Java Awt

Java AWT (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

Thejava.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Characteristics

- It is a set of native user interface components.
- It is very robust in nature.
- It includes various editing tools like graphics tool and imaging tools.
- It uses native window-system controls.
- It provides functionality to include shapes, colors and font classes.

Advantages

- It takes very less memory for development of GUI and executing programs.
- It is highly stable as it rarely crashes.
- It is dependent on operating system so performance is high.
- It is easy to use for beginners due to its easy interface.

Disadvantages

- The buttons of AWT does not support pictures.
- It is heavyweight in nature.
- Two very important components trees and tables are not present.
- Extensibility is not possible as it is platform dependen

Difference between AWT and Swing

No.	Java AWT	Java Swing
1)	AWT components are platform-dependent.	Java swing components are platform-independent.
2)	AWT components are heavyweight.	Swing components are lightweight.

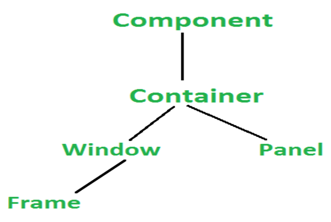
3)	AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.
4)	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT doesn't follows MVC(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC.

Why AWT is platform dependent?

Java AWT calls the native platform calls the native platform (operating systems) subroutine for creating API components like TextField, ChechBox, button, etc.

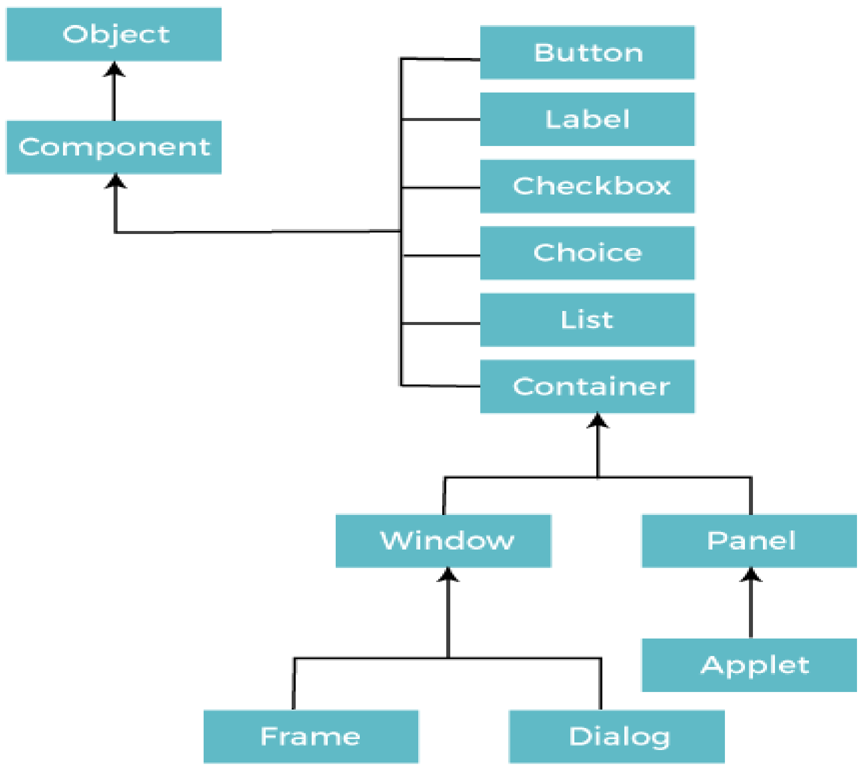
For example, an AWT GUI with components like TextField, label and button will have different look and feel for the different platforms like Windows, MAC OS, and Unix. The reason for this is the platforms have different view for their native components and AWT directly calls the native subroutine that creates those components.

In simple words, an AWT application will look like a windows application in Windows OS whereas it will look like a Mac application in the MAC OS.



Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Components

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

Container

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

It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

Note: A container itself is a component (see the above diagram), therefore we can add a container inside container.

Types of containers:

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

Panel

The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

Frame

The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

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.

Java AWT Example

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

1. By extending Frame class (inheritance)
2. By creating the object of Frame class (association)

AWT Example by Inheritance

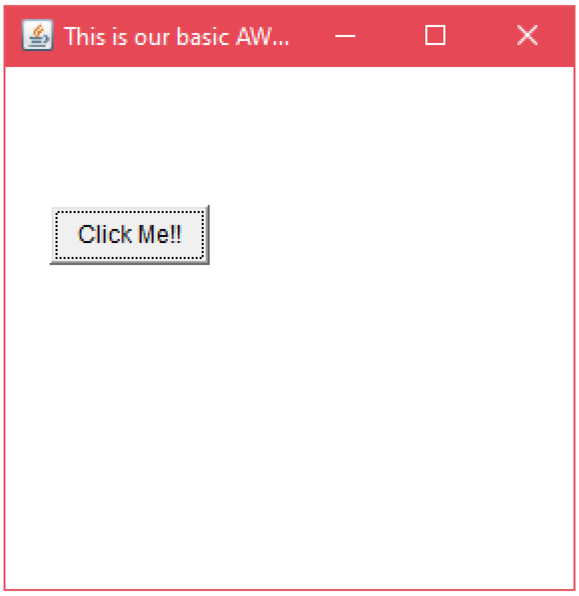
Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

AWTExample1.java

```
// importing Java AWT class
import java.awt.*;

// extending Frame class to our class AWTExample1
public class AWTExample1 extends Frame {

  // initializing using constructor
  AWTExample1() {

    // creating a button
     Button b = new Button("Click Me!!");

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

    // adding button into frame
     add(b);

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

    // setting the title of Frame
```

```
        setTitle("This is our basic AWT example");

        // no layout manager
        setLayout(null);

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

// main method
public static void main(String args[]) {

// creating instance of Frame class
AWTExample1 f = new AWTExample1();

}

}
```

Output:



AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are creating a TextField, Label and Button component on the Frame.

AWTExample2.java

```
// importing Java AWT class
import java.awt.*;
```

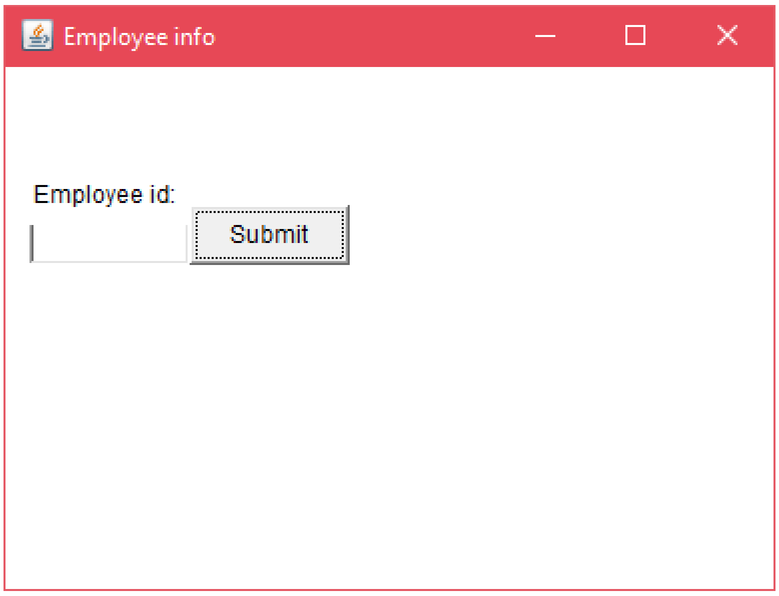
```
// class AWTExample2 directly creates instance of Frame class
class AWTExample2 {

   // initializing using constructor
   AWTExample2() {

      // creating a Frame
      Frame f = new Frame();

      // creating a Label
      Label l = new Label("Employee id:");

      // creating a Button
      Button b = new Button("Submit");

      // creating a TextField
      TextField t = new TextField();

      // setting position of above components in the frame
      l.setBounds(20, 80, 80, 30);
      t.setBounds(20, 100, 80, 30);
      b.setBounds(100, 100, 80, 30);

      // adding components into frame
      f.add(b);
      f.add(l);
      f.add(t);

      // frame size 300 width and 300 height
      f.setSize(400,300);

      // setting the title of frame
      f.setTitle("Employee info");

      // no layout
      f.setLayout(null);

      // setting visibility of frame
      f.setVisible(true);
}

// main method
public static void main(String args[]) {

// creating instance of Frame class
AWTExample2 awt_obj = new AWTExample2();

}

}
```

Java AWT Button

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

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

Output:



Java AWT Label

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

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

1. public class Label extends Component implements Accessible

AWT Label Fields

The java.awt.Component class has following fields:

1. static int LEFT: It specifies that the label should be left justified.
2. static int RIGHT: It specifies that the label should be right justified.
3. static int CENTER: It specifies that the label should be placed in center.

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse
java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener

FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- 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){}

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){

//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);

//register listener
b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

Swing in java

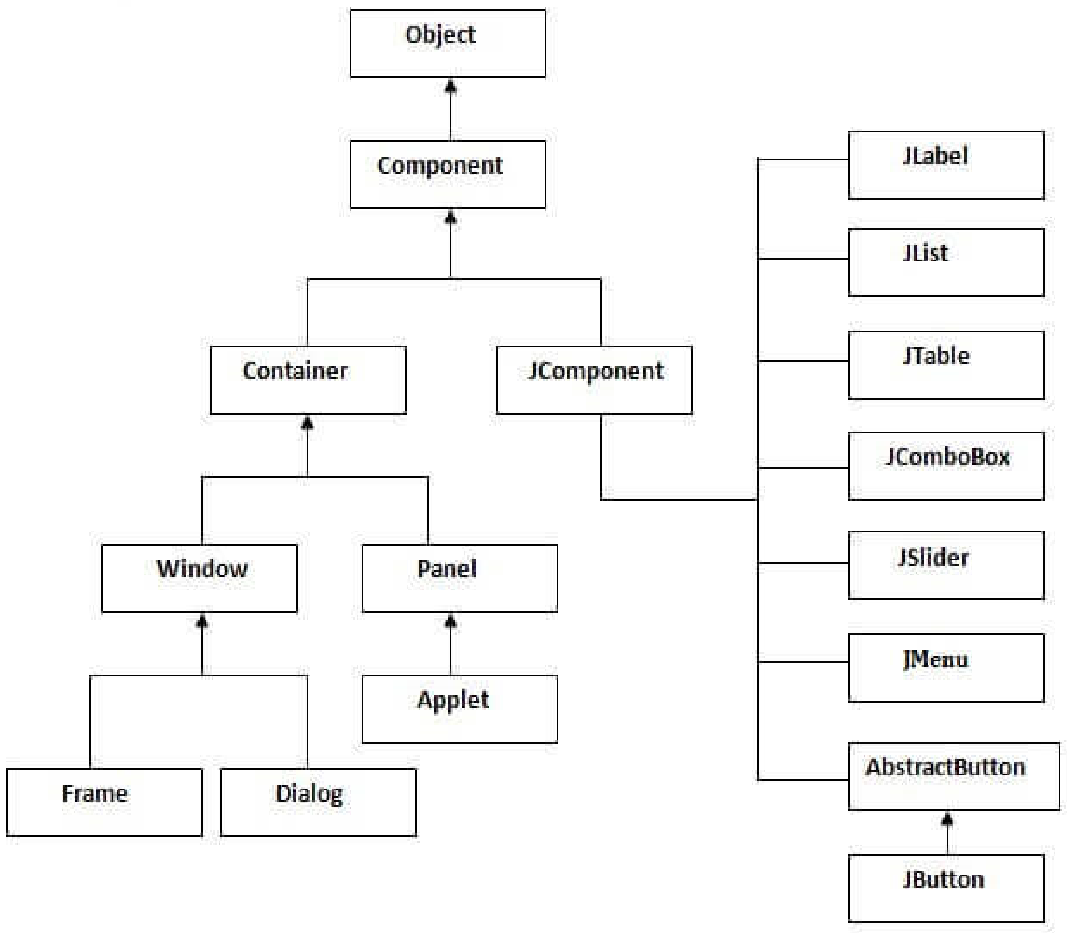
JAVA provides a rich set of libraries to create Graphical User Interface in a platform independent way. In this tutorial, we'll look at SWING GUI controls.

Swing is a part of JFC (Java Foundation Classes). Building Graphical User Interface in Java requires the use of Swings. Swing Framework contains a large set of components that allow a high level of customization and provide rich functionalities and is used to create window-based applications.

Java swing components are lightweight, platform-independent, provide powerful components like tables, scroll panels, buttons, lists, color chooser, etc. In this article, we'll see how to make frames using Swings in Java. Ways to create a frame:

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Methods:

1. By creating the object of Frame class (association)
2. By extending Frame class (inheritance)
3. Create a frame using Swing inside main()

Way 1: By creating the object of Frame class (association)

In this, we will see how to create a JFrame window by instantiating the JFrame class.

```
import javax.swing.*;

public class test1
{
    JFrame frame;

    test1()
    {
        // creating instance of JFrame with name "first way"
        frame=new JFrame("first way");
        // creates instance of JButton
        JButton button = new JButton("let's see");

        button.setBounds(200, 150, 90, 50);

        // setting close operation
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // adds button in JFrame
        frame.add(button);

        // sets 500 width and 600 height
        frame.setSize(500, 600);

        // uses no layout managers
        frame.setLayout(null);

        // makes the frame visible
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        new test1();
    }
}
```

Way 2: By extending Frame class (inheritance)
In this example, we will be inheriting JFrame class to create JFrame window and hence it won't be required to create an instance of JFrame class explicitly.

java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. The Java LayoutManagers facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

1. java.awt.BorderLayout

2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

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

Constructors of BorderLayout class:

- BorderLayout(): creates a border layout but with no gaps between the components.
- BorderLayout(int hgap, int vgap): creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class: Using BorderLayout() constructor

FileName: Border.java

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

public class Border
```

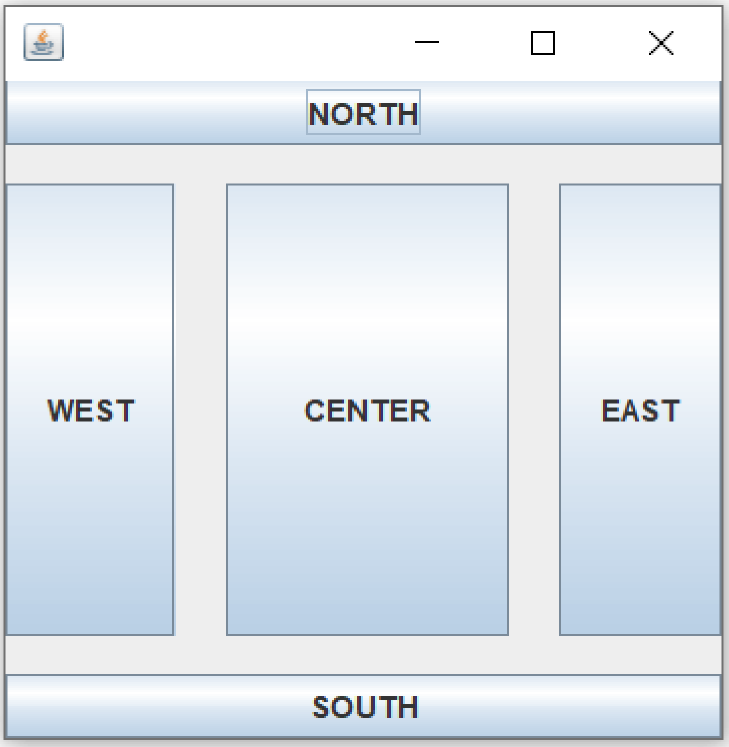
Example of BorderLayout class: Using BorderLayout(int hgap, int vgap) constructor

The following example inserts horizontal and vertical gaps between buttons using the parameterized constructor BorderLayout(int hgap, int gap)

FileName: BorderLayoutExample.java

```
// import statement
import java.awt.*;
import javax.swing.*;
public class BorderLayoutExample
{
JFrame jframe;
// constructor
BorderLayoutExample()
{
   // creating a Frame
   jframe = new JFrame();
   // create buttons
   JButton btn1 = new JButton("NORTH");
   JButton btn2 = new JButton("SOUTH");
   JButton btn3 = new JButton("EAST");
   JButton btn4 = new JButton("WEST");
   JButton btn5 = new JButton("CENTER");
    // creating an object of the BorderLayout class using
    // the parameterized constructor where the horizontal gap is 20
    // and vertical gap is 15. The gap will be evident when buttons are placed
    // in the frame
   jframe.setLayout(new BorderLayout(20, 15));
   jframe.add(btn1, BorderLayout.NORTH);
   jframe.add(btn2, BorderLayout.SOUTH);
   jframe.add(btn3, BorderLayout.EAST);
   jframe.add(btn4, BorderLayout.WEST);
   jframe.add(btn5, BorderLayout.CENTER);
   jframe.setSize(300,300);
   jframe.setVisible(true);
}
// main method
public static void main(String argvs[])
{
   new BorderLayoutExample();
}
}
```

Output:

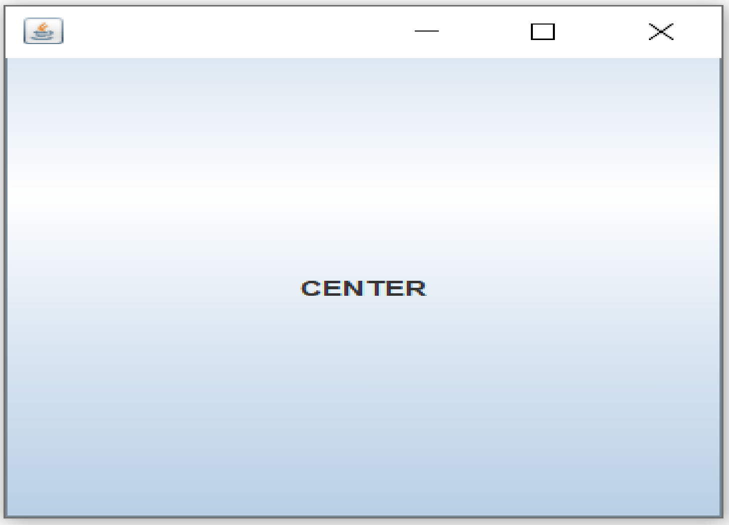


Java BorderLayout: Without Specifying Region

The add() method of the JFrame class can work even when we do not specify the region. In such a case, only the latest component added is shown in the frame, and all the components added previously get discarded. The latest component covers the whole area. The following example shows the same.

FileName: BorderLayoutWithoutRegionExample.java

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

public class BorderLayoutWithoutRegionExample
{
JFrame jframe;

// constructor
BorderLayoutWithoutRegionExample()
{
   jframe = new JFrame();

   JButton btn1 = new JButton("NORTH");
   JButton btn2 = new JButton("SOUTH");
   JButton btn3 = new JButton("EAST");
   JButton btn4 = new JButton("WEST");
   JButton btn5 = new JButton("CENTER");

   // horizontal gap is 7, and the vertical gap is 7
   // Since region is not specified, the gaps are of no use
   jframe.setLayout(new BorderLayout(7, 7));

   // each button covers the whole area
   // however, the btn5 is the latest button
   // that is added to the frame; therefore, btn5
   // is shown
   jframe.add(btn1);
   jframe.add(btn2);
   jframe.add(btn3);
   jframe.add(btn4);
   jframe.add(btn5);

   jframe.setSize(300,300);
   jframe.setVisible(true);
}

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



Output:

```
JFrame f;
Border()
{
   f = new JFrame();

    // creating buttons
   JButton b1 = new JButton("NORTH");; // the button will be labeled as NORTH
   JButton b2 = new JButton("SOUTH");; // the button will be labeled as SO
UTH
   JButton b3 = new JButton("EAST");; // the button will be labeled as EAST
   JButton b4 = new JButton("WEST");; // the button will be labeled as WE
ST
   JButton b5 = new JButton("CENTER");; // the button will be labeled as CENTER

   f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction
   f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Directi
on
   f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direction
   f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Directio
n
   f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center

   f.setSize(300, 300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new Border();
}
}
```

Java GridLayout

The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

1. GridLayout(): creates a grid layout with one column per component in a row.
2. GridLayout(int rows, int columns): creates a grid layout with the given rows and columns but no gaps between the components.
3. GridLayout(int rows, int columns, int hgap, int vgap): creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

Example of GridLayout class: Using GridLayout() Constructor

The GridLayout() constructor creates only one row. The following example shows the usage of the parameterless constructor.

Java FlowLayout

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.

Fields of FlowLayout class

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

Constructors of FlowLayout class

1. FlowLayout(): creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. FlowLayout(int align): creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignment and the given horizontal and vertical gap.

Java BoxLayout

The Java BoxLayout class is used to arrange the components either vertically or horizontally. For this purpose, the BoxLayout class provides four constants. They are as follows:

Note: The BoxLayout class is found in javax.swing package.

Fields of BoxLayout Class

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.
3. public static final int LINE_AXIS: Alignment of the components is similar to the way words are aligned in a line, which is based on the ComponentOrientation property of the container. If the ComponentOrientation property of the container is horizontal, then the components are aligned horizontally; otherwise, the components are aligned vertically. For horizontal orientations, we have two cases: left to right, and right to left. If the value ComponentOrientation property of the container is from left to right, then the components are rendered from left to right, and for right to left, the rendering of components is also from right to left. In the case of vertical orientations, the components are always rendered from top to bottom.
4. public static final int PAGE_AXIS: Alignment of the components is similar to the way text lines are put on a page, which is based on the ComponentOrientation property of the container. If the ComponentOrientation property of the container is horizontal, then components are aligned vertically; otherwise, the components are aligned horizontally. For horizontal orientations, we have two cases: left to right, and right to left. If the value ComponentOrientation property of the container is also from left to right, then the components are rendered from left to right, and for right to left, the rendering of components is from right to left. In the case of vertical orientations, the components are always rendered from top to bottom.

Constructor of BoxLayout class

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

Java CardLayout

The Java CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout Class

1. CardLayout(): creates a card layout with zero horizontal and vertical gap.
2. CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Commonly Used Methods of CardLayout Class

- public void next(Container parent): is used to flip to the next card of the given container.
- public void previous(Container parent): is used to flip to the previous card of the given container.
- public void first(Container parent): is used to flip to the first card of the given container.
- public void last(Container parent): is used to flip to the last card of the given container.
- public void show(Container parent, String name): is used to flip to the specified card with the given name.

Java GridBagLayout

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

The components may not be of the same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of the constraints object, we arrange the component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine the component's size. GridBagLayout components are also arranged in the rectangular grid but can have many different sizes and can occupy multiple rows or columns.

Constructor

GridBagLayout(): The parameterless constructor is used to create a grid bag layout manager.

GridBagLayout Fields

Modifier and Type	Field	Description
double[]	columnWeights	It is used to hold the overrides to the column weights.
int[]	columnWidths	It is used to hold the overrides to the column minimum width.
protected Hashtable<Component,GridBagConstraints>	comptable	It is used to maintains the association between a component and its gridbag constraints.
protected GridBagConstraints	defaultConstraints	It is used to hold a gridbag constraints instance containing the default values.
protected GridBagLayoutInfo	layoutInfo	It is used to hold the layout information for the gridbag.
protected static int	MAXGRIDSIZE	No longer in use just for backward compatibility
protected static int	MINSIZE	It is smallest grid that can be laid out by the grid bag layout.
protected static int	PREFERREDSIZE	It is preferred grid size that can be laid

		out by the grid bag layout.
int[]	rowHeights	It is used to hold the overrides to the row minimum heights.
double[]	rowWeights	It is used to hold the overrides to the row weights.

GridBagLayout Methods

Modifier and Type	Method	Description
void	addLayoutComponent(Component comp, Object constraints)	It adds specified component to the layout, using the specified constraints object.
void	addLayoutComponent(String name, Component comp)	It has no effect, since this layout manager does not use a per-component string.
protected void	adjustForGravity(GridBagConstraints constraints, Rectangle r)	It adjusts the x, y, width, and height fields to the correct values depending on the constraint geometry and pads.
protected void	AdjustForGravity(GridBagConstraints constraints, Rectangle r)	This method is for backwards compatibility only
protected void	arrangeGrid(Container parent)	Lays out the grid.

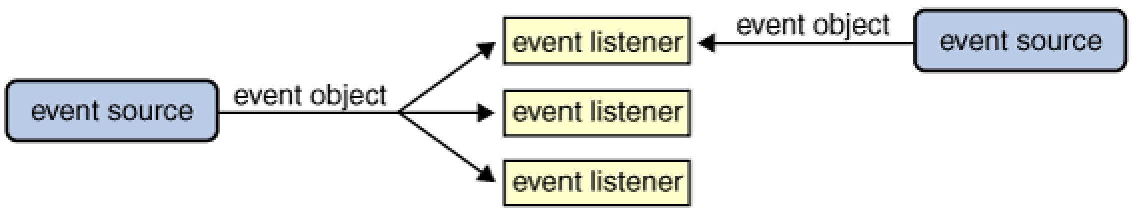
protected void	ArrangeGrid(Container parent)	This method is obsolete and supplied for backwards compatibility
GridBagConstraints	getConstraints(Component comp)	It is for getting the constraints for the specified component.
float	getLayoutAlignmentX(Container parent)	It returns the alignment along the x axis.
float	getLayoutAlignmentY(Container parent)	It returns the alignment along the y axis.
int[][]	getLayoutDimensions()	It determines column widths and row heights for the layout grid.
protected GridBagLayoutInfo	getLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
protected GridBagLayoutInfo	GetLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
Point	getLayoutOrigin()	It determines the origin of the layout area, in the graphics coordinate space of the target container.
double[][]	getLayoutWeights()	It determines the weights of the layout grid's columns and rows.
protected Dimension	getMinSize(Container parent, GridBagLayoutInfo info)	It figures out the minimum size of the master based on the

		information from getLayoutInfo.
protected Dimension	GetMinSize(Container parent, GridBagLayoutInfo info)	This method is obsolete and supplied for backwards compatibility only

Event listener

A More Complex Example

The event model, which you saw at its simplest in the preceding example, is quite powerful and flexible. Any number of event listener objects can listen for all kinds of events from any number of event source objects. For example, a program might create one listener per event source. Or a program might have a single listener for all events from all sources. A program can even have more than one listener for a single kind of event from a single event source.



Multiple listeners can register to be notified of events of a particular type from a particular source. Also, the same listener can listen to notifications from different objects.

Each event is represented by an object that gives information about the event and identifies the event source. Event sources are often components or models, but other kinds of objects can also be event sources.

Whenever you want to detect events from a particular component, first check the how-to section for that component. A list of the component how-to sections is here. The how-to sections give examples of handling the events that you are most likely to care about. In How to Use Color Choosers, for instance, you will find an example of writing a change listener to track when the color changes in the color chooser.

The following example demonstrates that event listeners can be registered on multiple objects and that the same event can be sent to multiple listeners. The example contains two event sources (JButton instances) and two event listeners. One of the event listeners (an instance of a class called MultiListener) listens for events from both buttons. When it receives an event, it adds the event's "action command" (which is set to the text on the button's label) to the top text area. The second event listener (an instance of a class called Eavesdropper) listens for events on only one of the buttons. When it receives an event, it adds the action command to the bottom text area.

The Event listener represent the interfaces responsible to handle events. Java provides us various Event listener classes but we will discuss those which are more frequently used. Every method of an event listener method has a single argument as an object which is subclass of EventObject class. For example, mouse event listener methods will accept instance of MouseEvent, where MouseEvent derives from EventObject.

EventListner interface

It is a marker interface which every listener interface has to extend.This class is defined in java.util package.

Class declaration

Following is the declaration for java.util.EventListener interface:

```
public interface EventListener
```

AWT Event Listener Interfaces:

Following is the list of commonly used event listeners.

Sr. No.	Control & Description
1	ActionListener This interface is used for receiving the action events.
2	ComponentListener This interface is used for receiving the component events.
3	ItemListener This interface is used for receiving the item events.
4	KeyListener This interface is used for receiving the key events.
5	MouseListener This interface is used for receiving the mouse events.
6	TextListener

	This interface is used for receiving the text events.
7	WindowListener This interface is used for receiving the window events.
8	AdjustmentListener This interface is used for receiving the adjusmtent events.
9	ContainerListener This interface is used for receiving the container events.
10	MouseMotionListener This interface is used for receiving the mouse motion events.
11	FocusListener This interface is used for receiving the focus events.

Java ActionListener Interface

The Java ActionListener is notified whenever you click on the button or menu item. It is notified against ActionEvent. The ActionListener interface is found in java.awt.event package

. It has only one method: actionPerformed().

actionPerformed() method

The actionPerformed() method is invoked automatically whenever you click on the registered component.

```
public abstract void actionPerformed(ActionEvent e);
```

How to write ActionListener

The common approach is to implement the ActionListener. If you implement the ActionListener class, you need to follow 3 steps:

Implement the ActionListener interface in the class:

```
1. public class ActionListenerExample Implements ActionListener
```

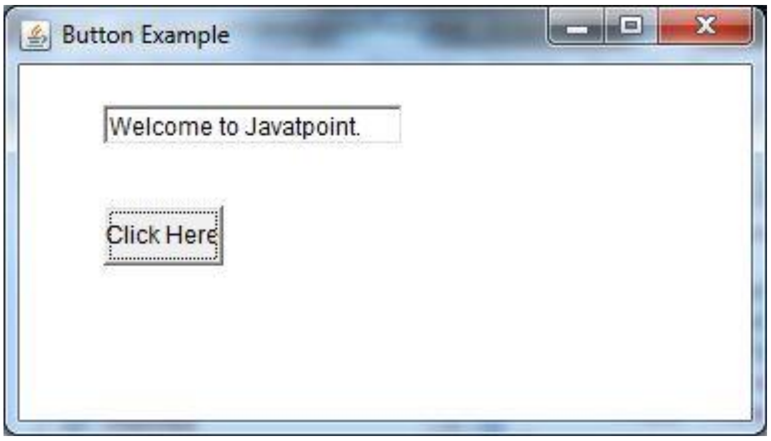
2) Register the component with the Listener:

```
1.  component.addActionListener(instanceOfListenerclass);
```

3) Override the actionPerformed() method:

```
public void actionPerformed(ActionEvent e){
        //Write the code here
        }  import java.awt.*;
        import java.awt.event.*;
        //1st step
        public class ActionListenerExample implements ActionListener{
        public static void main(String[] args) {
            Frame f=new Frame("ActionListener Example");
            final TextField tf=new TextField();
            tf.setBounds(50,50, 150,20);
            Button b=new Button("Click Here");
            b.setBounds(50,100,60,30);
            //2nd step
            b.addActionListener(this);
            f.add(b);f.add(tf);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
        }
        //3rd step
        public void actionPerformed(ActionEvent e){
                tf.setText("Welcome to Javatpoint.");
        }
        }
```

Output:



Java MouseListener Interface

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.

Methods of MouseListener interface

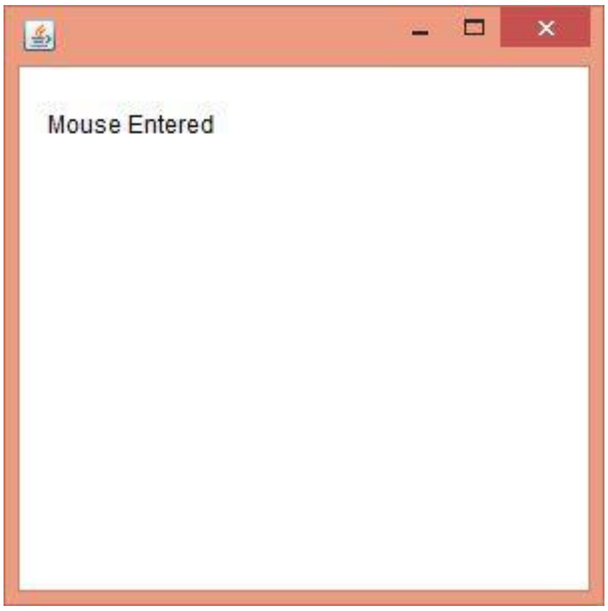
The signature of 5 methods found in MouseListener interface are given below:

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

Java MouseListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListene
r{
    Label l;
    MouseListenerExample(){
        addMouseListener(this);

        l=new Label();
        l.setBounds(20,50,100,20);
        add(l);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {
        l.setText("Mouse Clicked");
    }
    public void mouseEntered(MouseEvent e) {
        l.setText("Mouse Entered");
    }
    public void mouseExited(MouseEvent e) {
        l.setText("Mouse Exited");
    }
    public void mousePressed(MouseEvent e) {
        l.setText("Mouse Pressed");
    }
    public void mouseReleased(MouseEvent e) {
        l.setText("Mouse Released");
    }
  public static void main(String[] args) {
    new MouseListenerExample();
  }
  }
```

Output:


```
Mouse Entered
```

Java Adapter Classes

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*.

Pros 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.

The adapter classes are found in **java.awt.event, java.awt.dnd** and **javax.swing.event** [packages](). The Adapter classes with their corresponding listener interfaces are given below.

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

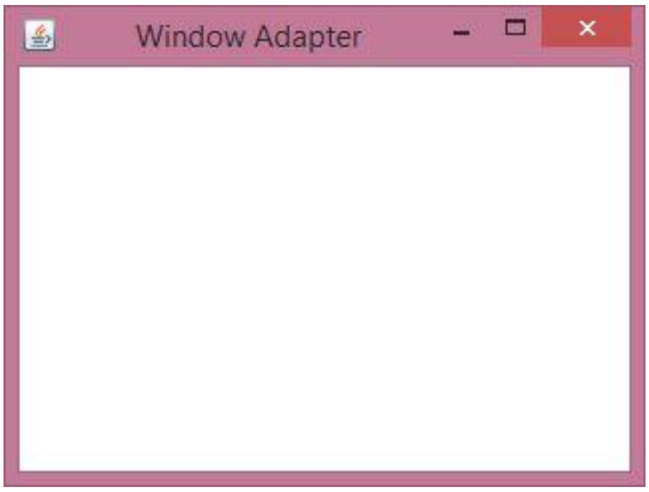
Adapter class	**Listener interface**
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Adapter class	**Listener interface**
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

```
/ importing the necessary libraries
import java.awt.*;
import java.awt.event.*;

public class AdapterExample {
// object of Frame
   Frame f;
// class constructor
    AdapterExample() {
// creating a frame with the title
        f = new Frame ("Window Adapter");
// adding the WindowListener to the frame
// overriding the windowClosing() method
        f.addWindowListener (new WindowAdapter() {
            public void windowClosing (WindowEvent e) {
                f.dispose();
            }
        });
         // setting the size, layout and
        f.setSize (400, 400);
        f.setLayout (null);
        f.setVisible (true);
    }

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

Output:



How to close AWT Window in Java

We can close the AWT Window or Frame by calling *dispose()* or *System.exit()* inside windowClosing() method. The windowClosing() method is found in **WindowListener** interface and **WindowAdapter** class.

The WindowAdapter class implements WindowListener interfaces. It provides the default implementation of all the 7 methods of WindowListener interface. To override the windowClosing() method, you can either use WindowAdapter class or WindowListener interface.

If you implement the WindowListener interface, you will be forced to override all the 7 methods of WindowListener interface. So it is better to use WindowAdapter class.

Different ways to override windowClosing() method

There are many ways to override windowClosing() method:

- By anonymous class
- By inheriting WindowAdapter class
- By implementing WindowListener interface

Close AWT Window Example 1: Anonymous class

In the following example, we are implementing the windowClosing() method of Window using the Anonymous class

WindowExample.java

```
// importing necessary awt libraries
import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

// class which inherits the Frame class
public class WindowExample extends Frame {
// class constructor
   WindowExample() {
// adding WindowListener to the Frame
// and using the windowClosing() method of WindowAdapter class
      addWindowListener (new WindowAdapter() {
         public void windowClosing (WindowEvent e) {
            dispose();
         }
      });
// setting the size, layout and visibility of frame
      setSize (400, 400);
      setLayout (null);
      setVisible (true);
   }
// main method
```

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

Output: