

Unit 1 GUI Programming [12 Hrs.]

Introduction to Swing:

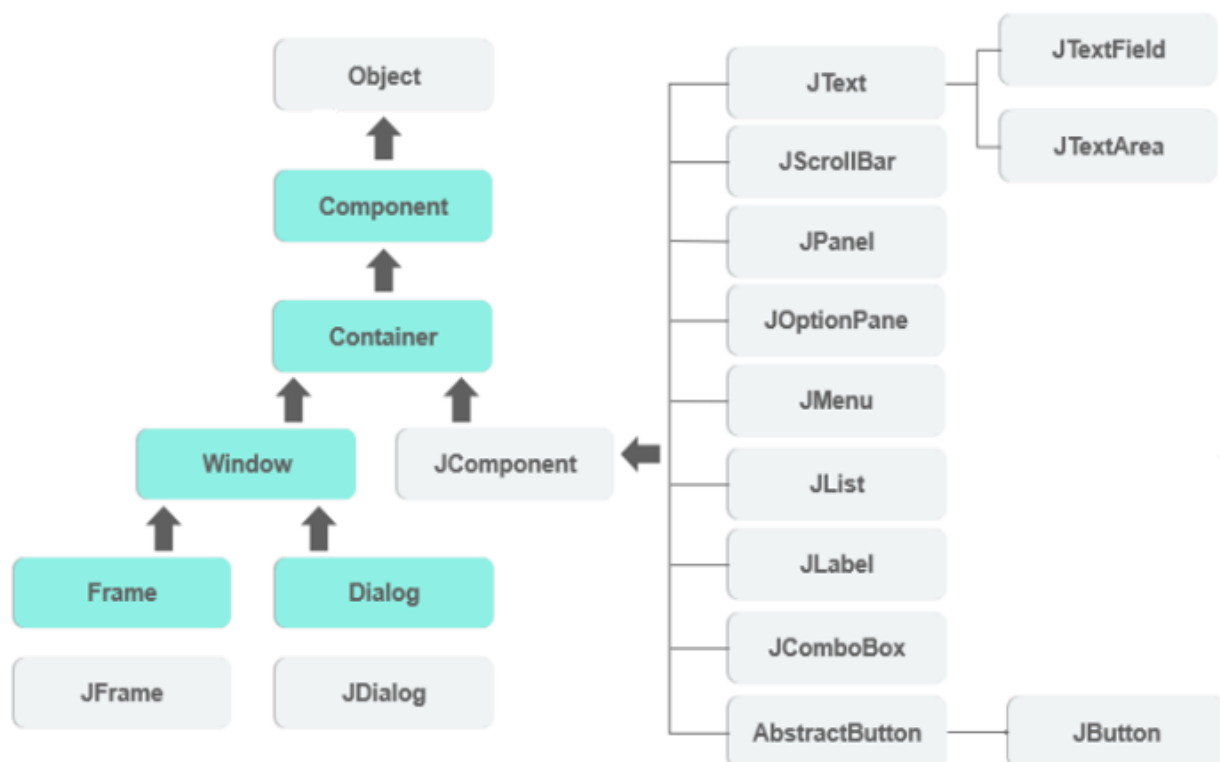
Swing in Java is a lightweight GUI toolkit which has a wide variety of widgets for building optimized window based applications. It is a part of the JFC(Java Foundation Classes). It is build on top of the AWT API and entirely written in java. It is platform independent unlike AWT and has lightweight components.

It becomes easier to build applications since we already have GUI components like button, checkbox etc. This is helpful because we do not have to start from the scratch.

Difference Between AWT and Swing

AWT	SWING
<ul style="list-style-type: none">• Platform Dependent	<ul style="list-style-type: none">• Platform Independent
<ul style="list-style-type: none">• Does not follow MVC	<ul style="list-style-type: none">• Follows MVC
<ul style="list-style-type: none">• Lesser Components	<ul style="list-style-type: none">• More powerful components
<ul style="list-style-type: none">• Does not support pluggable look and feel	<ul style="list-style-type: none">• Supports pluggable look and feel
<ul style="list-style-type: none">• Heavyweight	<ul style="list-style-type: none">• Lightweight

Java Swing Class Hierarchy



Container Class

Any class which has other components in it is called as a container class. For building GUI applications at least one container class is necessary.

Following are the three types of container classes:

1. **Panel** – It is used to organize components on to a window
2. **Frame** – A fully functioning window with icons and titles
3. **Dialog** – It is like a pop up window but not fully functional like the frame

Top level Containers

- It inherits Component and Container of AWT.
- It cannot be contained within other containers. ○ Heavyweight.
- Example: JFrame, JDialog, JApplet

Lightweight Containers

- It inherits JComponent class.
- It is a general purpose container.
- It can be used to organize related components together. ○ Example: JPanel

JFrame(a top level window in Swing)

Whenever you create a graphical user interface with Java Swing functionality, you will need a container for your application. In the case of Swing, this container is called a JFrame. All GUI applications require a JFrame. In fact, some Applets even use a JFrame. Why?

You can't build a house without a foundation. The same is true in Java: Without a container in which to put all other elements, you won't have a GUI application. In other words, the JFrame is required as the foundation or base container for all other graphical components.

Java Swing applications can be run on any system that supports Java. These applications are lightweight. This means that don't take up much space or use many system resources.

JFrame is a class in Java and has its own methods and constructors. Methods are functions that impact the JFrame, such as setting the size or visibility. Constructors are run when the instance is created: One constructor can create a blank JFrame, while another can create it with a default title.

When a new JFrame is created, you actually create an instance of the JFrame class. You can create an empty one, or one with a title. If you pass a string into the constructor, a title is created as follows:

```
JFrame f = new JFrame();  
//Or overload the constructor and give it a title:  
JFrame f1 = new JFrame("BCA Sixth Sem");
```

JPanel in Swing

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class. It doesn't have title bar.

```
JFrame f= new JFrame("Panel Example");
JPanel panel=new JPanel();
JLabel lbl=new JLabel("Testing label");
panel.add(lbl);
f.add(panel);
```

JButton

JButton is in implementation of a push button. It is used to trigger an action if the user clicks on it. JButton can display a text, an icon, or both.

```
JButton btn=new JButton("Click Me");
```

JLabel

Label is a simple component for displaying text, images or both. It does not react to input events.

```
JLabel lbl=new Label("This is a label");
```

JTextField

JTextField is a text component that allows editing of a single line of non-formatted text.

```
JTextField txt=new JTextField();
```

JCheckBox

JCheckBox is a box with a label that has two states: on and off. If the check box is selected, it is represented by a tick in a box. A check box can be used to show or hide a splashscreen at startup, toggle visibility of a toolbar etc.

```
JCheckBox chk1=new JCheckBox("BCA");
JCheckBox chk2=new JCheckBox("BBA");
```

JRadioButton

JRadioButton allows the user to select a single exclusive choice from a group of options. It is used with the ButtonGroup component.

```
JRadioButton rad1=new JRadioButton ("Male");
JRadioButton red2=new JRadioButton ("Female");
ButtonGroup grp=new ButtonGroup();
grp.add(rad1);
grp.add(rad2);
```

JComboBox

JComboBox is a component that combines a button or editable field and a drop-down list. The user can select a value from the drop-down list, which appears at the user's request. If you make the combo box editable, then the combo box includes an editable field into which the user can type a value.

```
String arr[]={“BCA”,“BBA”,“MCA”,“MBA”};
JComboBox cmb=new JComboBox(arr);
//or
JComboBox<String> cmb=new JComboBox<String>();
cmb.addItem(“BCA”);
cmb.addItem(“BBA”);
cmb.addItem(“MCA”);
cmb.addItem(“MBA”);
```

JList

JList is a component that displays a list of objects. It allows the user to select one or more items.

```
String arr[]={“BCA”,“BBA”,“MCA”,“MBA”};
JList cmb=new JList(arr);
```

JTextArea

A JTextArea is a multiline text area that displays plain text. It is lightweight component for working with text. The component does not handle scrolling. For this task, we use JScrollPane component.

```
JTextArea txt=new JTextArea();
```

JTable

The JTable class is a part of Java Swing Package and is generally used to display or edit two-dimensional data that is having both rows and columns. It is similar to a spreadsheet. This arranges data in a tabular form.

```
// Data to be displayed in the JTable
String[][] data = {
    { "Kundan Kumar Jha", "4031", "CSE" },
    { "Anand Jha", "6014", "IT" }
};

// Column Names
String[] columnNames = { "Name", "Roll Number", "Department" };
// Initializing the JTable
JTable j = new JTable(data, columnNames);
```

JMenu

The JMenuBar class is used to display menubar on the window or frame. It may have several menus.

The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.

The object of JMenuItem class adds a simple labeled menu item. The items used in a menu must belong to the JMenuItem or any of its subclass.

```
JMenuBar mb=new JMenuBar();
JMenu menu1=new JMenu("Courses");
JMenuItem item1=new JMenuItem("BCA");
JMenuItem item2=new JMenuItem("BBA");
menu1.add(item1);
menu1.add(item2);
mb.add(menu1);
```

Displaying Information in JComponent

The JComponent class is the base class of all Swing components except top-level containers. Swing components whose names begin with "J" are descendants of the JComponent class. For example, JButton, JScrollPane, JPanel, JTable etc. But, JFrame and JDialog don't inherit JComponent class because they are the child of top-level containers.

The JComponent class extends the Container class which itself extends Component. The Container class has support for adding components to the container.

```
class MyJComponent extends JComponent {
    public void paint(Graphics g) {
        g.setColor(Color.green);
        g.drawLine(30, 30, 100, 100);    //x1,y1,x2,y2
    }
}
```

Note: This Component must be added to frame to draw a line.

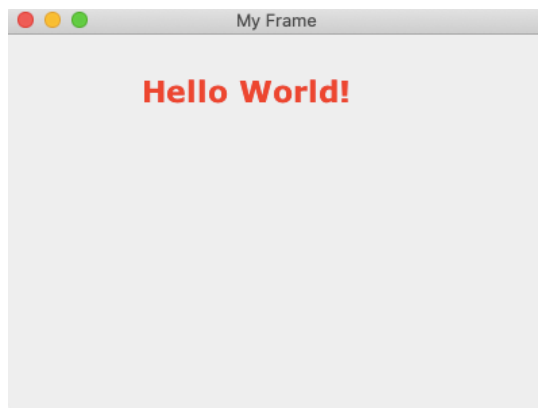
Example 2 – Displaying a String:

```
import javax.swing.*;
import java.awt.*;
class MyComponent extends JComponent{
    public void paint(Graphics g) {
        //setting font color
        g.setColor(Color.RED);
        //setting font
        g.setFont(new Font("Verdana",Font.BOLD,22));
        //displaying string
        g.drawString("Hello World!", 100, 50);
    }
}
```

```

public class ShapesEx {
    public static void main(String[] args) {
        JFrame jf=new JFrame("My Frame");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLocationRelativeTo(null);
        MyComponent comp=new MyComponent();
        jf.add(comp);
        jf.setVisible(true);
    }
}

```



Working with 2D Shapes:

Drawing a line

```

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

class MyComponent extends JComponent{
    public void paint(Graphics g) {
        //casting graphics to graphics 2d object
        //needed for more effects...on shapes
        Graphics2D g2=(Graphics2D)g;
        //setting color
        g2.setColor(Color.RED);
        //changing width
        g2.setStroke(new BasicStroke(10));
        //drawing a line
        g2.drawLine(120, 30, 50, 140); //x1,y1,x2,y2
    }
}

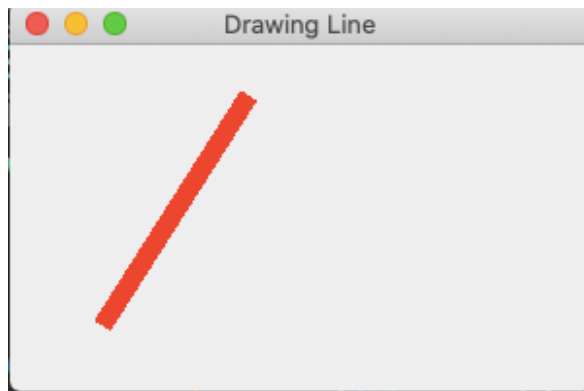
```

```

public class MyTest {
    public static void main(String[] args) {
        JFrame jf=new JFrame("Drawing Line");
        jf.setSize(300, 200); //width,height
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        /*MyComponent draw=new MyComponent();
        jf.add(draw);*/
        jf.add(new MyComponent());
        jf.setVisible(true);
    }
}

```

Output:



Drawing a Rectangle

```

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

class MyComponent extends JComponent{
    public void paint(Graphics g) {
        //casting graphics to graphics 2d object
        //needed for more effects...on shapes
        Graphics2D g2=(Graphics2D)g;
        //changing width
        g2.setStroke(new BasicStroke(10));
        //setting background color
        g2.setColor(Color.GREEN);
        g2.fillRect(30, 40, 120, 60);
        //drawing a rectangle
        g2.setColor(Color.RED);
        g2.drawRect(30, 40, 120, 60); //x,y,width,height
    }
}

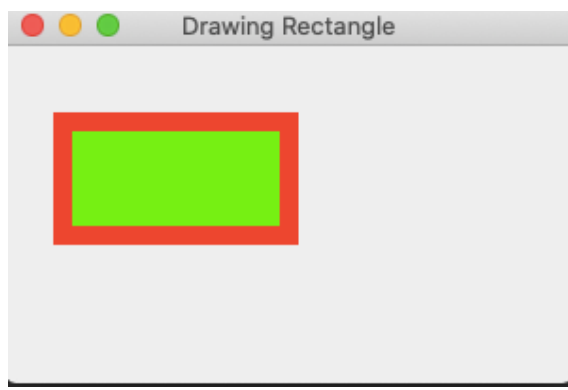
```

```

public class MyTest {
    public static void main(String[] args) {
        JFrame jf=new JFrame("Drawing Rectangle");
        jf.setSize(300, 200); //width,height
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        /*MyComponent draw=new MyComponent();
        jf.add(draw);*/
        jf.add(new MyComponent());
        jf.setVisible(true);
    }
}

```

Output:



Drawing a Square:

```

g2.drawRect(30, 40, 120, 120); //x,y,width,height
//height & width Same

```

Drawing a Circle:

```

g2.drawOval(30, 40, 120, 120); //x,y,width,height
//height & width Same

```

Drawing an Ellipse:

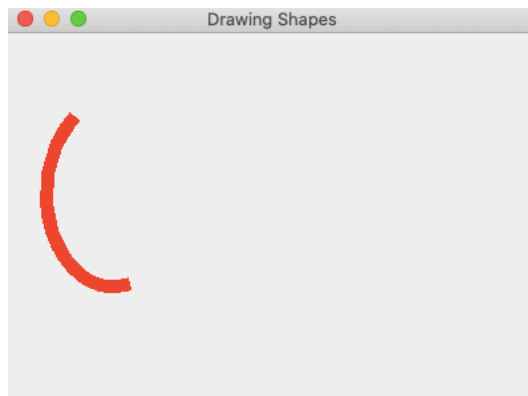
```

g2.drawOval(30, 40, 150, 80); //x,y,width,height
//height & width Different

```

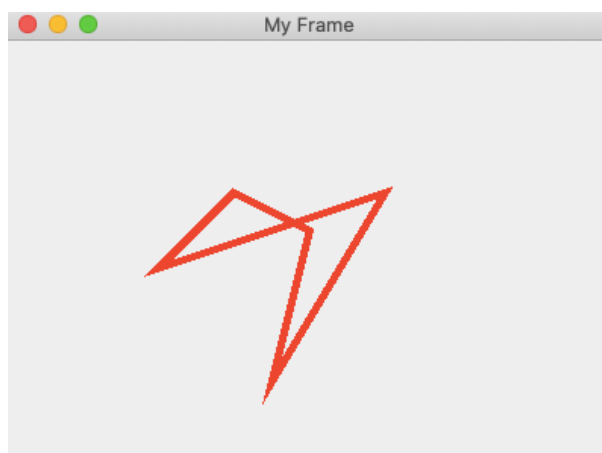

Drawing an Arc:

```
public void paint(Graphics g) {  
    Graphics2D g2=(Graphics2D)g;  
    g2.setColor(Color.RED);  
    g2.setStroke(new BasicStroke(10));  
    //g2.drawArc(x, y, width, height, startAngle, arcAngle);  
    g2.drawArc(30, 50, 100, 140, 130, 150);  
}
```



Drawing a Polygon

```
public void paint(Graphics g) {  
    Graphics2D g2=(Graphics2D)g;  
    g2.setColor(Color.RED);  
    g2.setStroke(new BasicStroke(5));  
    //drawing a polygon  
    int xPoly[] = {100, 150, 200, 175, 250};  
    int yPoly[] = {150, 100, 125, 225, 100};  
    g2.drawPolygon(xPoly, yPoly, 5);  
    //g2.drawPolyline(xPoints, yPoints, nPoints);  
}
```



Using Special Font and Color for Text:

```
JLabel lbl=new JLabel();
lbl.setText("This is a text!");
lbl.setSize(100, 50);
//setting color
lbl.setForeground(Color.RED);
//setting font
lbl.setFont(new Font("Verdana",Font.BOLD,22));
```

Displaying Images:

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

class SwingEx{
    SwingEx(){
        JFrame jf=new JFrame("My Frame");
        jf.setSize(350, 350);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        ImageIcon image=new ImageIcon
            ("/Users/mymack/Desktop/test.jpg");
        Image newimg=image.getImage().getScaledInstance(200, 200,
            Image.SCALE_SMOOTH);
        ImageIcon newImg=new ImageIcon(newimg);
        JLabel lbl=new JLabel(newImg);
        lbl.setLocation(60, 50);
        lbl.setSize(200, 200);
        jf.add(lbl);

        jf.setVisible(true);
    }
}

public class Example {
    public static void main(String[] args) {
        new SwingEx();
    }
}
```

Output:



Event handling

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. **Event describes the change in state of any object.** For Example: Pressing a button, entering a character in Textbox, Clicking or Dragging a mouse, etc.

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.

How Events are handled?

A source generates an Event and send it to one or more listeners registered with the source. Once event is received by the listener, they process the event and then return. Events are supported by a number of Java packages, like **java.util**, **java.awt** and **java.awt.event**.

Important Event Classes and Interface

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener

KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	

Example of ActionListener

```

JButton btn=new JButton("Click Here");
btn.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
        //event handing code here
    }
});

```

Example of ItemListener

```

JComboBox cmb=new JComboBox(arr);
cmb.addItemListener(new ItemListener(){
    public void actionPerformed(ItemEvent ie){
        if (ie.getStateChange() == ItemEvent.SELECTED)
            //event handing code here
    }
});

```

Keyboard and Mouse Events

The Java **MouseListener** is notified whenever you change the state of mouse. It is notified against **MouseEvent**. The **MouseListener** interface is found in **java.awt.event** package. It has five methods.

```
void mouseClicked(MouseEvent e);
void mouseEntered(MouseEvent e);
void mouseExited(MouseEvent e);
void mousePressed(MouseEvent e);
void mouseReleased(MouseEvent e);
```

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 has three methods.

```
void keyPressed(KeyEvent e);
void keyReleased(KeyEvent e);
void keyTyped(KeyEvent e);
```

Example of Mouse Events:

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

class SwingEx{
    SwingEx(){
        JFrame jf=new JFrame("My Frame");
        jf.setSize(350, 350);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        JButton btn=new JButton("Click Me!");
        btn.setSize(100, 50);
        btn.setLocation(100, 50);
        jf.add(btn);

        JLabel lbl=new JLabel("Result");
        lbl.setSize(100, 50);
        lbl.setLocation(100, 100);
        jf.add(lbl);

        //adding mouse click event
        btn.addMouseListener(new MouseListener() {
            public void mouseClicked(MouseEvent e) {
                lbl.setText("Mouse Clicked!");
            }
        })
    }
}
```

```

        public void mousePressed(MouseEvent e) {
            lbl.setText("Mouse Pressed!");
        }

        public void mouseReleased(MouseEvent e) {
            lbl.setText("Mouse Released!");
        }

        public void mouseEntered(MouseEvent e) {
            lbl.setText("Mouse Entered!");
        }

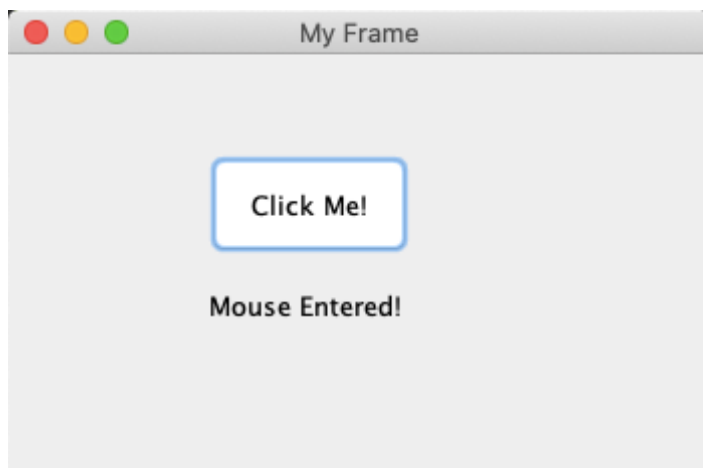
        public void mouseExited(MouseEvent e) {
            lbl.setText("Mouse Exited!");
        }
    });

    jf.setVisible(true);
}

public class Example {
    public static void main(String[] args) {
        new SwingEx();
    }
}

```

Output:



Example of Key Events:

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

class SwingEx{
    SwingEx(){
        JFrame jf=new JFrame("My Frame");
        jf.setSize(350, 350);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        JTextField txt=new JTextField();
        txt.setSize(200, 50);
        txt.setLocation(60, 100);
        jf.add(txt);

        JLabel lbl=new JLabel("Event Result");
        lbl.setSize(100, 50);
        lbl.setLocation(100, 150);
        jf.add(lbl);

        //adding Key Event
        txt.addKeyListener(new KeyListener() {
            public void keyTyped(KeyEvent e) {
                lbl.setText("Key Typed!");
            }

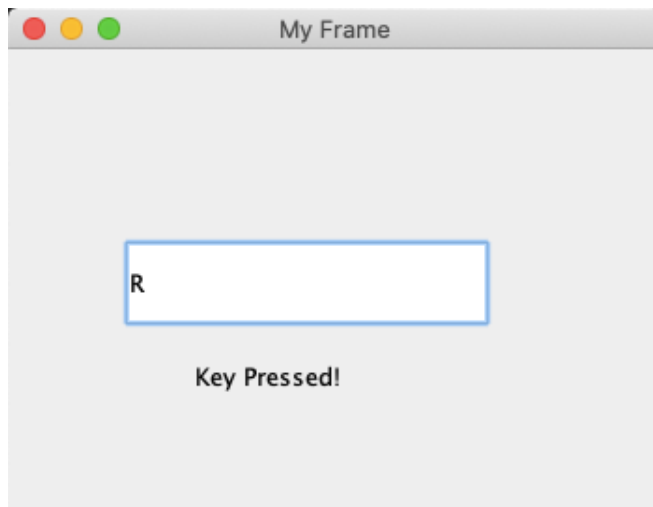
            public void keyPressed(KeyEvent e) {
                lbl.setText("Key Pressed!");
            }

            public void keyReleased(KeyEvent e) {
                lbl.setText("Key Released!");
            }
        });

        jf.setVisible(true);
    }
}

public class Example {
    public static void main(String[] args) {
        new SwingEx();
    }
}
```

Output:



Example of ItemListener:

Event triggered on Combo Box item selection.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class SwingEx{
    SwingEx(){
        JFrame jf=new JFrame("My Frame");
        jf.setSize(400,350);
        jf.setLocationRelativeTo(null);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        JComboBox cmb=new JComboBox();
        cmb.addItem("BCA");
        cmb.addItem("BBA");
        cmb.addItem("MCA");
        cmb.addItem("MBA");
        cmb.setSize(120, 60);
        cmb.setLocation(100,60);
        cmb.setFocusable(false);
        jf.add(cmb);

        cmb.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                if(e.getStateChange()==ItemEvent.SELECTED) {
                    //getting selected Item
                    String item=cmb.getSelectedItem()
                        .toString();
                }
            }
        });
    }
}
```



```

        //displaying
        System.out.println("Selected Item: "+item);
    }
}

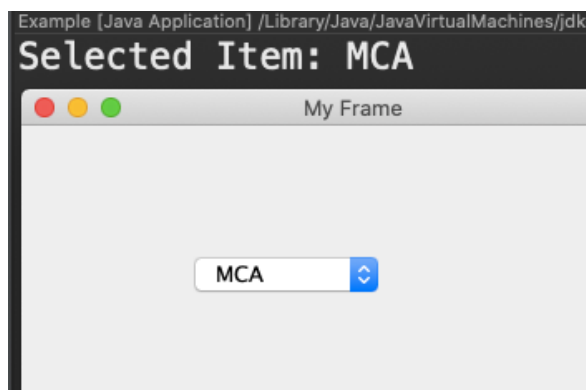
});

jf.setVisible(true);
}
}

public class Example {
    public static void main(String[] args) {
        new SwingEx();
    }
}

```

Output:



Adapter Class

Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.

Advantages of using Adapter classes:

- It assists the unrelated classes to work combinedly.
- It provides ways to use classes in different ways.
- It increases the transparency of classes.
- It provides a way to include related patterns in the class.
- It provides a pluggable kit for developing an application.
- It increases the reusability of the class.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Example 1 – Using Mouse Adapter Class

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

class SwingEx{
    SwingEx(){
        JFrame jf=new JFrame("My Frame");
        jf.setSize(400,350);
        jf.setLocationRelativeTo(null);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);

        JButton btn=new JButton("Click Me!");
        btn.setSize(120, 50);
        btn.setLocation(100, 50);
        jf.add(btn);
    }
}
```

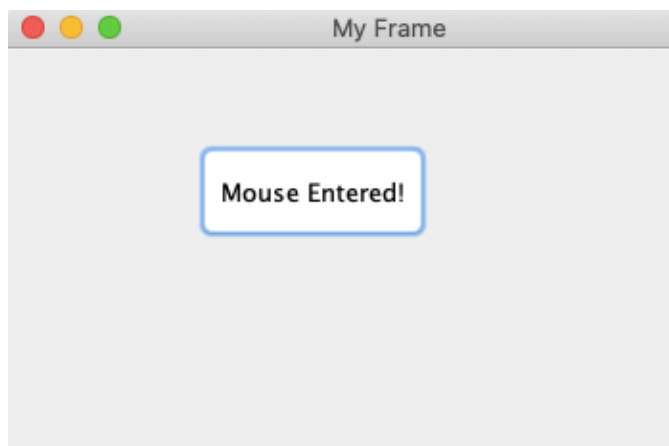
```

        //Using mouse adapter class
        btn.addMouseListener(new MouseAdapter() {
            //we don't need to implement all methods of MouseListener
            //we can use method as per the requirement
            public void mouseEntered(MouseEvent e) {
                btn.setText("Mouse Entered!");
            }
        });

        jf.setVisible(true);
    }
}

public class Example {
    public static void main(String[] args) {
        new SwingEx();
    }
}

```



Example 2 – Using Key Adapter Class

```

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

class SwingEx{
    SwingEx(){
        JFrame jf=new JFrame("My Frame");
        jf.setSize(400,350);
        jf.setLocationRelativeTo(null);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);

        JTextField txt=new JTextField();
    }
}

```

```

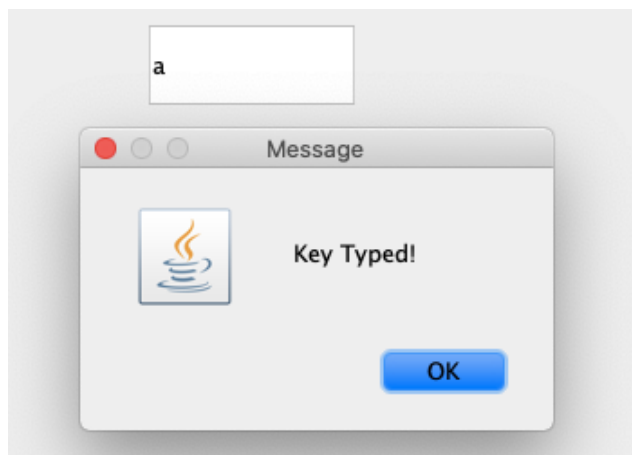
        txt.setSize(120, 50);
        txt.setLocation(100, 50);
        jf.add(txt);

        txt.addKeyListener(new KeyAdapter() {
            public void keyTyped(KeyEvent e) {
                JOptionPane.showMessageDialog(null, "Key Typed!");
            }
        });

        jf.setVisible(true);
    }
}

public class Example {
    public static void main(String[] args) {
        new SwingEx();
    }
}

```



Example – 3 Using Window Adapter Class

```

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

class SwingEx{
    SwingEx(){
        JFrame jf=new JFrame("My Frame");
        jf.setSize(300,250);
        jf.setLocationRelativeTo(null);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
    }
}

```

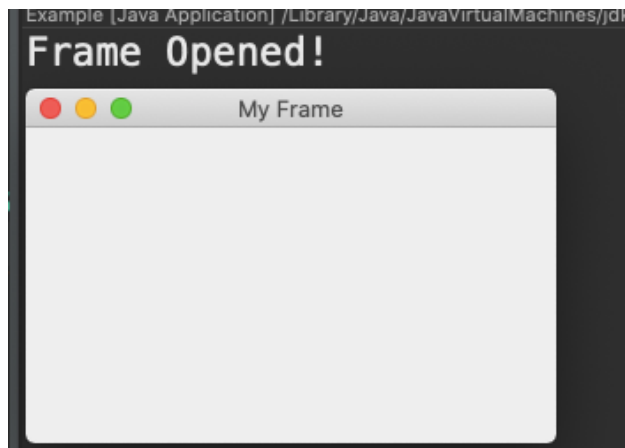
```

        //adding window event to jframe
        jf.addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent e) {
                System.out.println("Frame Opened!");
            }
        });

        jf.setVisible(true);
    }
}

public class Example {
    public static void main(String[] args) {
        new SwingEx();
    }
}

```

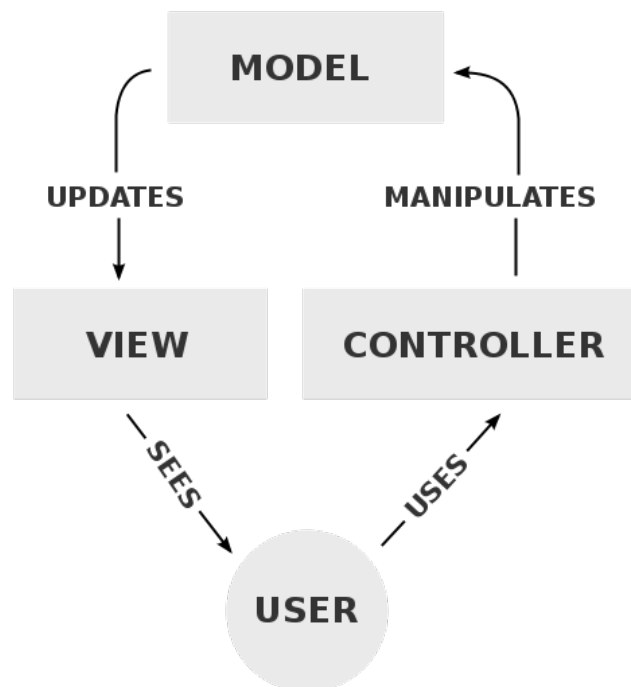


MVC Design Pattern

The Model-View-Controller is a well known software architectural pattern ideal to implement user interfaces on computers by dividing an application into three interconnected parts. Main goal of Model-View-Controller, also known as MVC, is to separate internal representations of an application from the ways information are presented to the user. Initially, MVC was designed for desktop GUI applications but it's quickly become an extremely popular pattern for designing web applications too.

MVC pattern has the three following components :

- **Model that manages data, logic and rules of the application**
- **View that is used to present data to user**
- **Controller that accepts input from the user and converts it to commands for the Model or View.**



The Model receives commands and data from the Controller. It stores these data and updates the View. The View lets to present data provided by the Model to the user. The Controller accepts inputs from the user and converts it to commands for the Model or the View.

Model

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

View

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

Controller

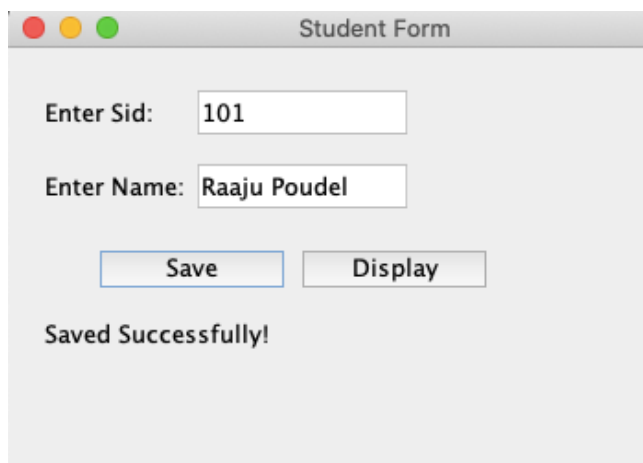
Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

Example of MVC design Pattern

3 classes for model, view and controller required + 1 class for startup.
So, in total 4 classes are required.

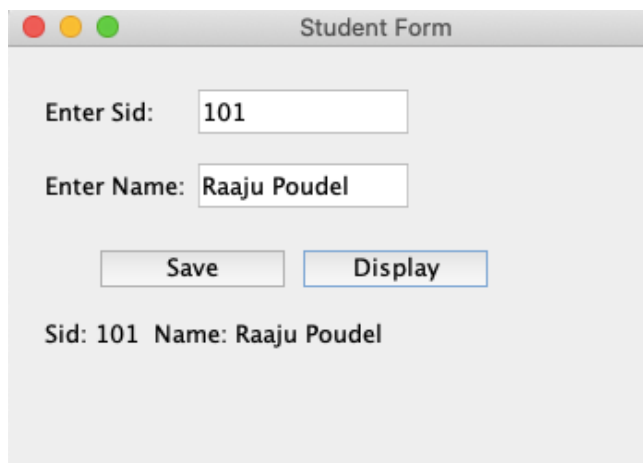
Understanding the UI,

When user clicks save button data will be saved in model as follows:



A screenshot of a 'Student Form' window. It contains two text input fields: 'Enter Sid:' with the value '101' and 'Enter Name:' with the value 'Raaju Poudel'. Below these fields are two buttons: 'Save' and 'Display'. The 'Save' button is highlighted with a blue border. Below the buttons, the text 'Saved Successfully!' is displayed.

When user clicks display button data inputted will be displayed in label as follows:



A screenshot of the same 'Student Form' window. The 'Display' button is now highlighted with a blue border. Below the buttons, the text 'Sid: 101 Name: Raaju Poudel' is displayed, showing the data entered in the form fields.

Source Code

StudentView.java

```
import javax.swing.*;
public class StudentView {
    public JLabel lbl1, lbl2, lbl3;
    public JTextField txt1, txt2;
    public JButton btn1, btn2;

    public StudentView() {
        JFrame jf = new JFrame("Student Form");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        lbl1 = new JLabel("Enter Sid:");
        lbl1.setSize(100, 30);
        lbl1.setLocation(20, 20);
        jf.add(lbl1);

        txt1 = new JTextField();
        txt1.setSize(120, 30);
        txt1.setLocation(100, 20);
        jf.add(txt1);

        lbl2 = new JLabel("Enter Name:");
        lbl2.setSize(100, 30);
        lbl2.setLocation(20, 60);
        jf.add(lbl2);

        txt2 = new JTextField();
        txt2.setSize(120, 30);
        txt2.setLocation(100, 60);
        jf.add(txt2);

        btn1 = new JButton("Save");
        btn1.setSize(100, 20);
        btn1.setLocation(50, 110);
        jf.add(btn1);

        btn2 = new JButton("Display");
        btn2.setSize(100, 20);
        btn2.setLocation(160, 110);
        jf.add(btn2);

        lbl3 = new JLabel("Result:");
```



```

        lbl3.setSize(200, 30);
        lbl3.setLocation(20, 140);
        jf.add(lbl3);

        jf.setVisible(true);
    }
}

```

StudentModel.java

```

public class StudentModel {
    private int sid;
    private String name;

    public void setId(int sid) {
        this.sid=sid;
    }

    public int getId() {
        return sid;
    }

    public void setName(String name) {
        this.name=name;
    }

    public String getName() {
        return name;
    }
}

```

StudentController.java

```

import javax.swing.*.*;
public class StudentController {
    StudentView v;
    StudentModel m;
    public void initController() {
        //initializing view
        v=new StudentView();
        //initializing model
        m=new StudentModel();
        //registering events
        v.btn1.addActionListener(e->saveClicked());
        v.btn2.addActionListener(e->displayClicked());
    }
}

```

```

public void saveClicked() {
    int sid=Integer.parseInt(v.txt1.getText());
    String name=v.txt2.getText();
    m.setId(sid);
    m.setName(name);
    JOptionPane.showMessageDialog(null, "Saved Successfully!");
}

public void displayClicked() {
    v.lbl3.setText("Sid: "+m.getId()+" Name: "+m.getName());
}
}

```

**And finally,
Driver Class...**

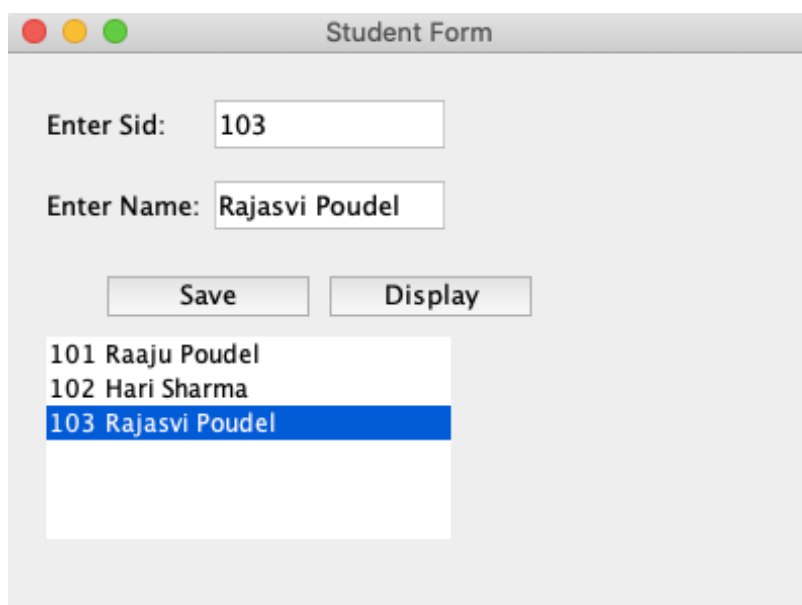
Example.java

```

//Driver Class
public class Example {
    public static void main(String[] args) {
        StudentController cont=new StudentController();
        cont.initController();
    }
}

```

Example – 2 (Displaying Multiple Rows using JList)



StudentView.java

```
import javax.swing.*;
public class StudentView {
    public JLabel lbl1, lbl2, lbl3;
    public JTextField txt1, txt2;
    public JButton btn1, btn2;
    //required for creating a empty list
    DefaultListModel lmodel;

    public StudentView() {
        JFrame jf=new JFrame("Student Form");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        lbl1=new JLabel("Enter Sid:");
        lbl1.setSize(100, 30);
        lbl1.setLocation(20, 20);
        jf.add(lbl1);

        txt1=new JTextField();
        txt1.setSize(120, 30);
        txt1.setLocation(100, 20);
        jf.add(txt1);

        lbl2=new JLabel("Enter Name:");
        lbl2.setSize(100, 30);
        lbl2.setLocation(20, 60);
        jf.add(lbl2);

        txt2=new JTextField();
        txt2.setSize(120, 30);
        txt2.setLocation(100, 60);
        jf.add(txt2);

        btn1=new JButton("Save");
        btn1.setSize(100, 20);
        btn1.setLocation(50, 110);
        jf.add(btn1);

        btn2=new JButton("Display");
        btn2.setSize(100, 20);
        btn2.setLocation(160, 110);
        jf.add(btn2);

        //creating a JList
        lmodel=new DefaultListModel();
    }
}
```

```

        JList jl=new JList(lmodel);
        jl.setSize(200, 100);
        jl.setLocation(20, 140);
        jf.add(jl);

        jf.setVisible(true);
    }
}

```

StudentModel.java

```

public class StudentModel {
    private int sid;
    private String name;

    public void setId(int sid) {
        this.sid=sid;
    }

    public int getId() {
        return sid;
    }

    public void setName(String name) {
        this.name=name;
    }

    public String getName() {
        return name;
    }
}

```

StudentController.java

```

import java.util.*;
import javax.swing.*;
public class StudentController {
    StudentView v;
    //for storing multiple data
    ArrayList<StudentModel> data;

    public void initController() {
        //model initialization not required & will be initialized
        //in add button clicked
        //initializing view
        v=new StudentView();
        //initializing ArrayList
        data=new ArrayList<>();
    }
}

```

```

        //registering events
        v.btn1.addActionListener(e->saveClicked());
        v.btn2.addActionListener(e->displayClicked());
    }

    public void saveClicked() {
        int sid=Integer.parseInt(v.txt1.getText());
        String name=v.txt2.getText();
        //transferring to model
        StudentModel m=new StudentModel();
        m.setId(sid);
        m.setName(name);
        //adding model to ArrayList
        data.add(m);
        JOptionPane.showMessageDialog(null, "Saved Successfully!");
    }

    public void displayClicked() {
        //clearing previous data'
        v.lmodel.clear();
        //getting data from ArrayList
        for(StudentModel st:data) {
            //System.out.println(st.getId()+" "+st.getName());
            v.lmodel.addElement(st.getId()+" "+st.getName());
        }
    }
}

```

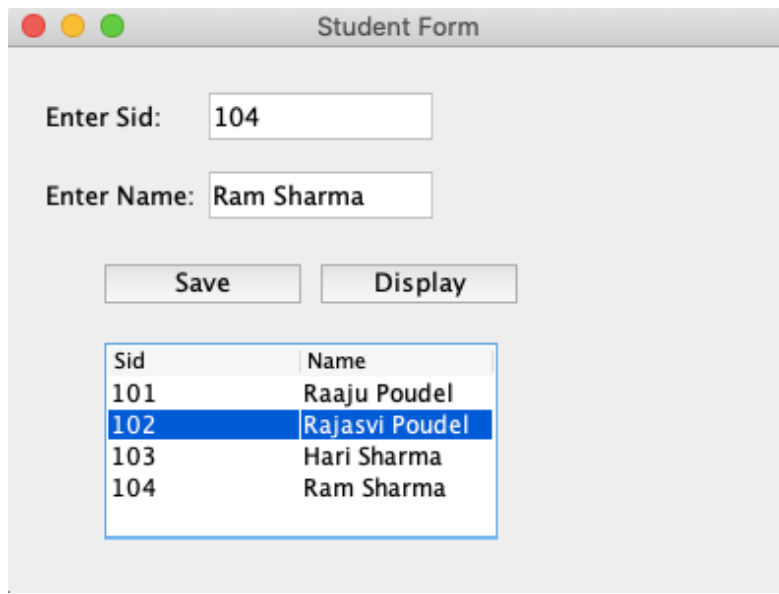
Example.java

```

//Driver Class
public class Example {
    public static void main(String[] args) {
        StudentController cont=new StudentController();
        cont.initController();
    }
}

```

Example -3 (Using JTable for Displaying Multiple Rows)



Sid	Name
101	Raaju Poudel
102	Rajasvi Poudel
103	Hari Sharma
104	Ram Sharma

StudentView.java

```
import javax.swing.*;
import javax.swing.table.*;
public class StudentView {
    public JLabel lbl1, lbl2, lbl3;
    public JTextField txt1, txt2;
    public JButton btn1, btn2;
    //required for creating a empty list
    DefaultTableModel tmodel;

    public StudentView() {
        JFrame jf=new JFrame("Student Form");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        lbl1=new JLabel("Enter Sid:");
        lbl1.setSize(100, 30);
        lbl1.setLocation(20, 20);
        jf.add(lbl1);

        txt1=new JTextField();
        txt1.setSize(120, 30);
        txt1.setLocation(100, 20);
        jf.add(txt1);

        lbl2=new JLabel("Enter Name:");
```

```

        lbl2.setSize(100, 30);
        lbl2.setLocation(20, 60);
        jf.add(lbl2);

        txt2=new JTextField();
        txt2.setSize(120, 30);
        txt2.setLocation(100, 60);
        jf.add(txt2);

        btn1=new JButton("Save");
        btn1.setSize(100, 20);
        btn1.setLocation(50, 110);
        jf.add(btn1);

        btn2=new JButton("Display");
        btn2.setSize(100, 20);
        btn2.setLocation(160, 110);
        jf.add(btn2);

        //creating empty table with default table model
        String cols[]= {"Sid","Name"};
        tmodel=new DefaultTableModel(cols,0); //0 rows
        JTable jt=new JTable(tmodel);
        JScrollPane jp=new JScrollPane(jt);
        jp.setLocation(50, 150);
        jp.setSize(200, 100);
        jf.add(jp);

        jf.setVisible(true);
    }
}

```

StudentModel.java

```

public class StudentModel {
    private int sid;
    private String name;

    public void setId(int sid) {
        this.sid=sid;
    }

    public int getId() {
        return sid;
    }

    public void setName(String name) {
        this.name=name;
    }
}

```

```

        public String getName() {
            return name;
        }
    }
}

```

StudentController.java

```

import java.util.*;
import javax.swing.*;

public class StudentController {
    StudentView v;
    //for storing multiple data
    ArrayList<StudentModel> data;

    public void initController() {
        //initializing ArrayList
        data=new ArrayList<>();
        //initializing view
        v=new StudentView();
        //registering events
        v.btn1.addActionListener(e->saveClicked());
        v.btn2.addActionListener(e->displayClicked());
    }

    public void saveClicked() {
        int sid=Integer.parseInt(v.txt1.getText());
        String name=v.txt2.getText();
        //transferring to model
        StudentModel m=new StudentModel();
        m.setId(sid);
        m.setName(name);
        //adding model to ArrayList
        data.add(m);
        JOptionPane.showMessageDialog(null, "Saved Successfully!");
    }

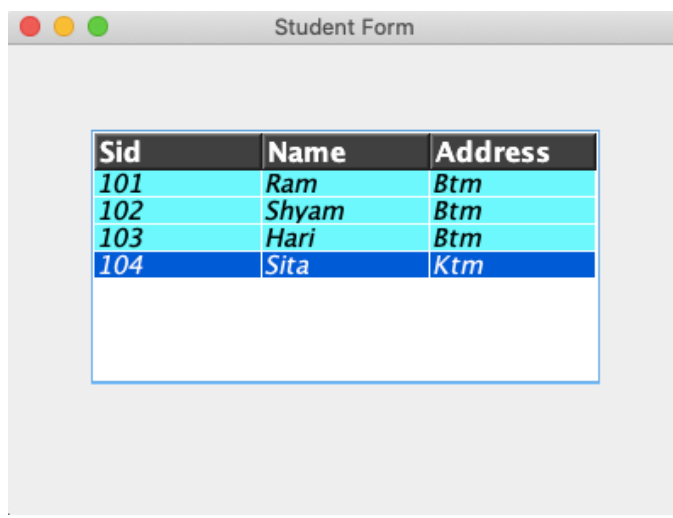
    public void displayClicked() {
        //clearing all rows
        v.tmodel.setRowCount(0);
        //getting data from ArrayList
        for(StudentModel st:data) {
            //putting data in Object
            Object[] obj= {st.getId(),st.getName()};
            v.tmodel.addRow(obj);
        }
    }
}

```


Example.java

```
//Driver Class
public class Example {
    public static void main(String[] args) {
        StudentController cont=new StudentController();
        cont.initController();
    }
}
```

Designing a JTable



```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
//Driver Class
public class Example {
    public static void main(String[] args) {
        JFrame jf=new JFrame("Student Form");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        //creating a JTable
        String cols[]= {"Sid","Name","Address"};
        DefaultTableModel tmodel=new DefaultTableModel(cols,0);
        JTable jt=new JTable(tmodel);
        tmodel.addRow(new Object[] {101,"Ram","Btm"});
        tmodel.addRow(new Object[] {102,"Shyam","Btm"});
        tmodel.addRow(new Object[] {103,"Hari","Btm"});
        tmodel.addRow(new Object[] {104,"Sita","Ktm"});
    }
}
```

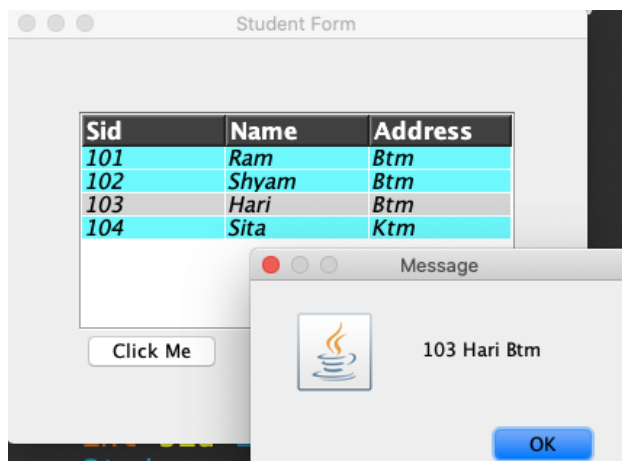
```

//designing JTable
//designing headings
jt.getTableHeader().setBackground(Color.DARK_GRAY);
jt.getTableHeader().setForeground(Color.WHITE);
jt.getTableHeader().setFont(new Font(null,Font.BOLD,16));
//designing rows
jt.setBackground(Color.CYAN);
jt.setFont(new Font("Consolas",Font.ITALIC,15));
JScrollPane jp=new JScrollPane(jt);
jp.setSize(300, 150);
jp.setLocation(50, 50);
jf.add(jp);

jf.setVisible(true);
}
}

```

Getting a Selected Row From JTable



```

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
//Driver Class
public class Example {
    public static void main(String[] args) {
        JFrame jf=new JFrame("Student Form");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);

        //creating a JTable
        String cols[]= {"Sid","Name","Address"};
        DefaultTableModel tmodel=new DefaultTableModel(cols,0);
    }
}

```

```

JTable jt=new JTable(tmodel);
tmodel.addRow(new Object[] {101,"Ram","Btm"});
tmodel.addRow(new Object[] {102,"Shyam","Btm"});
tmodel.addRow(new Object[] {103,"Hari","Btm"});
tmodel.addRow(new Object[] {104,"Sita","Ktm"});
//designing JTable
//designing headings
jt.getTableHeader().setBackground(Color.DARK_GRAY);
jt.getTableHeader().setForeground(Color.WHITE);
jt.getTableHeader().setFont(new Font(null,Font.BOLD,16));
//designing rows
jt.setBackground(Color.CYAN);
jt.setFont(new Font("Consolas",Font.ITALIC,15));
JScrollPane jp=new JScrollPane(jt);
jp.setSize(300, 150);
jp.setLocation(50, 50);
jf.add(jp);

JButton btn=new JButton("Click Me");
btn.setSize(100, 30);
btn.setLocation(50, 200);
jf.add(btn);

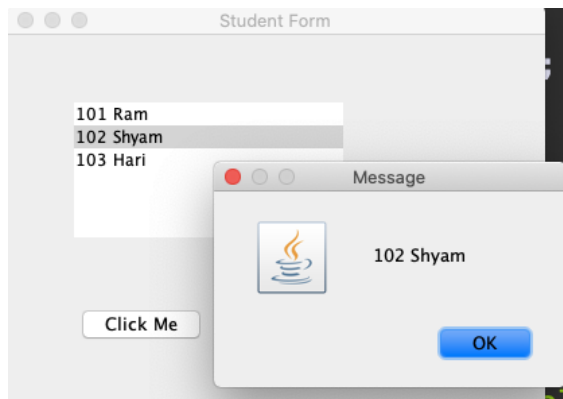
btn.addActionListener(e->{
    //getting row number
    int row=jt.getSelectedRow();
    if(row!=-1) {
        JOptionPane.showMessageDialog(null, "Please Select
            Row!");
    }else {
        //getting data
        int sid=Integer.parseInt(tmodel.getValueAt(row, 0)
            .toString());
        String name=tmodel.getValueAt(row, 1).toString();
        String address=tmodel.getValueAt(row, 2).toString();
        JOptionPane.showMessageDialog(null, sid+" "+name+"
            "+address);
    }

});

jf.setVisible(true);
}
}

```

Getting a Selected Row from JList



```
import java.awt.*;
import javax.swing.*;
//Driver Class
public class Example {
    public static void main(String[] args) {
        JFrame jf=new JFrame("Student Form");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setLocationRelativeTo(null);
        DefaultListModel lmodel=new DefaultListModel();
        JList jl=new JList(lmodel);
        lmodel.addElement("101 Ram");
        lmodel.addElement("102 Shyam");
        lmodel.addElement("103 Hari");
        jl.setSize(200, 100);
        jl.setLocation(50, 50);
        jf.add(jl);
        JButton btn=new JButton("Click Me");
        btn.setSize(100, 30);
        btn.setLocation(50, 200);
        jf.add(btn);

        btn.addActionListener(e->{
            int index=jl.getSelectedIndex();
            if(index!=-1) {
                JOptionPane.showMessageDialog(null, "Please select item!");
            }else {
                String data=jl.getSelectedValue().toString();
                JOptionPane.showMessageDialog(null, data);
            }
        });
        jf.setVisible(true);
    }
}
```

Layout Managers

The layout will specify the format or the order in which the components have got to be placed on the container.

Layout Manager may be a class or component that's responsible to rearrange the components on the container consistent with the required layout. A layout manager automatically arranges your controls within a window by using some sort of algorithm.

Each Container object features a layout manager related to it. A layout manager is an instance of any class that implements the LayoutManager interface. **The layout manager is about by the `setLayout()` method.** If no call to `setLayout()` is formed, then the default layout manager is employed. Whenever a container is resized (or sized for the primary time), the layout manager is employed to position each of the components within it.

The `setLayout()` method has the subsequent general form: **`void setLayout(LayoutManager layoutObj)`**

Here, `layoutObj` may be a regard to the specified layout manager. If you would like to disable the layout manager and position components manually, pass null for `layoutObj`. If you are doing this, you'll get to determine the form and position of every component manually, using the `setBounds()` method defined by Component.

Types of Layout Managers

AWT package provides following types of Layout Managers:

1. Flow Layout
2. Border Layout
3. Card Layout
4. Grid Layout
5. GridBag Layout

Swing provides following types of Layout Managers:

1. Box Layout
2. Group Layout
3. Spring Layout

Flow Layout

This layout will display the components in sequence from left to right, from top to bottom. The components will always be displayed in first-line and if the first line is filled, these components displayed to the next line automatically.

In this Layout Manager, initially, the container assumes as 1 row and 1 column of the window. Depends on the number of components and size of the window, the number of rows and columns count is decided dynamically.

Note: If the row contains only one component then the component is aligned in the center position of that row.

Creation of Flow Layout

`FlowLayout f1 = new FlowLayout();`

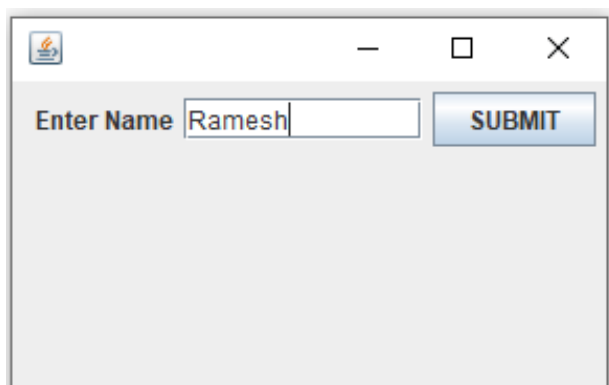
`FlowLayout f1 = new FlowLayout(int align);`

`FlowLayout f1 = new FlowLayout(int align, int hgap, int vgap);`

Example to demonstrate FlowLayout

```
import java.awt.*;
import javax.swing.*;
public class FlowLayoutDemo
{
    JFrame f;
    FlowLayoutDemo ()
    {
        f = new JFrame ();
        JLabel l1 = new JLabel ("Enter Name");
        JTextField tf1 = new JTextField (10);
        JButton b1 = new JButton ("SUBMIT");
        f.add (l1);
        f.add (tf1);
        f.add (b1);
        f.setLayout (new FlowLayout (FlowLayout.RIGHT));

        //setting flow layout of right alignment
        f.setSize (300, 300);
        f.setVisible (true);
    }
    public static void main (String[] args)
    {
        new FlowLayoutDemo ();
    }
}
```



Border Layout

This layout will display the components along the border of the container. This layout contains five locations where the component can be displayed. Locations are North, South, East, west, and Center.

The default region is the center. The above regions are the predefined static constants belongs to the BorderLayout class. Whenever if other regions' spaces are not in use automatically container selects as a center region default and component occupies surrounding region's spaces of the window and that damages look and feel of the user interface.

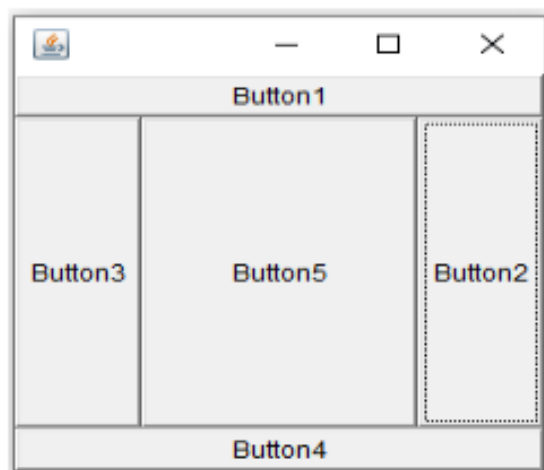
Creation of BorderLayout

BorderLayout bl = new BorderLayout();

BorderLayout bl = new BorderLayout(int vgap, int hgap);

Example of Border Layout

```
import java.awt.*;
public class BorderLayoutDemo
{
    public static void main (String[]args)
    {
        Frame f1 = new Frame ();
        f1.setSize (250, 250);
        Button b1 = new Button ("Button1");
        Button b2 = new Button ("Button2");
        Button b3 = new Button ("Button3");
        Button b4 = new Button ("Button4");
        Button b5 = new Button ("Button5");
        f1.add (b1, BorderLayout.NORTH);
        f1.add (b2, BorderLayout.EAST);
        f1.add (b3, BorderLayout.WEST);
        f1.add (b4, BorderLayout.SOUTH);
        f1.add (b5);
        f1.setVisible (true);
    }
}
```



Grid Layout

The layout will display the components in the format of rows and columns statically. The container will be divided into a table of rows and columns.

The intersection of a row and column cell and every cell contains only one component and all the cells are of equal size. According to Grid Layout Manager, the grid cannot be empty.

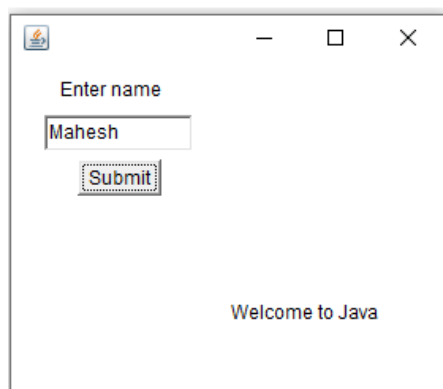
Creation of Grid Layout Manager

GridLayout gl = new GridLayout(int rows, int cols);

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

Example of Grid Layout

```
import java.awt.*;
import javax.swing.*;
public class GridLayoutDemo
{
    public static void main (String[]args)
    {
        Frame f1 = new Frame ();
        f1.setSize (250, 250);
        GridLayout ob = new GridLayout (2, 2);
        f1.setLayout (ob);
        Panel p1 = new Panel ();
        Label l1 = new Label ("Enter name");
        TextField tf = new TextField (10);
        Button b1 = new Button ("Submit");
        p1.add (l1);
        p1.add (tf);
        p1.add (b1);
        f1.add (p1);
        Panel p2 = new Panel ();
        f1.add (p2);
        Panel p3 = new Panel ();
        f1.add (p3);
        Label l2 = new Label ("Welcome to Java");
        f1.add (l2);
        f1.setVisible (true);
    }
}
```



Grid Bag Layout

The `GridBagLayout` lays out components in a grid of cells arranged in rows and columns similar to the `GridLayout`.

- All cells of the grid created by `GridBagLayout` do not have to be of the same size.
- A component does not have to be placed exactly in one cell.
- A component can span multiple cells horizontally as well as vertically.
- We can specify how a component inside its cell should be aligned.

The `GridBagLayout` and `GridBagConstraints` classes are used together. Both classes are in the `java.awt` package.

- `GridBagLayout` class defines a `GridBagLayout` layout manager.
- `GridBagConstraints` class defines constraints for a component in a `GridBagLayout`.

The following code creates an object of the `GridBagLayout` class and sets it as the layout manager for a `JPanel`:

```
JPanel panel = new JPanel();
GridBagLayout gridBagLayout = new GridBagLayout();
panel.setLayout(gridBagLayout);
```

GridBagConstraints

First, create a constraint object

```
GridBagConstraints gbc = new GridBagConstraints();
```

Then, set `gridx` and `gridy` properties in the constraint object

```
gbc.gridx = 0;
```

```
gbc.gridy = 0;
```

After that, add a `JButton` and pass the constraint object as the second argument to the `add()` method.

```
container.add(new JButton("B1"), gbc);
```

We can change the `gridx` property to 1. The `gridy` property remains as 0 as set previously.

```
gbc.gridx = 1;
```

Then, add another `JButton` to the container

```
container.add(new JButton("B2"), gbc);
```

Example:

```
import java.awt.*;
import javax.swing.*;
//Driver Class
public class Example {
    public static void main(String[] args) {
```

```

JFrame jf = new JFrame("Grid Bag Example");
jf.setSize(300, 200);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jf.setLocationRelativeTo(null);

Container cp = jf.getContentPane();
cp.setLayout(new GridBagLayout());

//creating buttons
JButton btn1=new JButton("Button 1");
JButton btn2=new JButton("Button 2");
JButton btn3=new JButton("Button 3");

//adding constraints
GridBagConstraints gbc=new GridBagConstraints();
gbc.gridx=1;
gbc.gridy=2;
cp.add(btn1,gbc);

gbc.gridx=3;
gbc.gridy=1;
cp.add(btn2,gbc);

gbc.gridx=0;
gbc.gridy=0;
cp.add(btn3,gbc);

jf.setVisible(true);
}
}

```



Card Layout

A card layout represents a stack of cards displayed on a container. At a time only one card can be visible and each can contain the only component.

Creation of Card Layout

```
CardLayout cl = new CardLayout();
```

```
CardLayout cl = new CardLayout(int hgap, int vgap);
```

To add the components in CardLayout we use add method:

```
add("Cardname", Component);
```

Methods of CardLayout

1. **first(Container)**: It is used to flip to the first card of the given container.
2. **last(Container)**: It is used to flip to the last card of the given container.
3. **next(Container)**: It is used to flip to the next card of the given container.
4. **previous(Container)**: It is used to flip to the previous card of the given container.
5. **show(Container, cardname)**: It is used to flip to the specified card with the given name.

Example of Card Layout

```
import java.awt.*;
import javax.swing.*;
//Driver Class
public class Example {
    public static void main(String[] args) {
        JFrame jf=new JFrame("My Frame");
        jf.setSize(400, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLocationRelativeTo(null);
        jf.setLayout(null);

        //creating jpanel to demonstrate cardlayout
        JPanel jp=new JPanel();
        jp.setSize(200, 80);
        jp.setLocation(80, 40);
        jp.setBackground(Color.CYAN);
        //adding cardlayout
        CardLayout card=new CardLayout(40,30);
        jp.setLayout(card);
        JLabel lbl1=new JLabel("Label 1");
        JLabel lbl2=new JLabel("Label 2");
        JLabel lbl3=new JLabel("Label 3");
        //adding level to cards
        jp.add("card1",lbl1);
        jp.add("card2",lbl2);
        jp.add("card3",lbl3);
```

```

//creating another JPanel for buttons
JPanel jp1=new JPanel();
    jp1.setSize(300, 100);
    jp1.setLocation(40, 120);
    jp1.setLayout(new FlowLayout());

    JButton prev=new JButton("Previous");
    JButton next=new JButton("Next");
    JButton first=new JButton("First");
    JButton last=new JButton("Last");
    JButton show=new JButton("Show Card");
    jp1.add(prev);
    jp1.add(next);
    jp1.add(first);
    jp1.add(last);
    jp1.add(show);

    //adding panels to frame
    jf.add(jp);
    jf.add(jp1);

    //adding click event to buttons
    prev.addActionListener(e->{
        //previous
        card.previous(jp);
    });

    next.addActionListener(e->{
        //next
        card.next(jp);
    });

    first.addActionListener(e->{
        card.first(jp);
    });

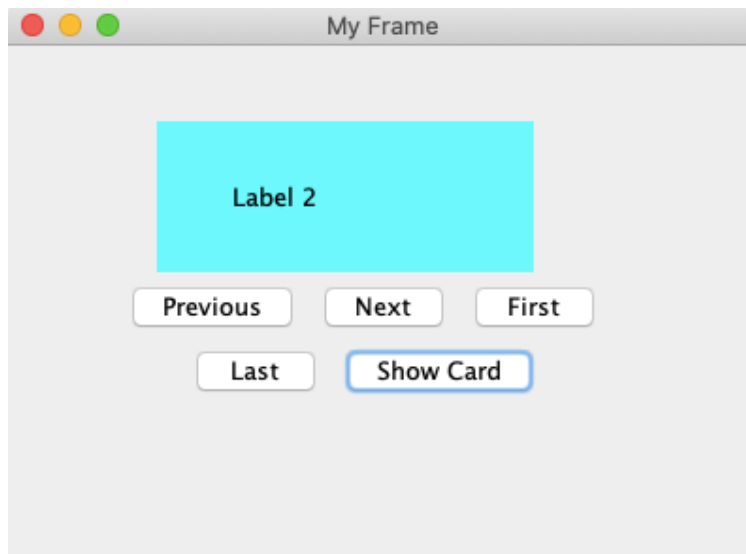
    last.addActionListener(e->{
        card.last(jp);
    });

    show.addActionListener(e->{
        card.show(jp, "card2");
    });

    jf.setVisible(true);
}
}

```

Output:



Box Layout

The Java `BoxLayout` class is used to arrange the components either vertically or horizontally.

For this purpose, the `BoxLayout` class provides following constants. They are as follows:

1. **`public static final int X_AXIS`**: Alignment of the components are horizontal from left to right.
2. **`public static final int Y_AXIS`**: Alignment of the components are vertical from top to bottom.

Box Layout is created as follows:

`BoxLayout(Container c, int axis)`: creates a box layout that arranges the components with the given axis.

Example of Box Layout

```
import java.awt.*;
import javax.swing.*;
//Driver Class
public class Example {
    public static void main(String[] args) {
        JFrame jf=new JFrame("My Frame");
        jf.setSize(300, 250);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLocationRelativeTo(null);
        //setting box layout
        Container c=jf.getContentPane();
        jf.setLayout(new BoxLayout(c,BoxLayout.Y_AXIS));
        //arranges top to bottom
    }
}
```

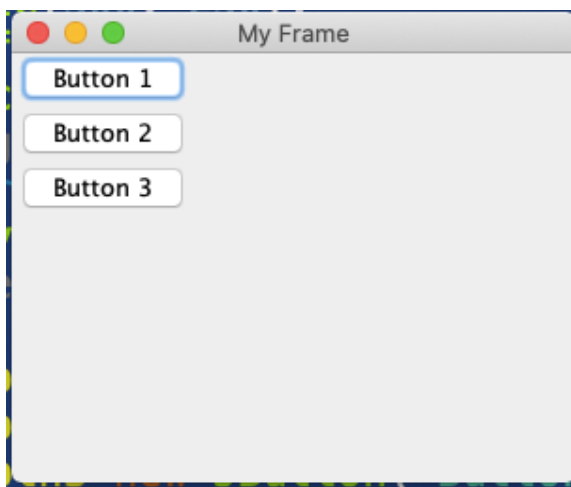
```

        JButton btn1=new JButton("Button 1");
        JButton btn2=new JButton("Button 2");
        JButton btn3=new JButton("Button 3");

        jf.add(btn1);
        jf.add(btn2);
        jf.add(btn3);

        jf.setVisible(true);
    }
}

```



Group Layout

GroupLayout groups its components and place them hierarchically in a container. The grouping is done by an instance of the **Group** class.

Group is an abstract class and two concrete classes that implement this class are **SequentialGroup** and **ParallelGroup**.

- **SequentialGroup** positions its elements sequentially one after the other while
- **ParallelGroup** aligns its elements on top of each other.

The **GroupLayout** class provides methods such as **createParallelGroup()** and **createSequentialGroup()** to create groups.

GroupLayout treats each axis independently. In other words, there is a group representing the horizontal axis and a group representing the vertical axis. Each component must exist in a horizontal and vertical group, otherwise, an **IllegalStateException** is thrown during layout, or when the minimum, preferred or maximum size is requested.

Creation of Group Layout:

GroupLayout(Container Host): For the specified Container, it creates a GroupLayout.

```
JPanel grpPanel = new JPanel();
GroupLayout grpLayout = new GroupLayout(grpPanel);
```

Example of Group Layout

```
import java.awt.*;
import javax.swing.*;
//Driver Class
public class Example {
    public static void main(String[] args) {
        JFrame jf = new JFrame("GroupLayout");
        jf.setSize(300, 200);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLocationRelativeTo(null);

        //adding group layout
        Container c = jf.getContentPane();
        GroupLayout grp = new GroupLayout(c);

        c.setLayout(grp);

        JButton btn1 = new JButton("Button 1");
        JButton btn2 = new JButton("Button 2");

        grp.setHorizontalGroup(grp.createSequentialGroup()
            .addComponent(btn1)
            .addComponent(btn2)
        );

        grp.setVerticalGroup(grp.createParallelGroup()
            .addComponent(btn1)
            .addComponent(btn2)
        );

        jf.setVisible(true);
    }
}
```



Adding gap between components:

```
grp.setHorizontalGroup(grp.createSequentialGroup()  
    .addComponent(btn1)  
    .addGap(50)  
    .addComponent(btn2)  
    );
```

Spring Layout

A **SpringLayout** arranges the children of its associated container according to a set of constraints. Constraints are nothing but horizontal and vertical distance between two-component edges. Every constraint is represented by a `SpringLayout.Constraint` object. Each child of a `SpringLayout` container, as well as the container itself, has exactly one set of constraints associated with them.

Each edge position is dependent on the position of the other edge. If a constraint is added to create a new edge, then the previous binding is discarded. `SpringLayout` doesn't automatically set the location of the components it manages.

Example of Spring Layout:

```
import java.awt.*;  
import javax.swing.*;  
//Driver Class  
public class Example {  
    public static void main(String[] args) {  
        JFrame jf = new JFrame("Spring Layout Example");  
        jf.setSize(300, 200);  
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jf.setLocationRelativeTo(null);  
  
        Container cp = jf.getContentPane();  
        //adding spring layout  
        SpringLayout spl=new SpringLayout();  
        cp.setLayout(spl);  
  
        //creating buttons
```



```

JButton btn1=new JButton("Button 1");
JButton btn2=new JButton("Button 2");
JButton btn3=new JButton("Button 3");
cp.add(btn1);
cp.add(btn2);
cp.add(btn3);

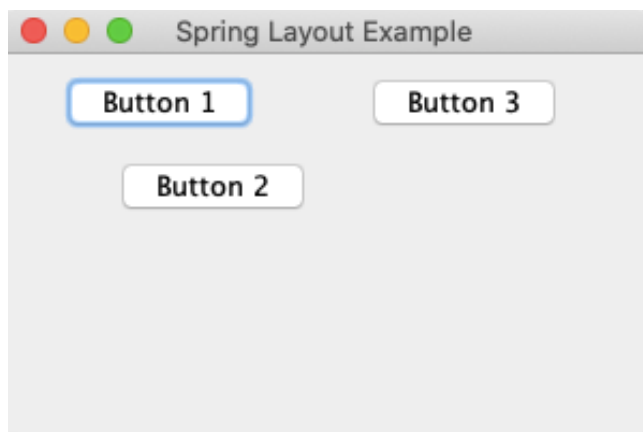
//adding constraints
//for button 1
spl.putConstraint(SpringLayout.WEST, btn1,
                  24, SpringLayout.WEST, cp);
spl.putConstraint(SpringLayout.NORTH, btn1,
                  9, SpringLayout.NORTH, cp);

//for button 2
spl.putConstraint(SpringLayout.WEST, btn2,
                  49, SpringLayout.WEST, cp);
spl.putConstraint(SpringLayout.NORTH, btn2,
                  10, SpringLayout.SOUTH, btn1);

//for button 3
spl.putConstraint(SpringLayout.WEST, btn3,
                  45, SpringLayout.EAST, btn1);
spl.putConstraint(SpringLayout.NORTH, btn3,
                  9, SpringLayout.NORTH, cp);

        jf.setVisible(true);
    }
}

```



Example 2:

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

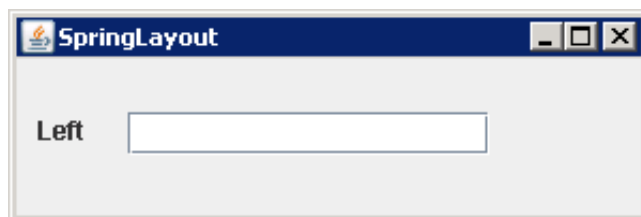
public class SpringSample {
    public static void main(String args[]) {
        JFrame frame = new JFrame("SpringLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = frame.getContentPane();

        SpringLayout layout = new SpringLayout();
        contentPane.setLayout(layout);

        JLabel left = new JLabel("Left");
        JTextField right = new JTextField(15);

        contentPane.add(left);
        contentPane.add(right);
        layout.putConstraint(SpringLayout.WEST, left, 10,
            SpringLayout.WEST, contentPane);
        layout.putConstraint(SpringLayout.NORTH, left, 25,
            SpringLayout.NORTH, contentPane);
        layout.putConstraint(SpringLayout.NORTH, right, 25,
            SpringLayout.NORTH, contentPane);
        layout.putConstraint(SpringLayout.WEST, right, 20,
            SpringLayout.EAST, left);

        frame.setSize(300, 100);
        frame.setVisible(true);
    }
}
```



Dialogs:

A **JDialog** is a top-level Swing container to host components and display a dialog. Creating a dialog window is very simple: just create a new class that inherits from the **JDialog** class. By default, a **JDialog** uses a **BorderLayout** as the layout manager.

Based on focus behavior of a **JDialog**, it can be categorized as

- Modal
- Modeless

When a modal **JDialog** is displayed, it blocks other displayed windows in the application. To make a **JDialog** modal, we can use its **setModal(true)** method.

A modeless **JDialog** does not block any other displayed windows in the application. By default, a **JDialog** is modeless.

Standard Dialogs

The **JOptionPane** class makes it easy for we to create and show standard modal dialogs. It contains many static methods to create different kinds of **JDialog**, fill them with details, and show them as a modal **JDialog**. When a **JDialog** is closed, the method returns a value to indicate the user's action on the **JDialog**.

We can display the following four kinds of standard dialogs:

- Message Dialog
- Confirmation Dialog
- Input Dialog
- Option Dialog

All four types of standard dialogs accept different types of arguments and return different types of values. The following table shows the list of arguments of these methods and their descriptions.

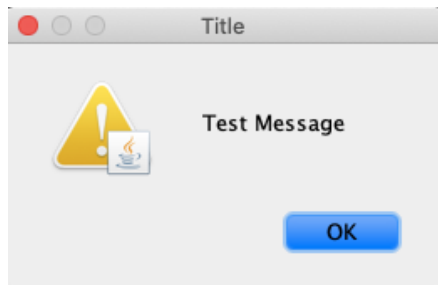
SN	Argument/Description
1	Component parentComponent The JDialog is centered on the parent component. If it is null, the JDialog is centered on the screen.
2	Object message Typically, it is a string. It can also display Swing component, Icon . If we pass any other object, the toString() method is called on that object and the returned string is displayed. We can also pass an array of objects and each element of the array will be displayed vertically one after another.

3	<p><code>int messageType</code> Set the type of the message. Depending on the type of message, a suitable icon is displayed in the dialog box. The available message types are:</p> <ul style="list-style-type: none"> • <code>ERROR_MESSAGE</code> • <code>INFORMATION_MESSAGE</code> • <code>WARNING_MESSAGE</code> • <code>QUESTION_MESSAGE</code> • <code>PLAIN_MESSAGE</code> <p><code>PLAIN_MESSAGE</code> type does not display any icon.</p>
4	<p><code>int optionType</code> Specify the buttons on the dialog box. We can use the following options.</p> <ul style="list-style-type: none"> • <code>DEFAULT_OPTION</code> • <code>YES_NO_OPTION</code> • <code>YES_NO_CANCEL_OPTION</code> • <code>OK_CANCEL_OPTION</code> <p>The <code>DEFAULT_OPTION</code> displays an OK button. Other options display a set of buttons, as their names suggest. We can customize the number of buttons and text by supplying the options arguments to the <code>showOptionDialog()</code> method.</p>
5	<p><code>Object[] options</code> Customize buttons on a dialog box. If we pass a <code>Component</code> object in the array, that component is displayed in the row of buttons. If we specify an <code>Icon</code> object, the icon is displayed in a <code>JButton</code>. Typically, we pass an array of strings as this argument to display a custom set of buttons in the dialog box.</p>
6	<p><code>String title</code> title of the dialog box. A default title is supplied if we do not pass this argument.</p>
7	<p><code>Object initialValue</code> Set the initial value that is displayed in the input dialog.</p>

Message Dialog

A message dialog shows information with the OK button. The method does not return any value.

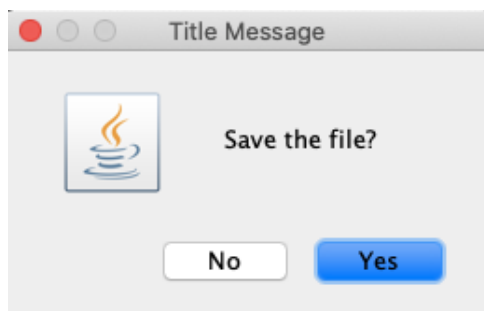
```
import javax.swing.*;
public class Example {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Test Message",
            "Title", JOptionPane.WARNING_MESSAGE);
    }
}
```



Confirm Dialog

We can display a confirmation dialog box by using the `showConfirmDialog()` method. The user's response is indicated by the return value.

```
import javax.swing.*;
public class Example {
    public static void main(String[] args) {
        int res = JOptionPane.showConfirmDialog(null,
            "Save the file?",
            "Title Message",
            JOptionPane.YES_NO_OPTION,
            //JOptionPane.YES_NO_OPTION
            //JOptionPane.OK_CANCEL_OPTION
            JOptionPane.QUESTION_MESSAGE);
        System.out.println(res);
        //yes/ok=0, no=1, cancel=2
    }
}
```

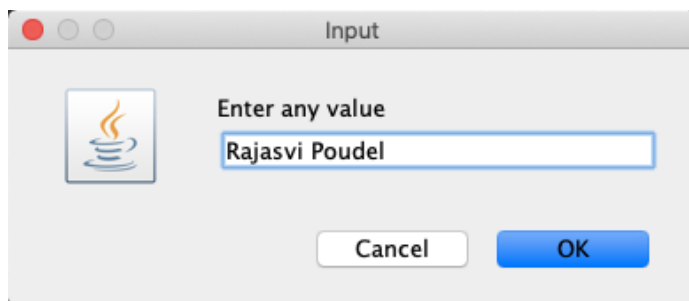


Input Dialog

We can ask the user for an input using the `showInputDialog()` method and specify an initial value for the user's input.

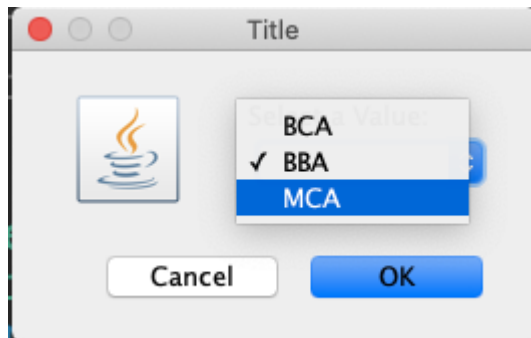
If we want the user to select a value from a list, we can pass an object array that contains the list. The UI will display the list in a suitable component such as a `JComboBox` or a `JList`.

```
import javax.swing.*;
public class Example {
    public static void main(String[] args) {
        String res=JOptionPane.showInputDialog(null,
            "Enter any value");
        System.out.println(res);
    }
}
```



Example 2:

```
import javax.swing.*;
public class Example {
    public static void main(String[] args) {
        /*Object res = JOptionPane.showInputDialog(parentComponent,
            message,
            title,
            messageType,
            icon,
            selectionValues,
            initialSelectionValue);
        */
        Object res = JOptionPane.showInputDialog(null,
            "Select a Value:",
            "Title",
            JOptionPane.INFORMATION_MESSAGE,
            null,
            new String[] {"BCA","BBA","MCA"},
            "BBA");
        System.out.println(res);
    }
}
```



Option Dialog

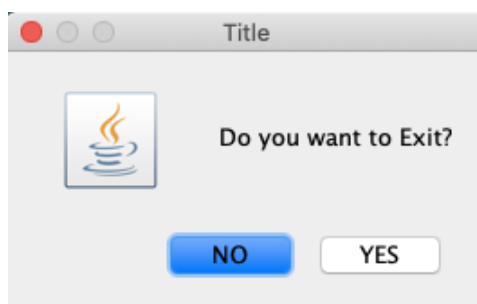
We can customize the option buttons using the `showOptionDialog()` method that is declared as follows:

```
int showOptionDialog(Component parentComponent,
                    Object message,
                    String title,
                    int optionType,
                    int messageType,
                    Icon icon,
                    Object[] options,
                    Object initialValue);
```

Example:

```
import javax.swing.*;
public class Example {
    public static void main(String[] args) {
        int res = JOptionPane.showOptionDialog(null,
            "Do you want to Exit?",
            "Title",
            JOptionPane.DEFAULT_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null,
            new String[] {"YES", "NO"},
            "NO");

        System.out.println(res);
    }
}
```



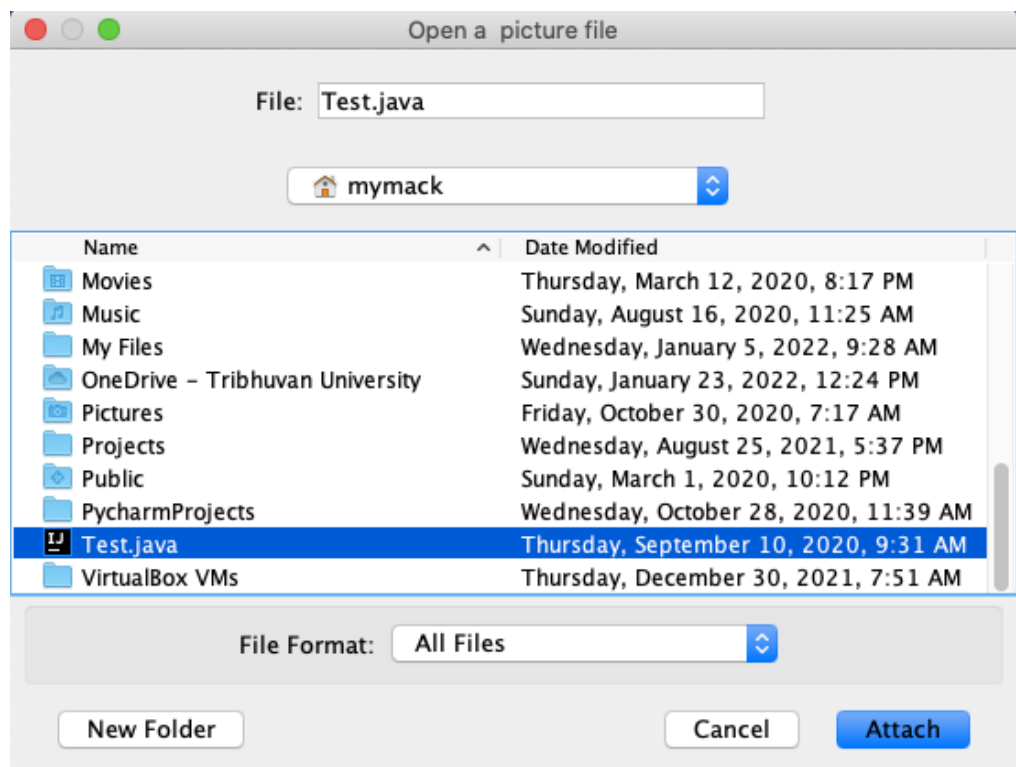
JFileChooser Dialog

A JFileChooser lets the user select a file from the file system. We can create an object of the JFileChooser class. It allows the user to choose only files, only directories, or both.

Example:

```
import javax.swing.*;
import java.io.*;
public class Example {
    public static void main(String[] args) {
        //display file chooser
        JFileChooser fch=new JFileChooser();
        fch.setDialogTitle("Open a picture file");
        int res = fch.showDialog(null,"Attach");

        if (res == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fch.getSelectedFile();
            System.out.println("we selected: " + selectedFile);
        }
    }
}
```

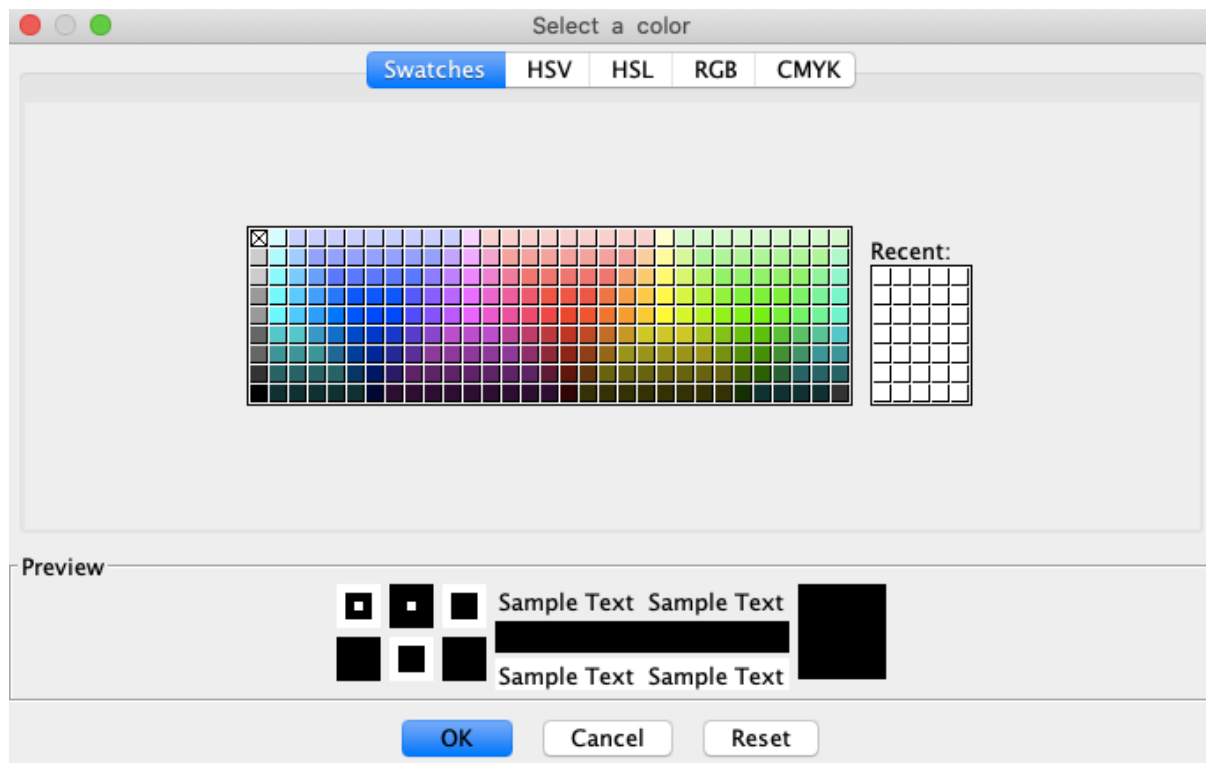


JColorChooser Dialog

A JColorChooser is a Swing component that lets we choose a color graphically in a JDialog.

```
import javax.swing.*;
import java.awt.*;
public class Example {
    public static void main(String[] args) {
        // Display a color chooser dialog
        Color color = JColorChooser.showDialog(null,
            "Select a color", Color.BLACK);
        //parent,message,initial color

        System.out.println("Selected Color: "+color);
    }
}
```



Custom JDialog

We can create custom JDialog as per our requirement and place different components on it as follows:

```
JDialog dialog=new JDialog();
```

Example:

```
import javax.swing.*;
public class Example {
    public static void main(String[] args) {
        JDialog jd=new JDialog();
        jd.setSize(200, 150);
        jd.setModal(true);
        jd.setTitle("Custom Dialog");

        JLabel lbl=new JLabel("I am Label");
        jd.add(lbl);

        jd.setVisible(true);
    }
}
```

