constructs a new, initially invisible window with the specified owner Window and a GraphicsConfiguration of a screen device. |

**Useful methods :**

**void addWindowListener(WindowListener l)**

Adds the specified window listener to receive window events from this window.

**void addWindowFocusListener(WindowFocusListener l)**

Adds the specified window focus listener to receive window events from this window.

**void dispose()** Releases all of the native screen resources used by this Window, its subcomponents, and all of its owned children.

**boolean isActive()** Returns whether this Window is active.

**void paint(Graphics g)** Paints the container.

**void toBack()** If this Window is visible, sends this Window to the back and may cause it to lose focus or activation if it is the focused or active Window.

**void toFront()** If this Window is visible, brings this Window to the front and may make it the focused Window.

# **MVC Architecture**

The Model-View-Controller (MVC) is a well-known design pattern in the web development field. It is way to organize our code. It specifies that a program or application shall consist of data model, presentation information and control information. The MVC pattern needs all these components to be separated as different objects.

The model designs based on the MVC architecture follow MVC design pattern. The application logic is separated from the user interface while designing the software using model designs.

The MVC pattern architecture consists of three layers:

* **Model:** It represents the business layer of application. It is an object to carry the data that can also contain the logic to update controller if data is changed.
* **View:** It represents the presentation layer of application. It is used to visualize the data that the model contains.
* **Controller:** It works on both the model and view. It is used to manage the flow of application, i.e. data flow in the model object and to update the view whenever data is changed.

A diagram of a browser

Description automatically generated

**Advantages of MVC Architecture in Java**

MVC architecture offers a lot of advantages for a programmer when developing applications, which include:

* Multiple developers can work with the three layers (Model, View, and Controller) simultaneously
* Offers improved scalability, that supplements the ability of the application to grow
* As components have a low dependency on each other, they are easy to maintain
* A model can be reused by multiple views which provides reusability of code
* Adoption of MVC makes an application more expressive and easy to understand
* Extending and testing of the application becomes easy

**Java AWT Example for creating Frames**

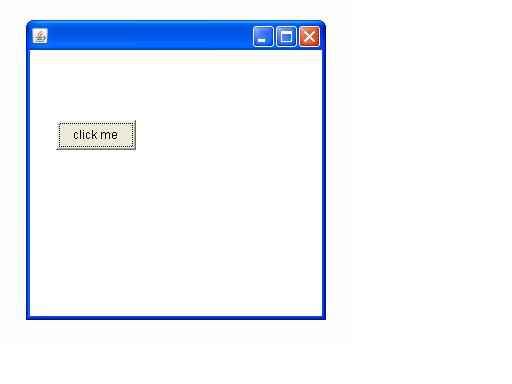
To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

* By extending Frame class (inheritance)
* By creating the object of Frame class (association)

**Simple example of AWT by extending Frame Class**

1. **import** java.awt.\*;
2. **class** First **extends** Frame{
3. First(){
4. Button b=**new** Button("click me");
5. b.setBounds(30,100,80,30);// setting button position
6. add(b);//adding button into frame
7. setSize(300,300);//frame size 300 width and 300 height
8. setLayout(**null**);//no layout manager
9. setVisible(**true**);//now frame will be visible, by default not visible
10. }
11. **public** **static** **void** main(String args[]){
12. First f=**new** First();
13. }}

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.



**Simple example of AWT by Creating an object for Frame Class**

1. **import** java.awt.\*;
2. **class** First2{
3. First2(){
4. Frame f=**new** Frame();
6. Button b=**new** Button("click me");
7. b.setBounds(30,50,80,30);
9. f.add(b);
10. f.setSize(300,300);
11. f.setLayout(**null**);
12. f.setVisible(**true**);
13. }
14. **public** **static** **void** main(String args[]){
15. First2 f=**new** First2();
16. }}

**The Layout Managers**

# **Introduction:**

We create several components like push buttons, checkboxes, radio buttons textfields etc., in GUI. After creating these components they should be placed in the frame or applet or container. While arranging them in the frame or applet or container, they can be arranged in a particular manner by using Layout manager. A layout manager belongs to java.awt package.

## Definition:

A layout manager is a class that is useful to arrange the components in a particular manner in a frame or container or applet.

The following classes represent the layout managers in java:

**• FlowLayout**

**• BorderLayout**

**• GridLayout**

**• CardLayout**

**• GridBagLayout**

***Flow Layout***

**FlowLayout is the default LayoutManager for a Panel and a Applet**. A FlowLayout adds components to the container in rows, working from left to right. When it can’t fit any more components in a row, it starts a new row. When the container gets resized, the components within it get repositioned based on the container’s new size. If sufficient space is available, components within FlowLayout containers are given their preferred size. If there is insufficient space, you do not see the components in their entirety.

**Constants**

FlowLayout defines three constants, all of which are used to specify alignment. The alignment tells FlowLayout where to start positioning the components on each row. Each component is still added from left to right, no matter what the alignment setting is.

public final static int **LEFT** is the constant for left alignment.

public final static int **CENTER** is the constant for center alignment and is the default.

public final static int **RIGHT** is the constant for right alignment.

**Constructors**

**public FlowLayout ()**

This constructor creates a FlowLayout using default settings: center alignment with a horizontal and vertical gap of five pixels. The gap is the space between the different components in the different directions. By default, there will be five pixels between components. The constructor is usually called within a call to setLayout(): setLayout (new FlowLayout()). Figure 7-1 shows how the default FlowLayout behaves with different screen sizes.

**public FlowLayout (int alignment)**

This version of the constructor creates a FlowLayout using the specified alignment and a horizontal and vertical gap of five pixels. Valid alignments are the FlowLayout constants.

**public FlowLayout (int alignment, int hgap, int vgap)**

The final version of the constructor is called by the other two. It requires you to explicitly specify the alignment, horizontal gap (hgap), and vertical gap (vgap). This creates a FlowLayout with an alignment of alignment, horizontal gap of hgap, and vertical gap of vgap. The units for gaps are pixels. It is possible to have negative gaps if you want components to be placed on top of one another.

**Example program which displays buttons in FlowLayout**

import java.awt.\*

public class FlowManager extends Frame{

public static void main(String argv[]){

FlowManager f=new FlowManager();

f.setTitle("FlowLayout manager display");

f.setLayout(new FlowLayout(FlowLayout.LEFT));

f.setSize(400,300);

f.setVisible(true);

}

FlowManager(){

add(new Button("One"));

add(new Button("Two"));

add(new Button("Three"));

add(new Button("Four"));

add(new Button("Five"));

add(new Button("Six"));

add(new Button("Seven"));

add(new Button("Eight"));

add(new Button("Nine"));

add(new Button("Ten"));

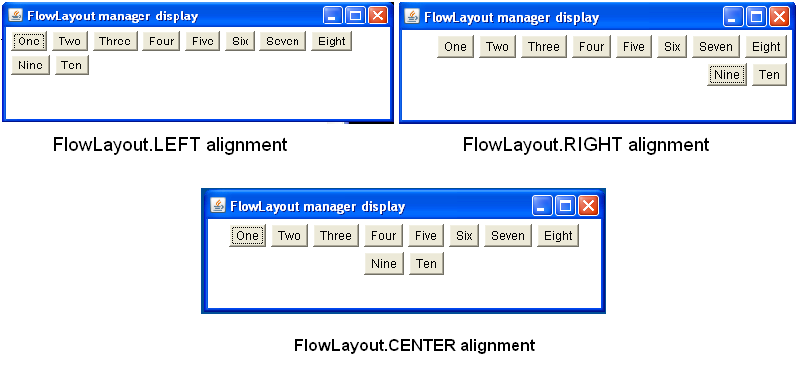
}//End of constructor

}//End of Application

**Output : 1.** f.setLayout(new FlowLayout(FlowLayout.LEFT));

**2.** f.setLayout(new FlowLayout(FlowLayout.RIGHT));

**3.** f.setLayout(new FlowLayout(FlowLayout.CENTER));



**Note : if alignment is not specified it takes CENTER alignment by default**

**BorderLayout**

BorderLayout is the default LayoutManager for a **Window**. It provides a very flexible way of positioning components along the edges of the window.

Border layout is useful to arrange the components in the four borders of the frame as well as in the center. The borders are identified with the names of directions. The top border is specified as ‘north’, the right side border is specified as ‘east’, the bottom one as ‘south’ and the left one as ‘west’. The center is represented as ‘center’.

To create a border layout we can use the following ways;

* **BorderLayout obj= new BorderLayout();**

This creates a border layout object without any gaps between the components.

* **BorderaLayout obj=new BorderLayout(int hgap,int vgap);**

Here **hgap** represents horizontal gap and **vgap** represents vertical gap between components in pixels.

While adding the components to the container the direction should be specified as

**add(“north”, component);**

Here the component is added in the container in north direction.

We can also add the component in the north direction as shown here;  
**add(component,BorderLayout,NORTH);**

**BorderLayout Example**

import java.awt.\*;  
import java.awt.event.\*;  
class BorderLayoutDemo extends Frame  
{  
 Button b1,b2,b3,b4,b5;  
    public BorderLayoutDemo() //Constructor  
    {  
    // Set the frame properties  
    setTitle("BorderLayout Demo");  
    setSize(400,400);

**BorderLayout b= new BorderLayout (6, 6);**  
    setLayout(b);  
    setVisible(true);  
    // Create buttons  
    b1=new Button(" button1");  
    b2=new Button("button2");  
    b3=new Button("button3");

    b4=new Button ("button4");

    b5=new Button ("button5");  
// Add buttons

add (b1,BorderLayout.NORTH);

add (b2,BorderLayout.SOUTH);

add (b3,BorderLayout.EAST);

add(b4,BorderLayout.WEST);

add(b5,BorderLayout.CENTER);

 }

Public static void main (String args[])  
    {  
    new BorderLayoutDemo();  
    }  
}

OutPut:

A screenshot of a computer program

Description automatically generated

**GridLayout**

The GridLayout layout manager is ideal for laying out objects in rows and columns, where each cell in the layout has the same size. Components are added to the layout from left to right, top to bottom.

GridLayout is useful to divide the container into a two dimensional grid form that contains several rows and columns. The container is divided into equal sized rectangles ,and one component is placed in each rectangle.

To create GridLayout object we can write as:

* **GridLayout obj=new GridLayout();**

This creates a GridLayout with a default of one column per component ,in a single row.

* **GridLayout obj=new GridLayout(int rows, int cols);**

This creates a GridLayout with specified number of rows and columns.

* **GridLayout obj=new GridLayout(int rows,int cols,int hgap,int vgap);**

Here **hgap** represents horizontal gap between components and **vgap** represents vertical gap between components.

**GridLayout Example**

import java.awt.\*;  
import java.awt.event.\*;  
class GridLayoutDemo extends Frame  
{  
Button b1,b2,b3,b4,b5;  
  
     public GridLayoutDemo() //Constructor  
    {  
    // Set the frame properties  
    setTitle("GridLayout Demo");  
    setSize(400,400);

**GridLayout grid= new GridLayout(2,2,5,5);**  
    setLayout(grid);  
    setVisible(true);  
    // Create buttons  
    b1=new Button(" button1");  
    b2=new Button("button2");  
    b3=new Button("button3");

    b4=new Button("button4");

// Add buttons

add(b1);

add(b2);

add(b3);

add(b4);

}

public static void main(String args[])  
{

  new GridLayoutDemo();  
 }  
}

OutPut:

A screenshot of a computer

Description automatically generated

**CardLayout**

The CardLayout layout manager is significantly different from the other layouts. Whereas the other layout managers attempt to display all the components within the container at once, a CardLayout displays only one component at a time. (That component could be a Component or another Container.)

A cardlayout object is a layout manager which treats each component as a card.

The first component added to a CardLayout object is the visible component when the container is first displayed.

To create CardLayout object we can use the following ways:

* **CardLayout obj=new CardLayout();**

Here the CardLayout object is created without ant gaps between the components.

* **CardLayout obj=new CardLayout(int hgap, int vgap);**

The preceeding statement creates a cardLayout with specified horizontal and vertical gaps between the components.

* While adding components to the container, we can use add() method as:

**add(“cardname”,component);**

* To retrieve the cards one by one the following methods can be used:
* **Void first (container);** to retrieve the first card.
* **Void last (container):** to retrieve the last card.
* **Void next (container) :** to go to next card.
* **Void previous (container):** to go back to previous card.
* **Void show (container, “card name”):** to see a particular card with the name specified.

**CardLayout Example**

// importing all the necessary packages

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.JFrame;

import javax.swing.\*;

public class CardLayoutDemo extends JFrame implements ActionListener

{

CardLayout cd;

JButton jb1, jb2, jb3;

Container con;

CardLayoutDemo() {

con = getContentPane();

cd = new CardLayout(70, 50);

con.setLayout(cd);

jb1 = new JButton("Hello");

jb2 = new JButton("Hey");

jb3 = new JButton("Hii");

jb1.addActionListener(this);

jb2.addActionListener(this);

jb3.addActionListener(this);

con.add("a", jb1);

con.add("b", jb2);

con.add("c", jb3);

}

public void actionPerformed(ActionEvent e) {

cd.next(con);

}

public static void main(String[] args) {

CardLayoutDemo cl1 = new CardLayoutDemo();

cl1.setTitle("Checking how Card Layout works");

cl1.setSize(400, 400);

cl1.setResizable(false);

cl1.setVisible(true);

cl1.setDefaultCloseOperation(EXIT\_ON\_CLOSE);

}

}

OutPut:

A screenshot of a computer

Description automatically generated

Only the component that is added first will be visible. We can see other components by using the methods card.next(this) ,card.last(this) ,card.previous(this) etc.

**GridBagLayout**

GridBagLayout class represents grid bag layout manager where the components are arranged in rows and columns. This layout is more flexible as compared to other layouts since in this layout, the components can span more than one row or column and the size of the component can be adjusted to fit the display area. The interaction of rows and columns where a component can be placed is called a ’grid’ or ‘display area’.

When positioning the components by using grid bag layout, it is necessary to apply some container or conditions on the components regarding their position, size and space in or around the components etc..such constraints are specified using **GridBagConstraints** class.

To create grid bag layout, we can create an object **to GridBagLayout** class as:

**GridBagLayout obj=new GridBagLayout();**

The following figure displays an applet whose layout is GridBagLayout



## EVENT HANDLING

## Introduction

Any action that user performs on a GUI component must be listened and necessary action should to be taken. For example, if a user clicks on a *Exit* button, then we need to write code to exit the program. So for this, we need to know that the user has clicked the button. This process of knowing is called as listening and the action done by the user is called an event. Writing the corresponding code for a user action is called as **Event handling.**

**Event Handling :** 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. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

**Event Handling Components :** Event Handling has three main components they are ,

**1. Event:** Event is a change of state of an object. Example : clicking a Button, dragging a mouse etc.

**2. Event Source:** Its an object that generates an event. Example : Button ,Window etc

**3. Event Listener:** Its an object that listens to the event. A listener gets notified when an event occurs. Example : ActionListener ,keyListener etc.

**Event Delegation Model :** Event Delegation model is simple when compared to earlier approach. **Source** generates an **Event** and sends it to one or more **listeners**. In this process the listener simply waits until it receives an event. Once received , the listener process the event and then returns. The advantage of this design is that the application logic that process the events is separated from the user interface logic that generates the event. A user interface element is able to “delegate “ the processing of an event to separate piece of code.

In this event delegation model , **listeners must register with a source in order to receive an event notification.** The advantage of doing this is ,notifications are sent only to listeners that want to receive them.

The Delegation Event Model has the following key participants namely:

* **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler.
* **Listener** - It is also known as **EventHandler**. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event and then returns.

**A source must register with listeners , in order to make listeners to receive notifications about specific type of event.** Each type of event has its own registration method.

The general form is : **public void addTypeListener(TypeListener el)**

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

This Event Delegation Model is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

**Important Event Classes and ListenerInterfaces**

A screenshot of a computer

Description automatically generated

**Some of the useful Event Sources**

Event Source and their correspoing registering methods with the listener

**Button :** public void addActionListener(ActionListener a){}

**MenuItem :** public void addActionListener(ActionListener a){}

**TextField :** public void addActionListener(ActionListener a){}

public void addTextListener(TextListener a){}

**TextArea :** public void addTextListener(TextListener a){}

**Checkbox :** public void addItemListener(ItemListener a){}

**Choice :** public void addItemListener(ItemListener a){}

**List :** public void addActionListener(ActionListener a){}

public void addItemListener(ItemListener a){}

**Window**  : public void addWindowListener(WindowListener a){}

**Event Listener Interfaces and their abstract methods**

**The ActionListener Interface**

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

**void actionPerformed(ActionEvent *ae*)**

**The ItemListener Interface**

This interface containsthe **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 containsthree 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 containsfive 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 containstwo 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 TextListener Interface**

This interface containsthe **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 WindowListener Interface**

This interface containsseven methods. The **windowActivated( )** and **indowDeactivated( )**

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*)**

**The ComponentListener Interface**

This interface containsfour 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 contains 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 AdjustmentListener Interface**

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

**void adjustmentValueChanged(AdjustmentEvent *ae*)**

A screenshot of a computer program

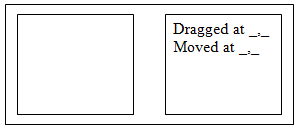
Description automatically generated

A screenshot of a computer program

Description automatically generated

**Mouse Motion Listener**

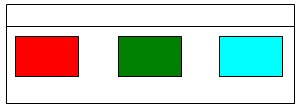
mouseDragged  
mouseMoved  
  
Application : In first TextArea user perform any MouseMotionAction (Dragged or Moved), simultaneously the action name should appear in the second TextArea along with the location(row and column number).

[](http://2.bp.blogspot.com/_TU87B3VcjYc/S48r9XprXCI/AAAAAAAAE7o/Zz2NMP4U4jE/s1600-h/img1.jpg)

import java.awt.\*;  
import java.awt.event.\*;  
class FrameMouseMotion implements MouseMotionListener  
{  
    Frame f;  
    TextArea ta1, ta2;  
  
    FrameMouseMotion()  
    {  
        f = new Frame();  
        ta1=new TextArea(20,20);  
        ta2=new TextArea(20,20);  
      
        f.add(ta1);  
        f.add(ta2);  
          
        f.setLayout(new FlowLayout());  
        ta1.addMouseMotionListener(this);  
  
        f.setSize(400,400);  
        f.setVisible(true);  
    }  
    public void mouseDragged(MouseEvent e)  
    {  
        int x = e.getX();  
        int y = e.getY();  
        ta2.append("Dragged"+" at "+x+","+y+"\n");  
    }  
    public void mouseMoved(MouseEvent e)  
    {  
        int x = e.getX();  
        int y = e.getY();  
        ta2.append("Moved"+" at "+x+","+y+"\n");  
    }  
    public static void main(String [] args)  
    {  
        FrameMouseMotion x = new FrameMouseMotion();  
    }  
}

Mouse Listener

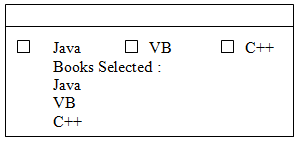
mouseEntered  
mouseExited  
mousePressed  
mouseReleased  
mouseClicked  
  
Applet :     Application displays three canvas with Red, Green and Cyan color. On selecting particular canvas the background color is changed to that canvas color.

[](http://1.bp.blogspot.com/_TU87B3VcjYc/S48p4IcTieI/AAAAAAAAE7Y/otgJRA0gp5M/s1600-h/img1.jpg)

import java.awt.\*;  
import java.awt.event.\*;  
public class FrameMouseCanvas extends Applet implements MouseListener  
{  
    Canvas c1,c2,c3;  
    public void init()  
    {  
        c1=new Canvas();  
        c2=new Canvas();  
        c3=new Canvas();  
  
        add(c1);  
        add(c2);  
        add(c3);  
          
        c1.setSize(100,100);  
        c2.setSize(100,100);  
        c3.setSize(100,100);  
  
        c1.setBackground(Color.red);  
        c2.setBackground(Color.green);  
        c3.setBackground(Color.cyan);  
  
        c1.addMouseListener(this);  
        c2.addMouseListener(this);  
        c3.addMouseListener(this);  
    }  
    public void mouseEntered(MouseEvent e)  
    {   
        if(e.getSource()==c1)  
        {  
            setBackground(Color.red);  
        }  
        else if(e.getSource()==c2)  
        {  
            setBackground(Color.green);  
        }  
        else if(e.getSource()==c3)  
        {  
            setBackground(Color.cyan);  
        }  
    }  
    public void mouseExited(MouseEvent e)  
    {  
        if(e.getSource()==c1||e.getSource()==c2||e.getSource()==c3)  
        {  
            setBackground(Color.white);  
        }  
    }  
    public void mousePressed(MouseEvent e)  
    {  
    }  
    public void mouseClicked(MouseEvent e)  
    {  
    }  
    public void mouseReleased(MouseEvent e)  
    {  
    }  
}  
  
Click new -> File -> html    and save this file as FrameMouseCanvas.html  
<BODY>  
<Applet code=FrameMouseCanvas.class height=200 width=400 >  
</Applet>  
</BODY>

Item Listener

List  
Checkbox  
Choice  
CheckboxGroup (Radio Buttons)  
  
Public void itemStateChanged(ItemEvent) {        ------         ------ }

[](http://3.bp.blogspot.com/_TU87B3VcjYc/S43vIXbGNwI/AAAAAAAAE6k/PJJoAZtE1YQ/s1600-h/img1.jpg)

import java.awt.\*;  
import java.applet.\*;  
import java.awt.event.\*;  
public class FrameCheckbox extends Applet implements ItemListener  
{  
    Checkbox cb1, cb2, cb3;  
  
    public void init()  
{  
        cb1 = new Checkbox("Java");  
        cb2 = new Checkbox("VB");  
        cb3 = new Checkbox("C++");  
  
        add(cb1);  
        add(cb2);  
        add(cb3);  
          
        cb1.addItemListener(this);  
        cb2.addItemListener(this);  
        cb3.addItemListener(this);  
    }  
    public void itemStateChanged(ItemEvent e)  
    {  
        repaint();  
    }  
    public void paint(Graphics g)  
    {  
        g.drawString("Books Selected",100,100);  
        if(cb1.getState())  
        {  
            g.drawString("Java",100,120);  
        }

  if(cb2.getState())  
        {  
            g.drawString("VB",100,130);  
        }  
        if(cb3.getState())  
        {  
            g.drawString("C++",100,140);  
        }  
    }  
}

Click new -> File -> html    and save this file as FrameCheckbox.html  
  
<BODY>  
<Applet code=FrameCheckbox.class height=200 width=400 >  
</Applet>  
</BODY>

**Adapter Classes in Java**

An adapter class provides the default implementation of all methods in an event listener interface. Adapter classes are very useful when you want to process only few of the events that are handled by a particular event listener interface. You can define a new class by extending one of the adapter classes and implement only those events relevant to you.

It is one which contains null body definition for those methods which are inheriting from  
appropriate Listener.  
  
In java.awt.event.\* we have Listener interface called WidowListener which contains seven abstract methods. In the derived class implements WindowListener interface; it is mandatory for derived class to define all the methods even though the derived class is not required.

Adapter classes provide implementation of all methods of interface they are implementing.

Examples of adapter classes for corresponding interfaces are:

MouseAdapter (class)  MouseListener (interface)

WindowAdapter (class)  WindowListener (interface)

Below Table Indicates Listener Interface with their respective adapter class.

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