UNIT V

EVENT DRIVEN PROGRAMMING

Graphics programming - Frame – Components - working with 2D shapes - Using color, fonts, and images - Basics of event handling - event handlers - adapter classes - actions - mouse events - AWT event hierarchy - Introduction to Swing – layout management - Swing Components – Text Fields , Text Areas – Buttons- Check Boxes – Radio Buttons – Lists- choices- Scrollbars – Windows –Menus – Dialog Boxes

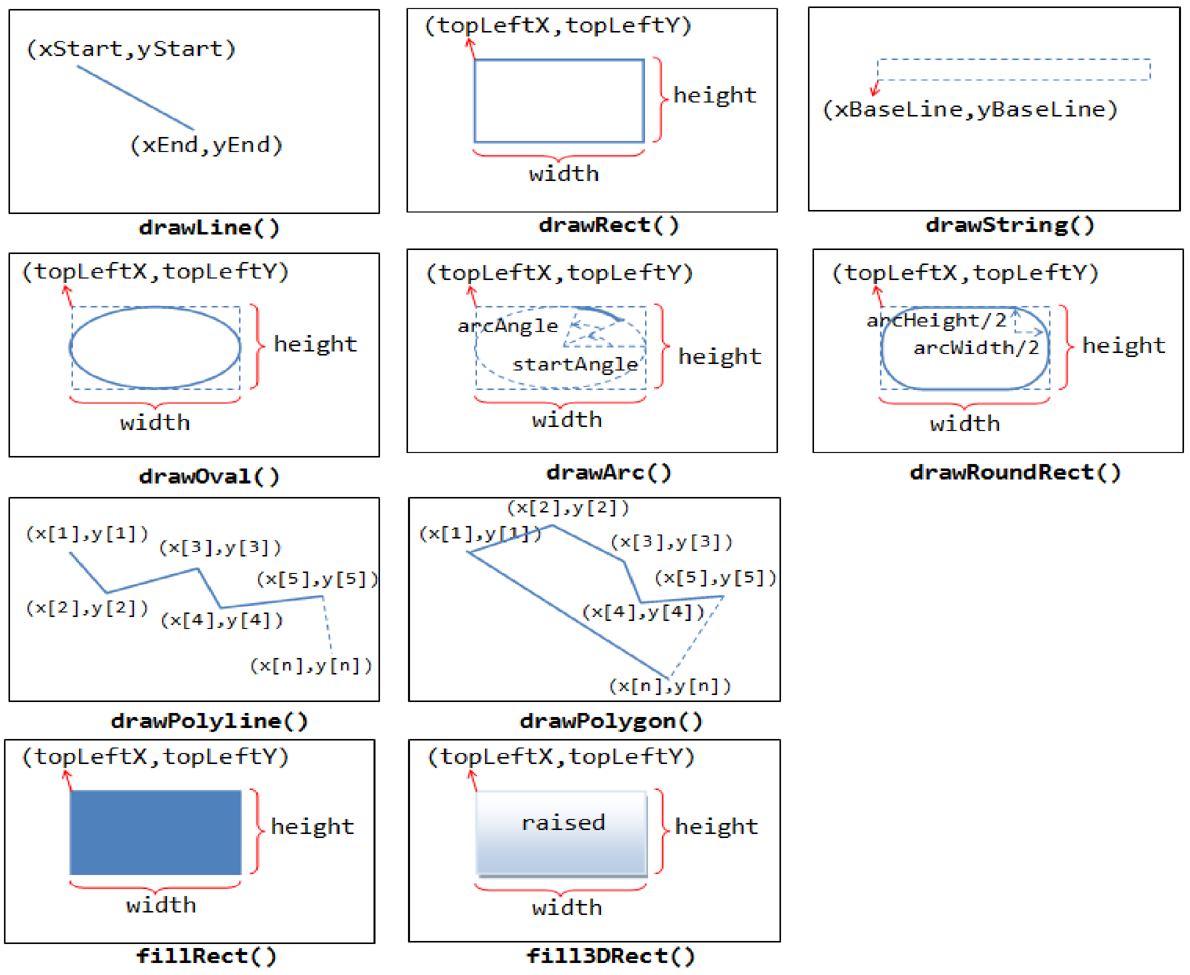
GRAPHICS PROGRAMMING

The **Graphics class** is the abstract super **class** for all **graphics** contexts which allow an application to draw onto components that can be realized on various devices, or onto off-screen images as well. A **Graphics** object encapsulates all state information required for the basic rendering operations that **Java** supports. The graphics class defines a number of drawing functions, Each shape can be drawn edge-only or filled. To draw shapes on the screen, we may call one of the methods available in the graphics class. The most commonly used drawing methods included in the graphics class are listed below.

Drawing Methods of the Graphics Class

Sr.No	Method	Description
1.	clearRect()	Erase a rectangular area of the canvas.
2.	copyAre()	Copies a rectangular area of the canvas to another area
3.	drawArc()	Draws a hollow arc
4.	drawLine()	Draws a straight line
5	drawOval()	Draws a hollow oval
6	drawPolygon()	Draws a hollow polygon
7	drawRect()	Draws a hollow rectangle
8.	drawRoundRect()	Draws a hollow rectangle with rounded corners
9.	drawString()	Display a text string

10.	FillArc()	Draws a filled arc
11.	fillOval()	Draws a filled Oval
12.	fillPolygon()	Draws a filled Polygon
13.	fillRect()	Draws a filled rectangle
14.	fillRoundRect()	Draws a filled rectangle with rounded corners
15.	getColor()	Retrieves the current drawing color
16.	getFont()	Retrieves the currently used font
17.	getFontMetrics()	Retrieves the information about the current font
18.	setColor()	Sets the drawing color
19.	setFont()	Sets the font

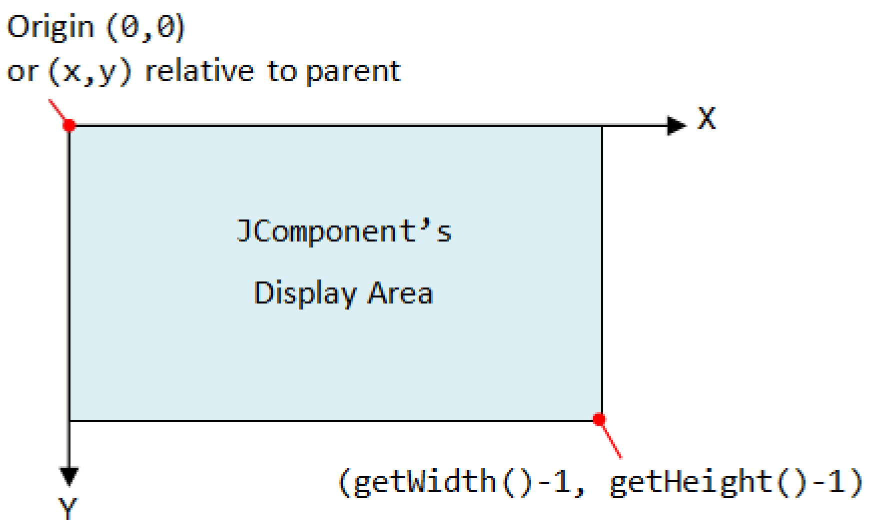


Graphics Class Methods

```
// Graphics context's current color.
void setColor(Color c)
Color getColor()

// Graphics context's current font.
void setFont(Font f)
Font getFont()

// Set/Get the current clip area. Clip area shall be rectangular and no rendering is performed outside the clip area.
void setClip(int xTopLeft, int yTopLeft, int width, int height)
void setClip(Shape rect)
public abstract void clipRect(int x, int y, int width, int height) // intersects the current clip with the given rectangle
Rectangle getClipBounds()  // returns an Rectangle
Shape getClip()           // returns an object (typically Rectangle) implements Shape
```

Graphics Coordinate System

In Java Windowing Subsystem (like most of the 2D Graphics systems), the origin (0,0) is located at the top-left corner. EACH component/container has its own coordinate system, ranging for (0,0) to (width-1, height-1) as illustrated.



You can use method getWidth() and getHeight() to retrieve the width and height of a component/container. You can use getX() or getY() to get the top-left corner (x,y) of this component's origin relative to its parent.

Lines

Lines are drawn by means of the drawLine() method.

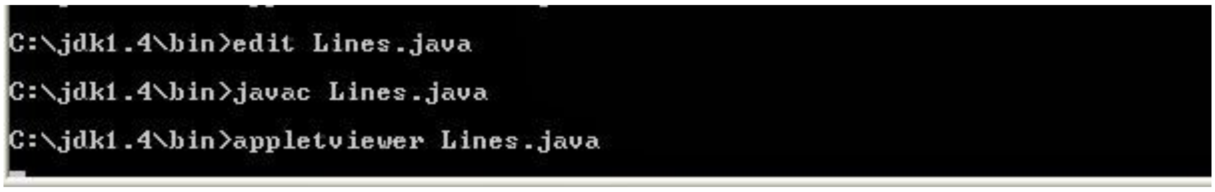
Syntax

void drawLine(int startX, int startY, int endX, int endY)

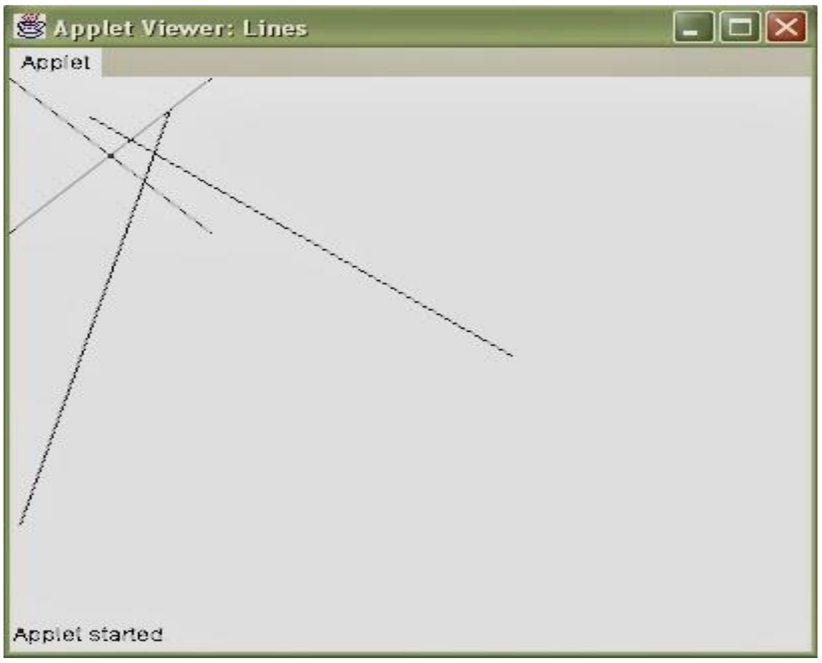
drawLine() displays a line in the current drawing color that begins at (start X, start Y) and ends at (endX, end Y).

```
//Drawing Lines

import java.awt.*;

import java.applet.*;

/*

<applet code="Lines" width=300 Height=250>

</applet>

*/

public class Lines extends Applet

{

    public void paint(Graphics g)

    {

        g.drawLine(0,0,100,100);

        g.drawLine(0,100,100,0);

        g.drawLine(40,25,250,180);

        g.drawLine(5,290,80,19);

    }

}
```

After this you can compile your java applet program as shown below:



```
C:\jdk1.4\bin>edit Lines.java
C:\jdk1.4\bin>javac Lines.java
C:\jdk1.4\bin>appletviewer Lines.java
```



"Output of Lines.class"

Circles and Ellipses

The Graphics class does not contain any method for circles or ellipses. To draw an ellipse, use drawOval(). To fill an ellipse, use fillOval().

Syntax

void drawOval(int top, int left, int width, int height)

void fillOval(int top, int left, int width, int height)

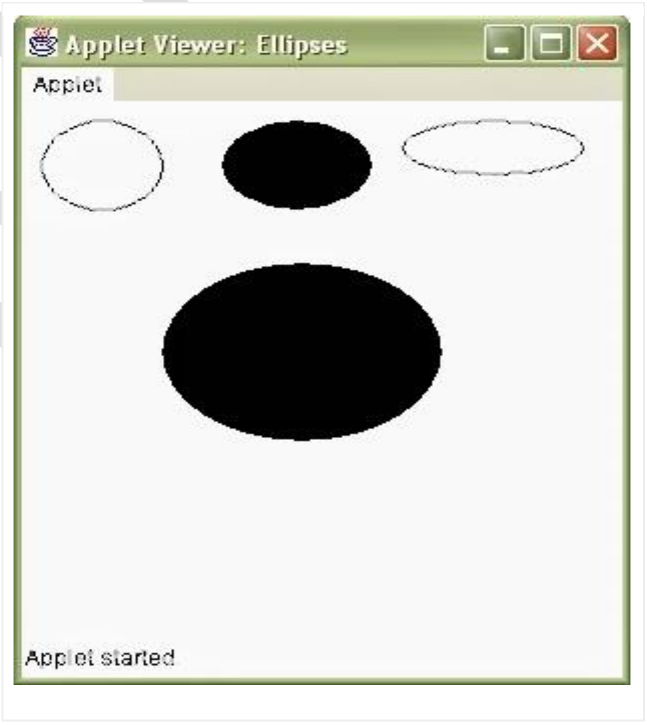
The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by (top,left) and whose width and height are specified by width and height. To draw a circle, specify a square as the bounding rectangle i.e get height = width.

The following program draws serveral ellipses:

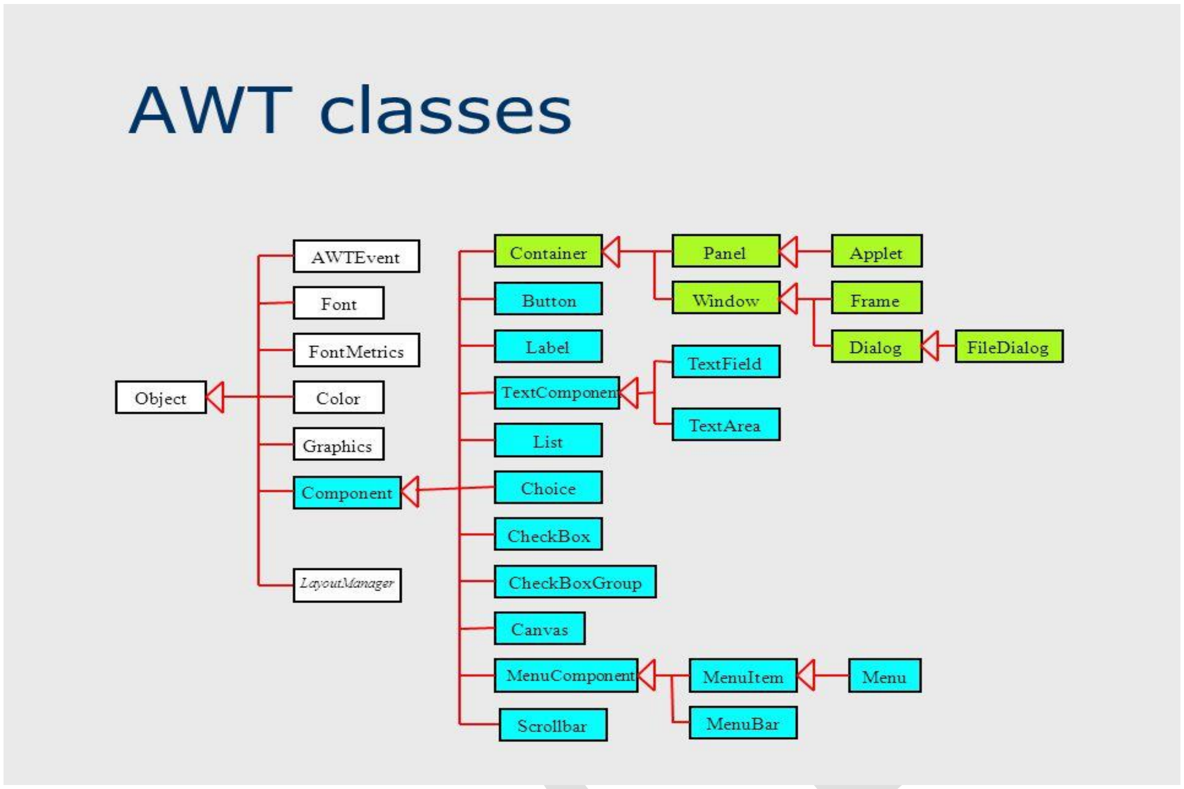
```
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 Height=300>
</applet>
*/
public class Ellipses extends Applet
{
        public void paint(Graphics g)
        {
                g.drawOval(10,10,60,50);
                g.fillOval(100,10,75,50);
                g.drawOval(190,10,90,30);
                g.fillOval(70,90,140,100);
        }
}
```

After this you can comiple your java applet program as shown below:

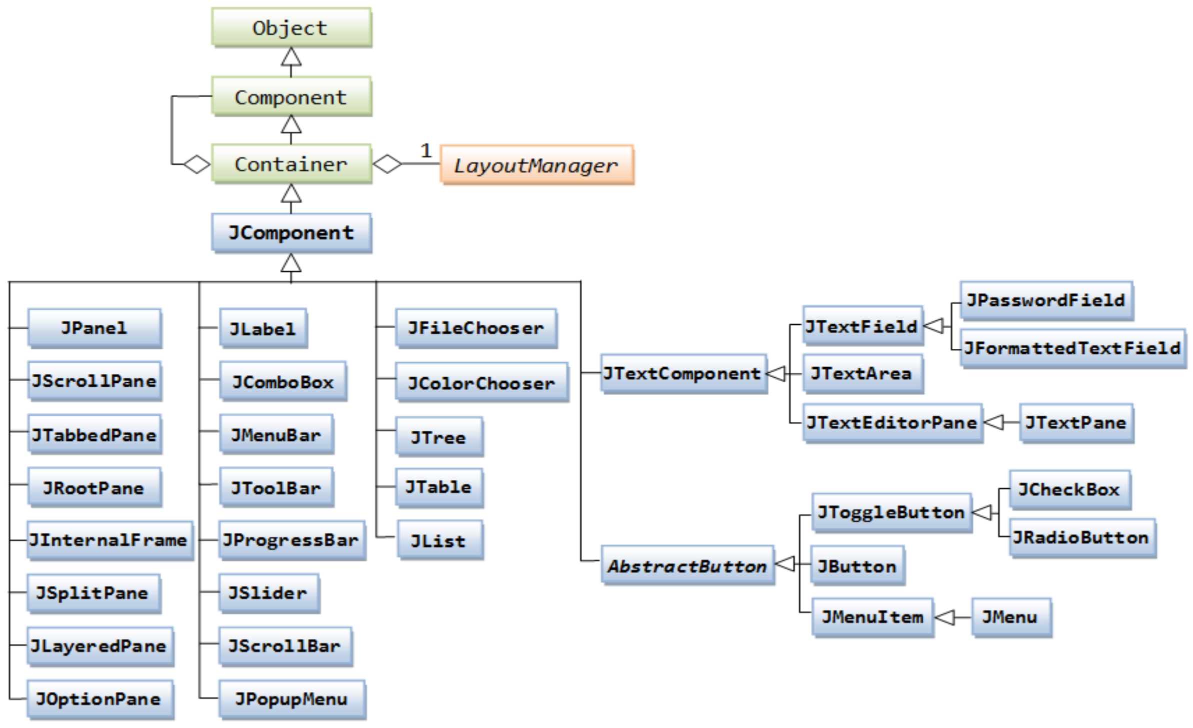
```
c:\jdk1.4\bin\javac Ellipses.java
c:\jdk1.4\bin\appletviewer Ellipses.java
```



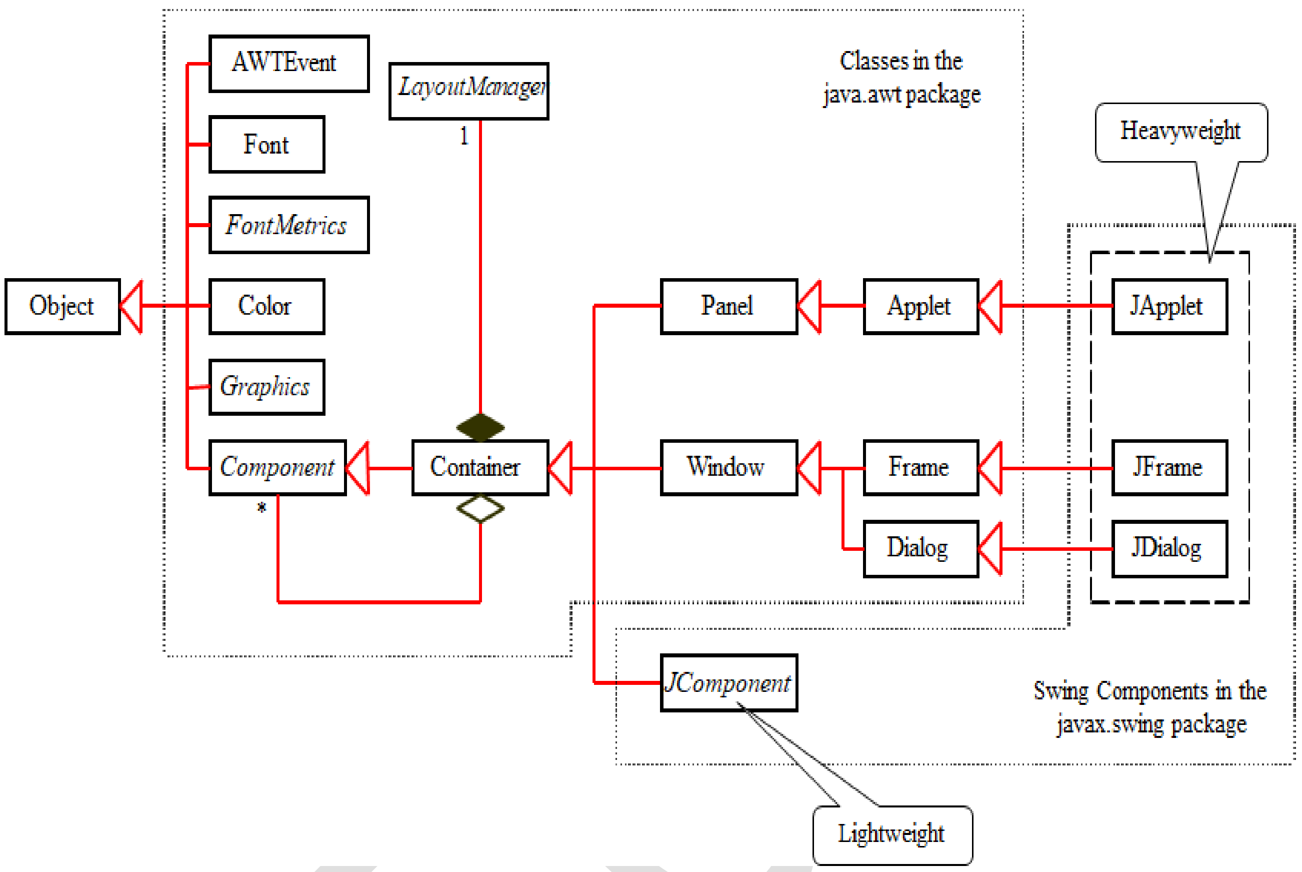
AWT Classes



Swing Classes



AWT and Swing Classes



Difference Between AWT and Swing

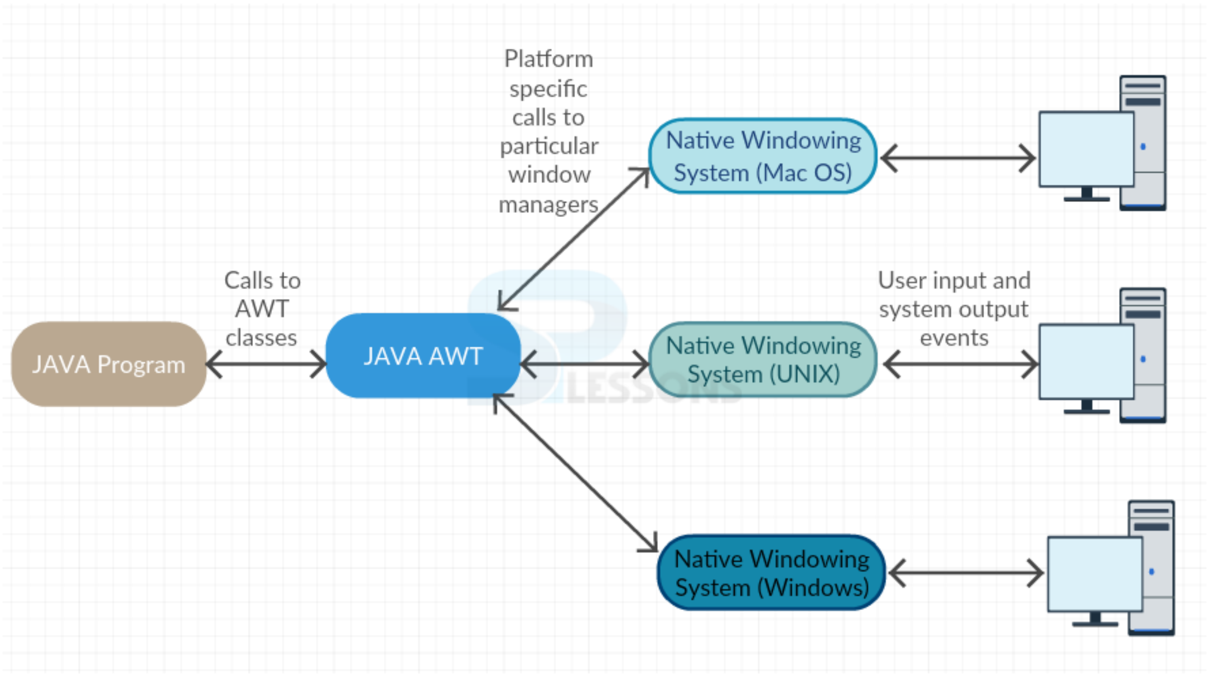
AWT	Swing
AWT components are heavyweight components	Swing components are lightweight components
AWT doesn't support pluggable look and feel	Swing supports pluggable look and feel
AWT programs are not portable	Swing programs are portable
AWT is old framework for creating GUIs	Swing is new framework for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

OVERVIEW OF AWT (ABSTRACT WINDOW TOOLKIT)

The interface between the user and application program is known as an user-interface. A user-interface has multiple forms, which ranges from commands to graphic clicks.

In low-level operating systems, the user has to communicate with the commands to interact with the application. Now, a user can interact with the application in just a click by tracking the mouse and reading the keyboard. Abstract Window Toolkit provides a better object-oriented interface to these low-level operating systems by developing its design and the performance.

AWT has a set of tools that are used in various platforms by including them in the graphical user interface(GUI). Every platform has its GUI Toolkit and the interface elements enhances the look & feel of the applications.


Platform specific calls to particular window managers
Native Windowing System (Mac OS)
Calls to AWT classes
User input and system output events
JAVA Program
JAVA AWT
Native Windowing System (UNIX)
Native Windowing System (Windows)

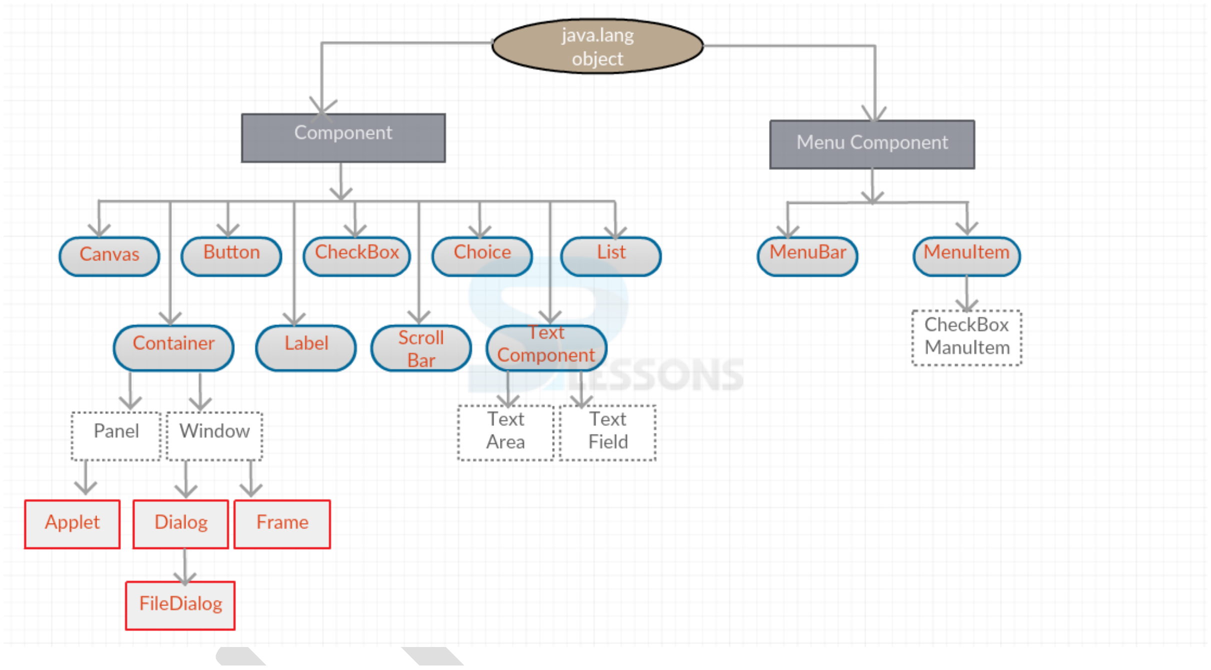
AWT elements like Window, Frame, Panel, Button, Textfield, TextArea, Listbox, Combobox, Label, and Checkbox will be explained in the next chapters. But, for further proceeding, one should have a knowledge of Java programming, executing programs, and text editors like notepad.

The main drawback of using AWT is that the GUI interface designed on a platform may change its features when displayed on the other platforms and is a heavy-weight application because it has to communicate with the operating system for its resources. This makes AWT less popular these days.

Structure of AWT

- AWT hierarchy
- Terminology used
- Layouts introduction

Structure of AWT contains many components that are to be placed in GUI interface.



A java object has **components** and **menu components**. Components play a key role by providing Canvas, Buttons, Checkboxes, Choices, List, Scroll bar, TextComponent and Container. The TextComponent consists of textfield and textarea.
The Container is further divided into 4 main parts: Window, Frame, Panel, and Dialog. All the applets are maintained and developed in a container.
MenuComponent will have MenuBar and MenuItem, which is further divided into CheckBoxMenuItem.

Term	**Description**
Component	An object can be displayed on screen, interact with the user, and has graphical representation.
Container	An ordered list of components is a container.

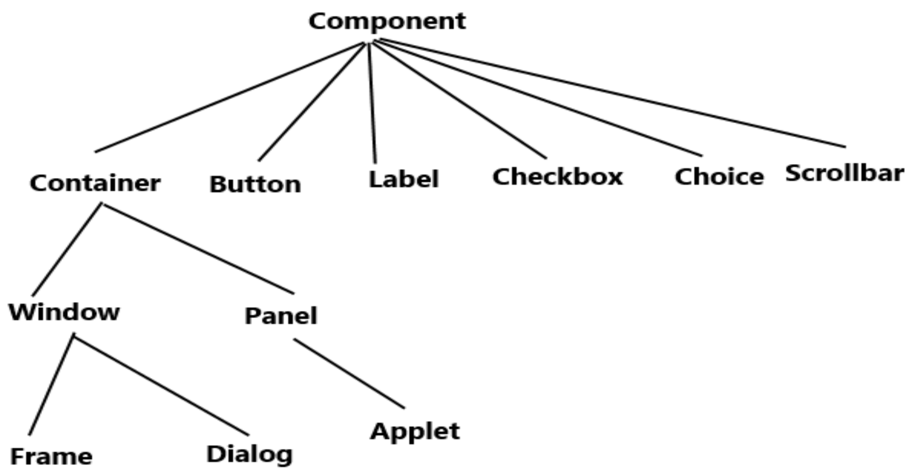
Panel	The space provided for components by the application is called as a panel. That space can also be acquired by other panels too.
Window	The screen area on the desktop displaying the data and performing the operations specified is called as a window.
Frame	A high-level window with corners, titles, menus, and borders is called as a frame.
Canvas	Canvas is a blank area, which displays an application and receives the input in the form of events.

All the components built are placed in containers to get a required layout. All the layouts are managed by the layout manager and gives directions while arranging the components to get the desired output.

AWT COMPONENTS

A Graphical User Interface Application consists a set of elements called as **Components**. A Component is an abstract class, which produces an object of some features like size, position and projects them to the screen by taking the events.

AWT Components are placed in **containers** and are controlled by it.

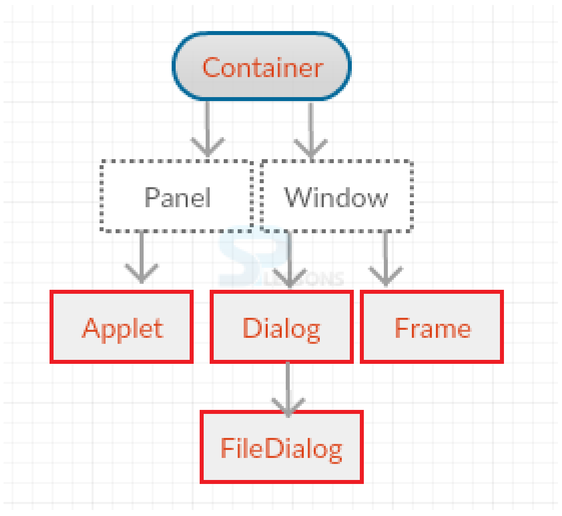


AWT Container

AWT Container is a component and holds all other components.

- Container is a sub-class of the component class and, super-class of all container classes.

- AWT Container Container sub-classes are also containers i.e containers contains other containers.
- Layout Manager is assigned to every container, which helps in making changes using the setlayout()method.
- Applets has the default container with default LayoutManager called FlowLayout.



Declaration :

java.awt.Container class has the following declaration:

public class Container extends Component

Some of the commonly used Container class methods are:

- add() : Component is included in the container by overloading this method.
- invalidate() : The components that are setup in the container can be invalidated.
- validated(): The components that are invalidated by using the above method invalidate() can be validated again using validate().

Java AWT Panel

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

AWT Panel class declaration

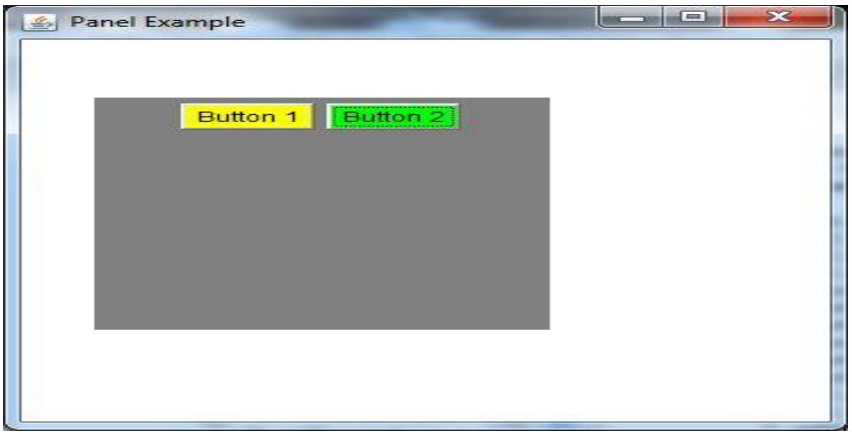
```
public class Panel extends Container implements Accessible
```

Java AWT Panel Example

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

```
public class PanelExample {
     PanelExample()
        {
        Frame f= new Frame("Panel Example");
        Panel panel=new Panel();
        panel.setBounds(40,80,200,200);
        panel.setBackground(Color.gray);
        Button b1=new Button("Button 1");
        b1.setBounds(50,100,80,30);
        b1.setBackground(Color.yellow);
        Button b2=new Button("Button 2");
        b2.setBounds(100,100,80,30);
        b2.setBackground(Color.green);
        panel.add(b1); panel.add(b2);
        f.add(panel);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
        }
        public static void main(String args[])
        {
        new PanelExample();
        }
}
```

Output:



Java AWT Dialog

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

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

Frame vs Dialog

Frame and Dialog both inherits Window class. Frame has maximize and minimize buttons but Dialog doesn't have.

AWT Dialog class declaration

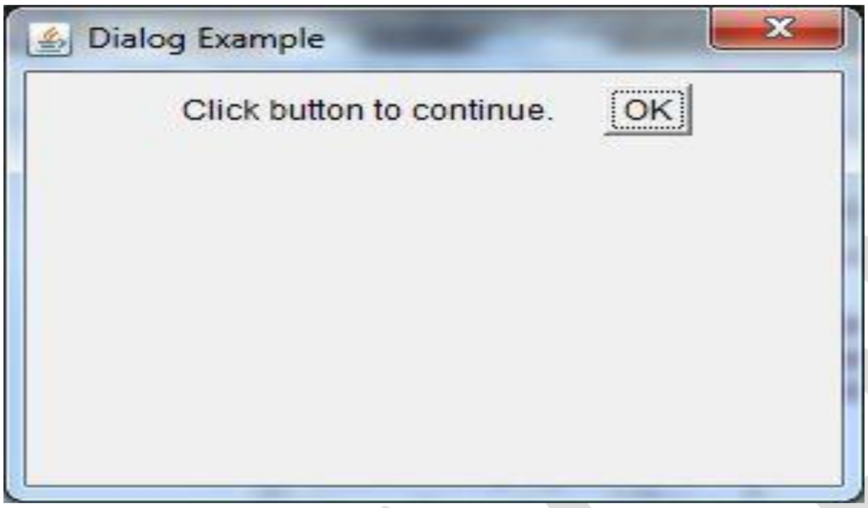
```
public class Dialog extends Window
```

Java AWT Dialog Example

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

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

Output:



AWT Frame

Frame is considered as a window with a menu bar, borders and titles in real-time environment. A frame can be modified as per the requirement. The default LayoutManager is BorderLayout. Frames can divide the data into different windows.

Declaration :

java.awt.Frame class can be declared as follows:

```
public class Frame extends Window implements MenuContainer
```

To create a frame without a title bar, use Frame() constructor that does not have any arguments.

```
Frame f = new Frame();
```

Window title can be declared using strings as below:

```
Frame f = new Frame("Window Title");
```

A component can be added to a frame by using add() method. For example,

```
Frame f = new Frame("Example Window");
```

```
        f.add(new Button("Correct");
```

Otherwise this can be used, if present inside the class.

```
        this.f(new Button("Correct");
```

A Frame can be given a size using the setsize() method. For example,

```
        f.setsize(150,100);
```

After adding all the required components , it must be made visible as it is initially set to invisible. This can be done by using setVisible() method. For example,

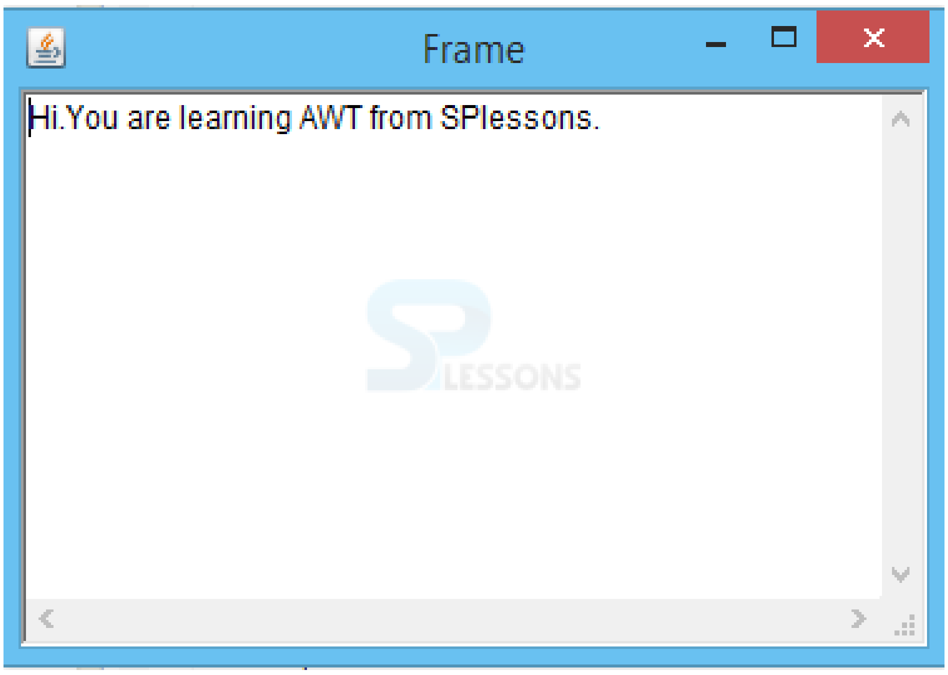
```
        f.setVisible(true);
```

Otherwise if one wants the exact size, use pack() method.

Example

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class Splesson {

  public static void main(String[] args) {

    Frame f = new Frame("App Frame");
    f.setSize(250, 250);
    f.setLocation(300,200);
    f.add(BorderLayout.CENTER, new TextArea(10, 40));
    f.setVisible(true);
    f.addWindowListener ( new WindowAdapter () {
                          public void windowClosing ( WindowEvent evt )
                          {
                                  System.exit(0);
                          }
              });
     }
}
```

Output



Hi.You are learning AWT from SPlessons.

Java AWT Button

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.

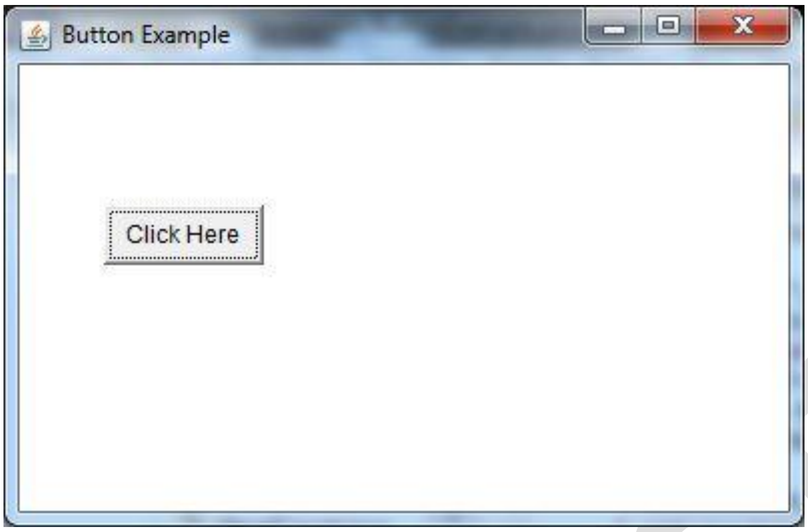
AWT Button Class declaration

```
public class Button extends Component implements Accessible
```

Java AWT Button Example

```
import java.awt.*;
public class ButtonExample {
public static void main(String[] args) {
    Frame f=new Frame("Button Example");
    Button b=new Button("Click Here");
    b.setBounds(50,100,80,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



Java AWT Label

The object of 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 an application but a user cannot edit it directly.

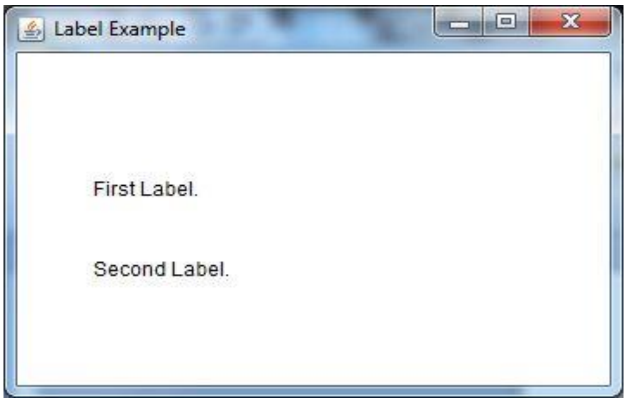
AWT Label Class Declaration

```
public class Label extends Component implements Accessible
```

Java Label Example

```
import java.awt.*;
class LabelExample{
public static void main(String args[]){
    Frame f= new Frame("Label Example");
    Label l1,l2;
    l1=new Label("First Label.");
    l1.setBounds(50,100, 100,30);
    l2=new Label("Second Label.");
    l2.setBounds(50,150, 100,30);
    f.add(l1); f.add(l2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



Java AWT TextField

The object of a TextField class is a text component that allows the editing of a single line text. It inherits TextComponent class.

AWT TextField Class Declaration

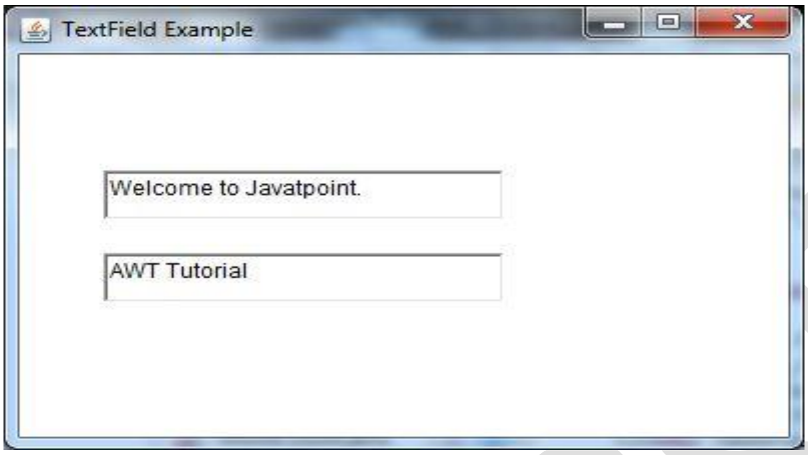
```
public class TextField extends TextComponent
```

Java AWT TextField Example

```
import java.awt.*;
class TextFieldExample{
public static void main(String args[]){
    Frame f= new Frame("TextField Example");
    TextField t1,t2;
    t1=new TextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new TextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
```

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

Output:



Welcome to Javatpoint.

AWT Tutorial

Java AWT TextArea

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

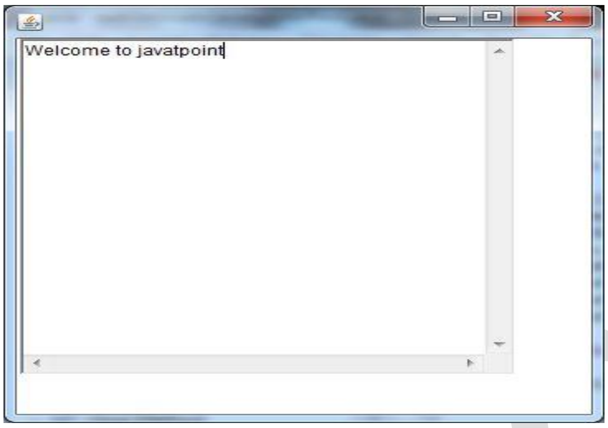
AWT TextArea Class Declaration

```
public class TextArea extends TextComponent
```

Java AWT TextArea Example

```
import java.awt.*;
public class TextAreaExample
{
    TextAreaExample(){
      Frame f= new Frame();
         TextArea area=new TextArea("Welcome to javatpoint");
      area.setBounds(10,30, 300,300);
      f.add(area);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
public static void main(String args[])
{
  new TextAreaExample();
}  }
```

Output:



Java AWT Checkbox

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

AWT Checkbox Class Declaration

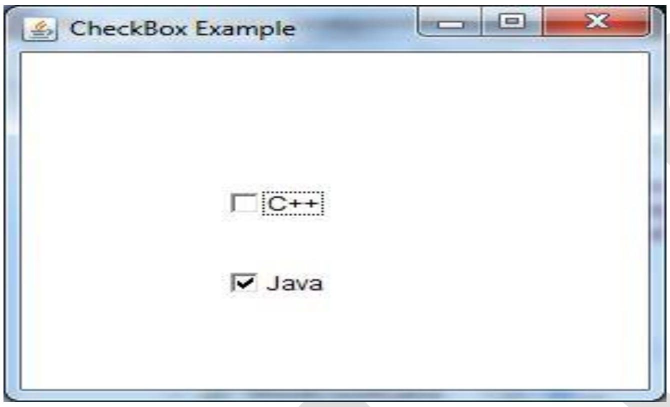
```
public class Checkbox extends Component implements ItemSelectable, Accessible
```

Java AWT Checkbox Example

```
import java.awt.*;
public class CheckboxExample
{
       CheckboxExample(){
       Frame f= new Frame("Checkbox Example");
       Checkbox checkbox1 = new Checkbox("C++");
       checkbox1.setBounds(100,100, 50,50);
       Checkbox checkbox2 = new Checkbox("Java", true);
       checkbox2.setBounds(100,150, 50,50);
       f.add(checkbox1);
```

```
        f.add(checkbox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
    new CheckboxExample();
}  }
```

Output:



Java AWT Choice

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

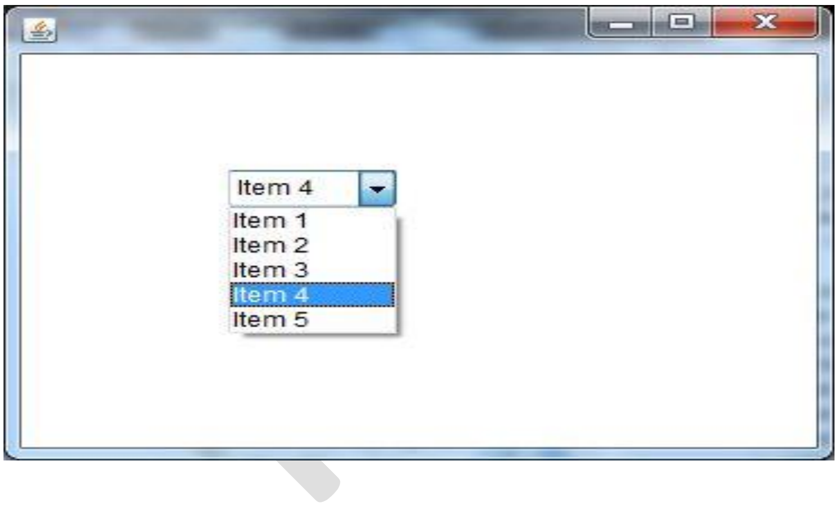
AWT Choice Class Declaration

public class Choice **extends** Component **implements** ItemSelectable, Accessible

Java AWT Choice Example

```
import java.awt.*;
public class ChoiceExample
{
        ChoiceExample(){
        Frame f= new Frame();
        Choice c=new Choice();
        c.setBounds(100,100, 75,75);
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
        f.add(c);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
      new ChoiceExample();
}
}
```

Output:



Java AWT List

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

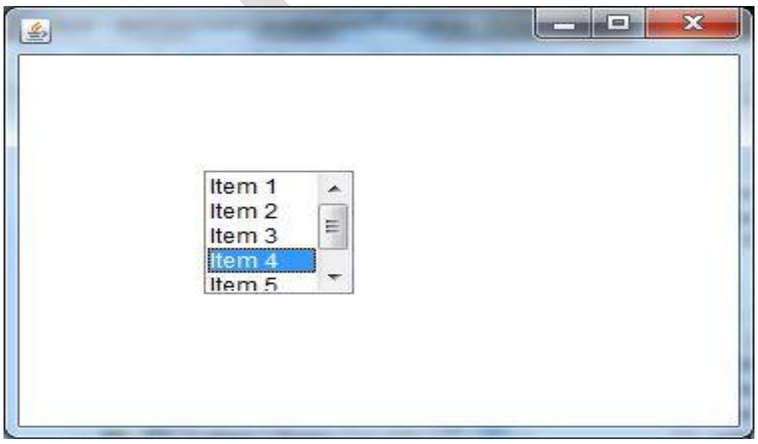
AWT List class Declaration

public class List **extends** Component **implements** ItemSelectable, Accessible

Java AWT List Example

```
import java.awt.*;
public class ListExample
{
        ListExample(){
        Frame f= new Frame();
        List l1=new List(5);
        l1.setBounds(100,100, 75,75);
        l1.add("Item 1");
        l1.add("Item 2");
        l1.add("Item 3");
        l1.add("Item 4");
        l1.add("Item 5");
        f.add(l1);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
   new ListExample();
}
}
```

Output:



Java AWT Canvas

The Canvas control represents a blank rectangular area where the application can draw or trap input events from the user. It inherits the Component class.

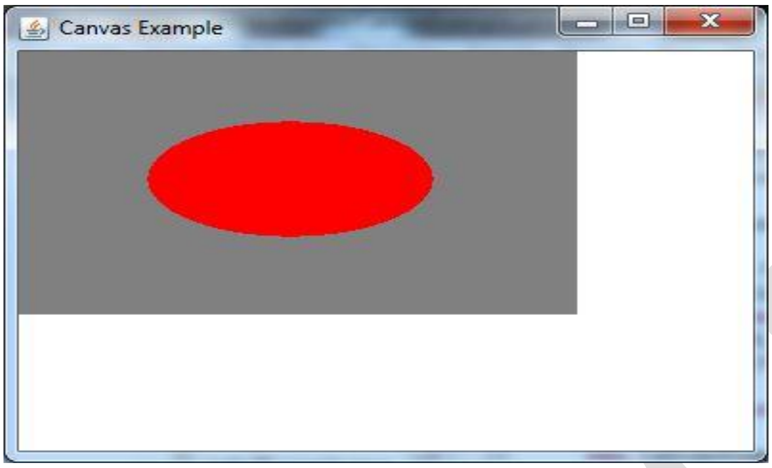
AWT Canvas class Declaration

```
public class Canvas extends Component implements Accessible
```

Java AWT Canvas Example

```
import java.awt.*;
public class CanvasExample
{
  public CanvasExample()
  {
    Frame f= new Frame("Canvas Example");
    f.add(new MyCanvas());
    f.setLayout(null);
    f.setSize(400, 400);
    f.setVisible(true);
  }
  public static void main(String args[])
  {
    new CanvasExample();
  }
}
class MyCanvas extends Canvas
{
        public MyCanvas() {
        setBackground (Color.GRAY);
        setSize(300, 200);
     }
  public void paint(Graphics g)
  {
    g.setColor(Color.red);
    g.fillOval(75, 75, 150, 75);
  }
}
```

Output:



Java AWT Scrollbar

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

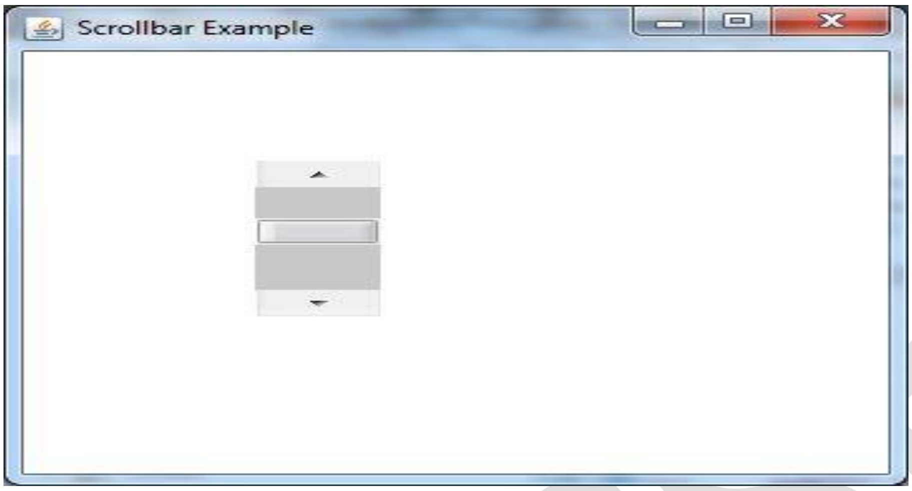
AWT Scrollbar class declaration

```
public class Scrollbar extends Component implements Adjustable, Accessible
```

Java AWT Scrollbar Example

```
import java.awt.*;
class ScrollbarExample{
ScrollbarExample(){
        Frame f= new Frame("Scrollbar Example");
        Scrollbar s=new Scrollbar();
        s.setBounds(100,100, 50,100);
        f.add(s);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[]){
        new ScrollbarExample();
}
}
```

Output:



Java AWT MenuItem and Menu

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

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration

```
public class MenuItem extends MenuComponent implements Accessible
```

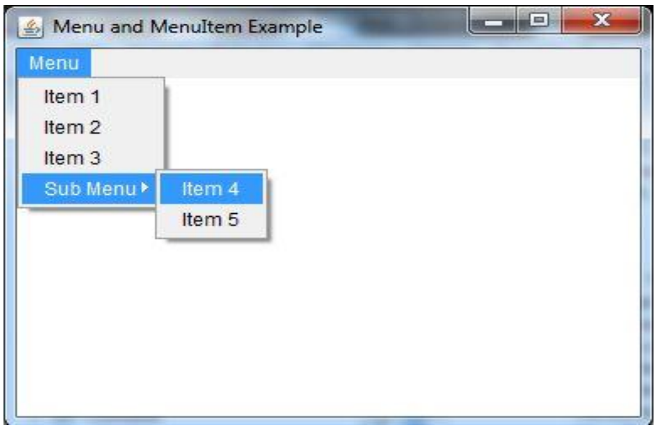
AWT Menu class declaration

```
public class Menu extends MenuItem implements MenuContainer, Accessible
```

Java AWT MenuItem and Menu Example

```
import java.awt.*;
class MenuExample
{
        MenuExample(){
        Frame f= new Frame("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("Menu");
        Menu submenu=new Menu("Sub Menu");
        MenuItem i1=new MenuItem("Item 1");
        MenuItem i2=new MenuItem("Item 2");
        MenuItem i3=new MenuItem("Item 3");
        MenuItem i4=new MenuItem("Item 4");
        MenuItem i5=new MenuItem("Item 5");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        submenu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```

Output:



JAVA APPLET

Applet is java program that can be embedded into HTML pages. Java applets runs on the java enables web browsers such as mozila and internet explorer. Applet is designed to run remotely on the client browser, so there are some restrictions on it. Applet can't access system resources on the local computer. Applets are used to make the web site more dynamic and entertaining.

Advantages of Applet:

- Applets are cross platform and can run on Windows, Mac OS and Linux platform
- Applets can work all the version of Java Plugin
- Applets runs in a sandbox, so the user does not need to trust the code, so it can work without security approval
- Applets are supported by most web browsers
- Applets are cached in most web browsers, so will be quick to load when returning to a web page
- User can also have full access to the machine if user allows

Disadvantages of Java Applet:

- Java plug-in is required to run applet
- Java applet requires JVM so first time it takes significant startup time
- If applet is not already cached in the machine, it will be downloaded from internet and will take time
- Its difficult to desing and build good user interface in applets compared to HTML technology

Life Cycle of a Applet

1. **init( ) :** The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.
2. **start( ) :** The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )** is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.
3. **paint( ) :** The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which

the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.

paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.

The **paint( )** method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

4. **stop( ) :** The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.
5. **destroy( ) :** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

Creating Hello World applet example:

Let's begin with the HelloWorld applet :

```
// A Hello World Applet
// Save file as Demo.java

import java.applet.Applet;
import java.awt.Graphics;

// Demo class extends Applet
public class Demo extends Applet
{
   // Overriding paint() method
   @Override
   public void paint(Graphics g)
   {
        g.drawString("Hello World", 20, 20);
    }

}
```

How to execute Applet Program

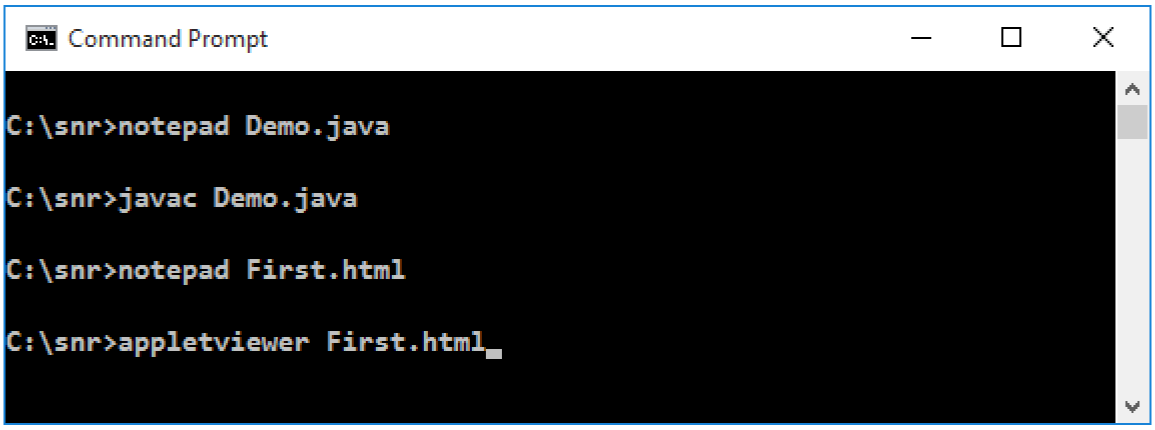
To run the above Demo applet, write one more file **First.html**.

<applet code="Demo.class" width="200" height="225">

</applet>

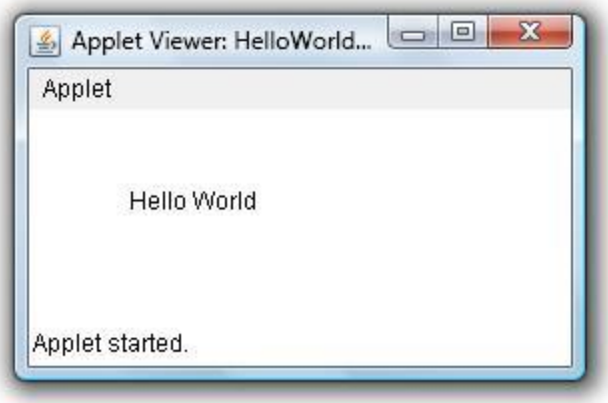
Write source codes and execute the applet as in following screen.

Compilations Steps



```
C:\snr>notepad Demo.java

C:\snr>javac Demo.java

C:\snr>notepad First.html

C:\snr>appletviewer First.html
```

Output



```
Hello World

Applet started.
```

If you include a comment at the head of your Java source code file that contains the APPLET tag then your code is documented with a prototype of the necessary HTML statements, and you can run your compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the HelloWorld source file looks like this :

```
// A Hello World Applet
// Save file as HelloWorld.java
import java.applet.Applet;
import java.awt.Graphics;

/*
<applet code="HelloWorld" width=200 height=60>
</applet>
*/

// HelloWorld class extends Applet
public class HelloWorld extends Applet
{
    // Overriding paint() method
    @Override
    public void paint(Graphics g)
    {
        g.drawString("Hello World", 20, 20);
    }

}
```

With this approach, first compile HelloWorld.java file and then simply run below command to run applet :

```
appletviewer HelloWorld
```

Example 2 for drawing Images Example Java in Applets

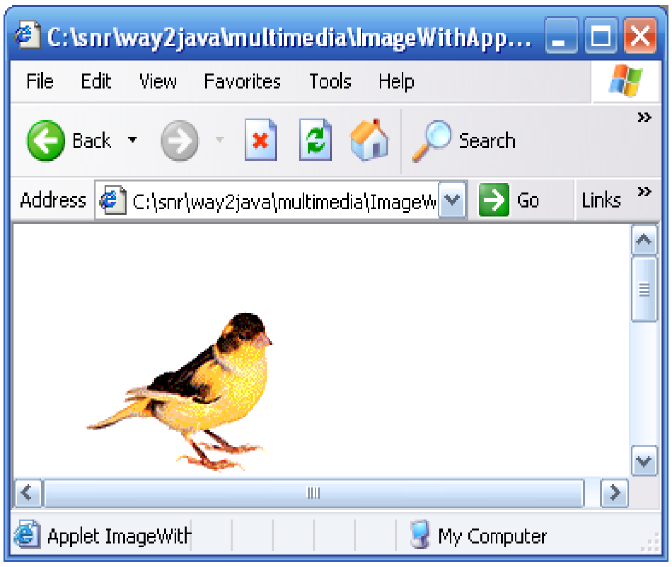
While browsing, you come across, very often, beautiful and attractive images and animations. They can be done with applets. To draw the images, the **java.awt.Graphics** class comes with a method **drawImage()**. With the help of this method and **ImageObserver** at the background, images can be drawn on the applet window. **Following program explains.**

```
import java.awt.Graphics;
import java.applet.Applet;
import java.net.URL;
public class ImageWithApplet extends Applet
{
 ublic void paint( Graphics g )
 {
  URL url1 = getCodeBase();
  Image img = getImage(url1, "bird2.gif");
  g.drawImage(img, 60, 120, this);
 }
}
```

HTML file to run the above applet – ImageWithApplet.html

```
<applet code="ImageWithApplet.class"  width="350"  height="350">
</applet>
```

Output


C:\snr\way2java\multimedia\ImageWithApp...
File Edit View Favorites Tools Help
Back
Search
Address C:\snr\way2java\multimedia\ImageW
Go
Links
Applet ImageWith
My Computer

Explanation

Any image to be drawn should be converted into an object of **Image** class (like ImageIcon in case of Swing). The **getImage()** method of Applet class returns an object of Image, **img**. This method takes two parameters – the address of the image (bird2.gif) as an object of URL and the image itself as a string. Place the image, bird2.gif, in the same folder where the applet exists. **getCodeBase()** method of Applet class returns an object of URL (here, **url1**) that contains the address of the image (as image and applet exist in the same folder).

- *URL url1 = getCodeBase();*
- *Image img = getImage(url1, "bird2.gif");*

EVENT HANDLING

An *event* in Java is an object that is created when something changes within a graphical user interface. If a user clicks on a button, clicks on a combo box, or types characters into a text field, etc., then an event triggers, creating the relevant event object. This behavior is part of Java's Event Handling mechanism and is included in the Swing GUI library.

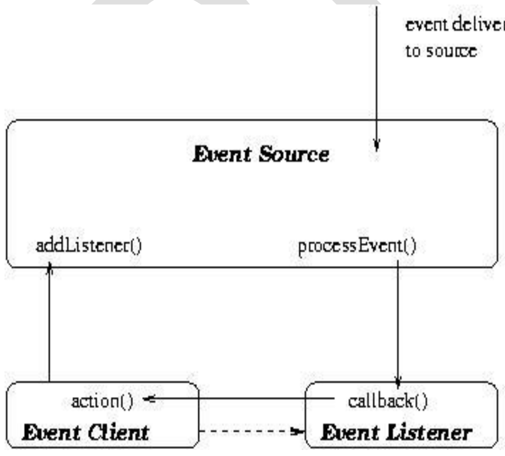
For example, let's say we have a *JButton*. If a user clicks on the *JButton,* a button click event is triggered, the event will be created, and it will be sent to the relevant event listener (in this case, the *ActionListener*). The relevant listener will have implemented code that determines the action to take when the event occurs.

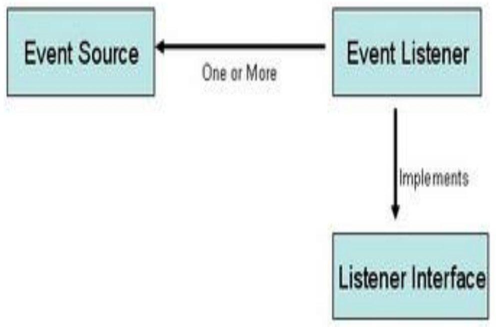
Note that an event source *must* be paired with an event listener, or its triggering will result in no action.

How Events Work

Event handling in Java is comprised of two key elements:

- **The event source**, which is an object that is created when an event occurs. Java provides several types of these event sources.
- **The event listener**, the object that "listens" for events and processes them when they occur.





To implement the ActionListener interface, the listener class must have a method called actionPerformed that receives an ActionEvent object as a parameter.

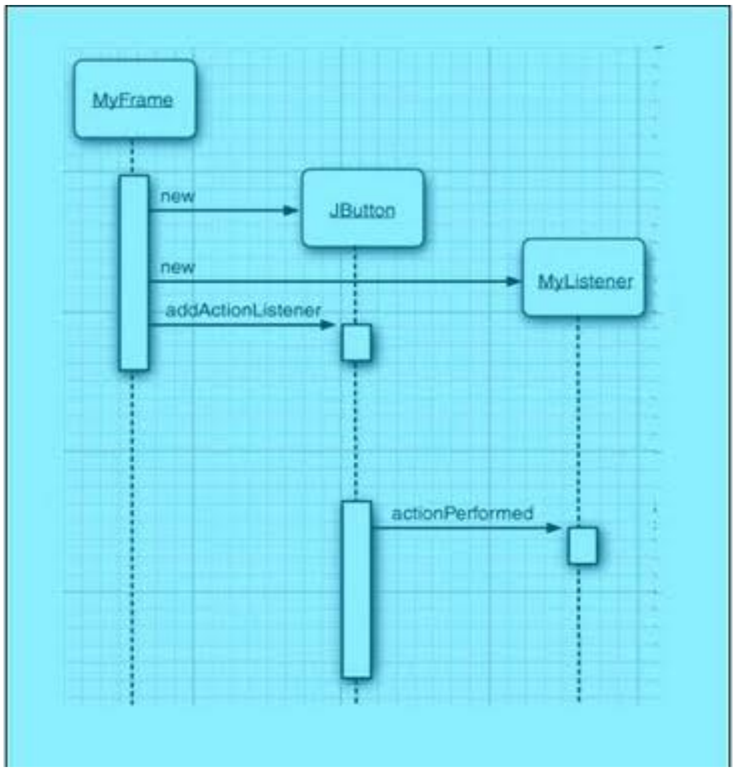
```
ActionListener listener = . . .;
JButton button = new JButton("Ok");
button.addActionListener(listener);
```

To implement the ActionListener interface, the listener class must have a method called actionPerformed that receives an ActionEvent object as a parameter.

```
class MyListener implements ActionListener{
 public void actionPerformed(ActionEvent event)    {
        // code to handle button click goes here
        .                                    .                                    .
    }
}
```

Whenever the user clicks the button, the JButton object creates an ActionEvent object and calls listener.actionPerformed(event), passing that event object. An event source such as a button can have multiple listeners. In that case, the button calls the actionPerformed methods of all listeners whenever the user clicks the button.

You can understand the event notification system by looking at the image below:

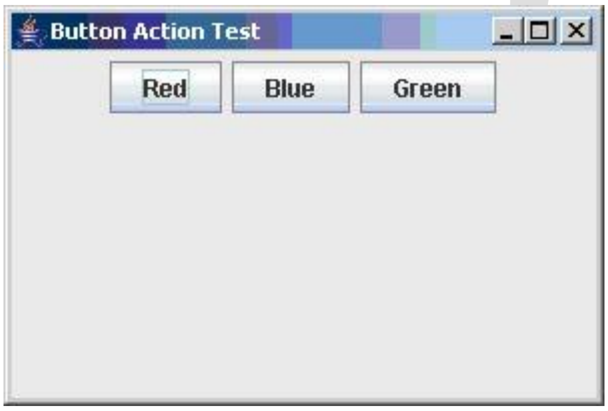


Simple Example

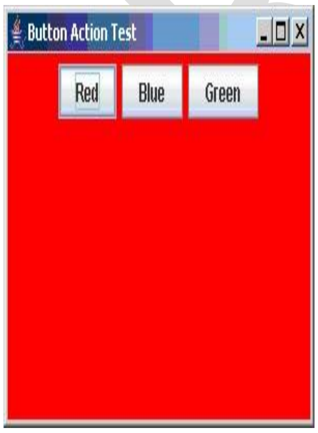
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TestButtonActions
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
      {
         public void run()
         {
            MyTestFrame frame = new MyTestFrame();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
         }
      });
   }
}

/**
 * A frame with a button panel
 */
class MyTestFrame extends JFrame
{
   public MyTestFrame()
   {
      setTitle("Button Action Test");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // create buttons
      JButton redButton = new JButton("Red");
      JButton blueButton = new JButton("Blue");
      JButton greenButton = new JButton("Green");

      buttonPanel = new JPanel();

      // add buttons to panel
      buttonPanel.add(redButton);
      buttonPanel.add(blueButton);
      buttonPanel.add(greenButton);

      // add panel to frame
      add(buttonPanel);

      // create button actions
      HandleActions greenAction = new HandleActions(Color.GREEN);
      HandleActions blueAction = new HandleActions(Color.BLUE);
      HandleActions redAction = new HandleActions(Color.RED);

      // associate actions with buttons
      greenButton.addActionListener(greenAction);
      blueButton.addActionListener(blueAction);
      redButton.addActionListener(redAction);
   }

   /**
    * An action listener that sets the panel's background color.
    */
   private class HandleActions implements ActionListener
   {
      public HandleActions(Color c)
      {
         backgroundColor = c;
      }

      public void actionPerformed(ActionEvent event)
```

```
         {
            buttonPanel.setBackground(backgroundColor);
         }

         private Color backgroundColor;
      }

      private JPanel buttonPanel;

      public static final int DEFAULT_WIDTH = 300;
      public static final int DEFAULT_HEIGHT = 200;
}
```

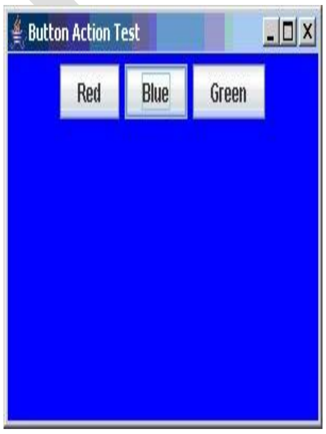
If you compile and run this class your output will be like below:



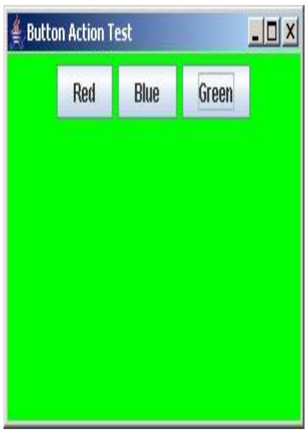
On click of Red



On click of blue:



On click of green :



There are several types of events and listeners in Java: each type of event is tied to a corresponding listener. For this discussion, let's consider a common type of event, an *action event* represented by the Java class *ActionEvent*, which is triggered when a user clicks a button or the item of a list.

At the user's action, an *ActionEvent* object corresponding to the relevant action is created. This object contains both the event source information and the specific action taken by the user. This event object is then passed to the corresponding *ActionListener* object's method:

void actionPerformed(ActionEvent e)

This method is executed and returns the appropriate GUI response, which might be to open or close a dialog, download a file, provide a digital signature, or any other of the myriad actions available to users in an interface.

Types of Events

Here are some of the most common types of events in Java:

- ***ActionEvent***: Represents a graphical element is clicked, such as a button or item in a list. Related listener: *ActionListener*.
- ***ContainerEvent***: Represents an event that occurs to the GUI's container itself, for example, if a user adds or removes an object from the interface. Related listener: *ContainerListener*.
- ***KeyEvent***: Represents an event in which the user presses, types or releases a key. Related listener: *KeyListener*.
- ***WindowEvent***: Represents an event relating to a window, for example, when a window is closed, activated or deactivated. Related listener: *WindowListener*.
- ***MouseEvent***: Represents any event related to a mouse, such as when a mouse is clicked or pressed. Related listener: *MouseListener*.

Note that multiple listeners and event sources can interact with one another. For example, multiple events can be registered by a single listener, if they are of the same type. This means that, for a similar set of components that perform the same type of action, one event listener can handle all the events. Similarly, a single event can be bound to multiple listeners, if that suits the program's design (although that is less common).

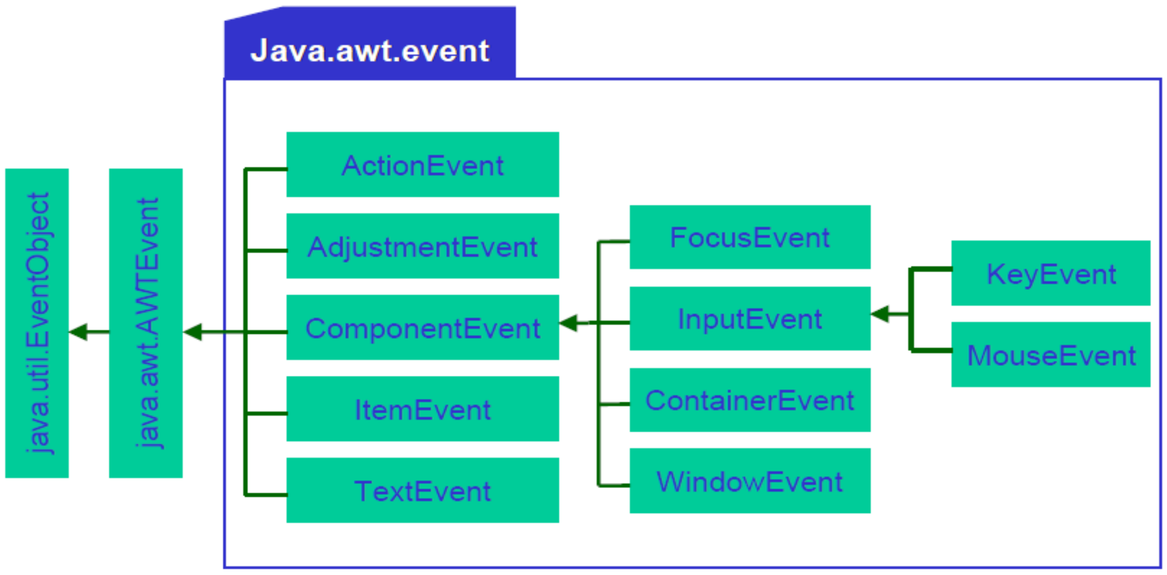


Figure. Event Hierarchy

Event Class	Listener Interface	Listener Methods
ActionEvent	ActionListener	actionPerformed()
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
ComponentEvent	ComponentListener	componentHidden()
		componentMoved()
		componentResized()
		componentShown()
ContainerEvent	ContainerListener	componentAdded()
		componentRemoved()
FocusEvent	FocusListener	focusGained()
		focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed()
		keyReleased()
		keyTyped()

MouseEvent	MouseListener	mouseClicked()
		mouseEntered()
		mouseExited()
		mousePressed()
		mouseReleased()
MouseMotionEvent	MouseMotionListener	mouseDragged()
		mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated()
		windowClosed()
		windowClosing()
		windowDeactivated()
		windowDeiconified()
		windowIconified()
		windowOpened()

Component	Events Generated	Meaning
Button	ActionEvent	User clicked on the button
Checkbox	ItemEvent	User selected or deselected an item
CheckboxMenuItem	ItemEvent	User selected or deselected an item
Choice	ItemEvent	User selected or deselected an item
Component	ComponentEvent	Component moved, resized, hidden, or shown
	FocusEvent	Component gained or lost focus
	KeyEvent	User pressed or released a key
	MouseEvent	User pressed or released mouse button, mouse entered or exited component, or user moved or dragged mouse. Note: MouseEvent has two corresponding listeners, MouseListener and MouseMotion Listener.

Container	**ContainerEvent**	**Component added to or removed from container**
List	ActionEvent	User double-clicked on list item
	ItemEvent	User selected or deselected an item
MenuItem	ActionEvent	User selected a menu item
Scrollbar	AdjustmentEvent	User moved the scrollbar
TextComponent	TextEvent	User changed text
TextField	ActionEvent	User finished editing text
Window	WindowEvent	Window opened, closed, iconified, deiconified, or close requested

MOUSE EVENT HANDLING

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

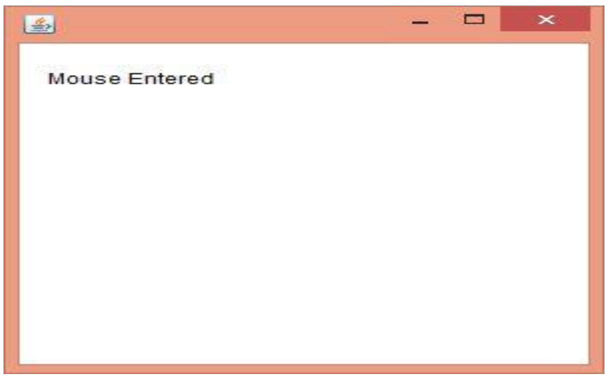
The signature of 5 methods found in MouseListener interface are given below:

- **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 MouseListener{
    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 MouseMotionListener Interface

The Java MouseMotionListener is notified whenever you move or drag mouse. It is notified against MouseEvent. The MouseMotionListener interface is found in java.awt.event package. It has two methods.

Methods of MouseMotionListener interface

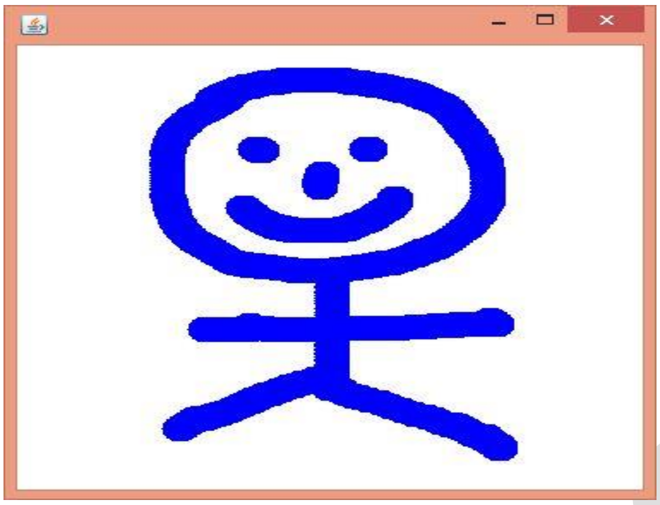
The signature of 2 methods found in MouseMotionListener interface are given below:

- **public abstract void** mouseDragged(MouseEvent e);
- **public abstract void** mouseMoved(MouseEvent e);

Java MouseMotionListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseMotionListenerExample extends Frame implements MouseMotionListen
er{
   MouseMotionListenerExample(){
      addMouseMotionListener(this);

      setSize(300,300);
      setLayout(null);
      setVisible(true);
   }
public void mouseDragged(MouseEvent e) {
   Graphics g=getGraphics();
   g.setColor(Color.BLUE);
   g.fillOval(e.getX(),e.getY(),20,20);
}
public void mouseMoved(MouseEvent e) {}

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

Output:



OTHER LISTENER INTERFACES

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.

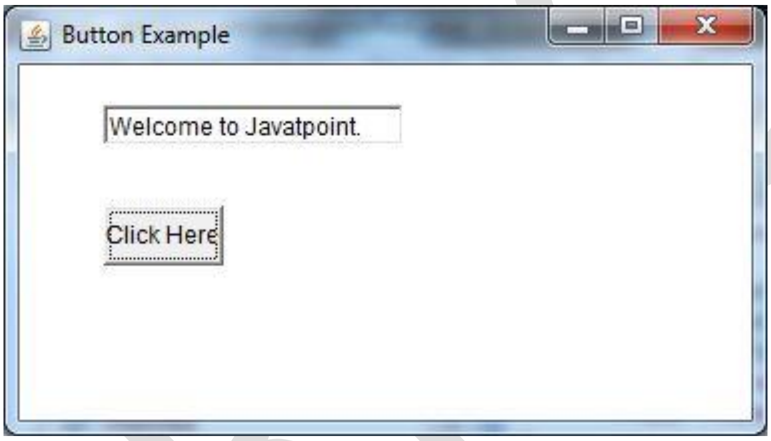
- **public abstract void** actionPerformed(ActionEvent e);

Java ActionListener Example: On Button click

```
import java.awt.*;
import java.awt.event.*;
public class ActionListenerExample {
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);

    b.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
            tf.setText("Welcome to Javatpoint.");
    }
    });
    f.add(b);f.add(tf);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



Java ItemListener Interface

The Java ItemListener is notified whenever you click on the checkbox. It is notified against ItemEvent. The ItemListener interface is found in java.awt.event package.

It has only one method: **itemStateChanged().**

itemStateChanged() method

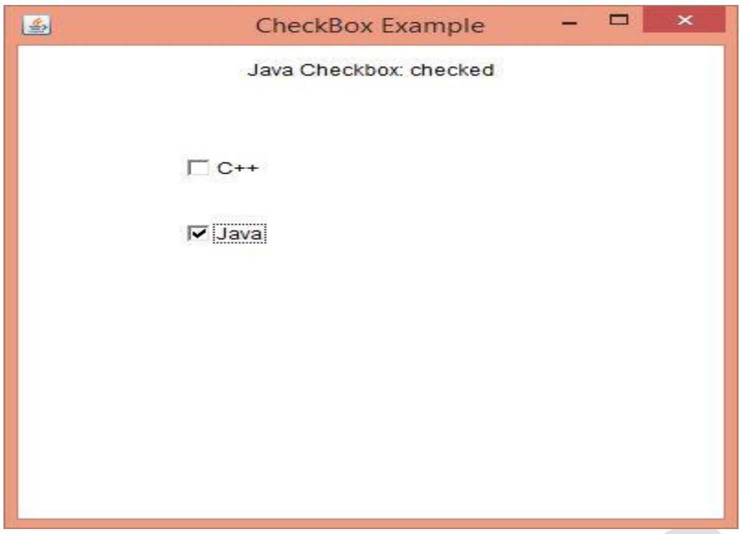
The itemStateChanged() method is invoked automatically whenever you click or unclick on the registered checkbox component.

- **public abstract void** itemStateChanged(ItemEvent e);

Java ItemListener Example

```
import java.awt.*;
import java.awt.event.*;
public class ItemListenerExample implements ItemListener{
   Checkbox checkBox1,checkBox2;
   Label label;
   ItemListenerExample(){
      Frame f= new Frame("CheckBox Example");
      label = new Label();
      label.setAlignment(Label.CENTER);
      label.setSize(400,100);
      checkBox1 = new Checkbox("C++");
      checkBox1.setBounds(100,100, 50,50);
      checkBox2 = new Checkbox("Java");
      checkBox2.setBounds(100,150, 50,50);
      f.add(checkBox1); f.add(checkBox2); f.add(label);
      checkBox1.addItemListener(this);
      checkBox2.addItemListener(this);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
   public void itemStateChanged(ItemEvent e) {
      if(e.getSource()==checkBox1)
         label.setText("C++ Checkbox: "
         + (e.getStateChange()==1?"checked":"unchecked"));
      if(e.getSource()==checkBox2)
      label.setText("Java Checkbox: "
      + (e.getStateChange()==1?"checked":"unchecked"));
    }
public static void main(String args[])
{
   new ItemListenerExample();
}   }
```

Output:



[Java KeyListener Interface]

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.

Methods of KeyListener interface

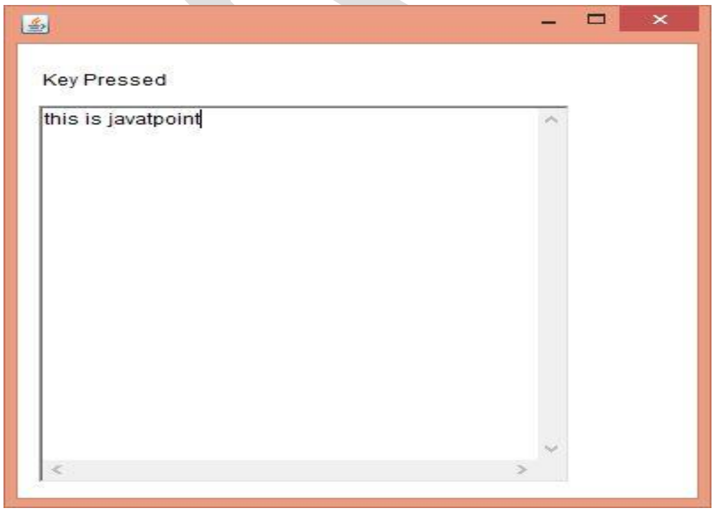
The signature of 3 methods found in KeyListener interface are given below:

- **public abstract void** keyPressed(KeyEvent e);
- **public abstract void** keyReleased(KeyEvent e);
- **public abstract void** keyTyped(KeyEvent e);

Java KeyListener Example

```
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener{
    Label l;
    TextArea area;
    KeyListenerExample(){

        l=new Label();
        l.setBounds(20,50,100,20);
        area=new TextArea();
        area.setBounds(20,80,300, 300);
        area.addKeyListener(this);

        add(l);add(area);
        setSize(400,400);
        setLayout(null);
        setVisible(true);
    }
    public void keyPressed(KeyEvent e) {
        l.setText("Key Pressed");
    }
    public void keyReleased(KeyEvent e) {
        l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e) {
        l.setText("Key Typed");
    }

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

Output:



```
Key Pressed
this is javatpoint
```

Java WindowListener Interface

The Java WindowListener is notified whenever you change the state of window. It is notified against WindowEvent. The WindowListener interface is found in java.awt.event package. It has three methods.

Methods of WindowListener interface

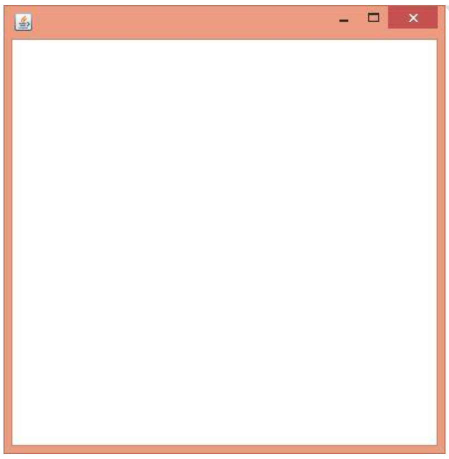
The signature of 7 methods found in WindowListener interface are given below:

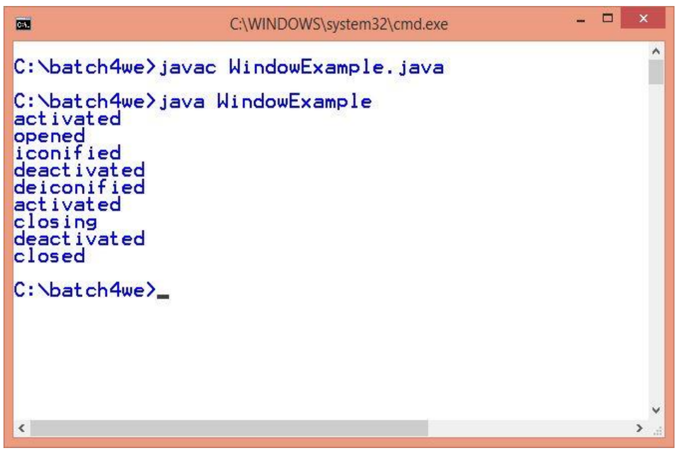
- **public abstract void** windowActivated(WindowEvent e);
- **public abstract void** windowClosed(WindowEvent e);
- **public abstract void** windowClosing(WindowEvent e);
- **public abstract void** windowDeactivated(WindowEvent e);
- **public abstract void** windowDeiconified(WindowEvent e);
- **public abstract void** windowIconified(WindowEvent e);
- **public abstract void** windowOpened(WindowEvent e);

Java WindowListener Example

```
import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
public class WindowExample extends Frame implements WindowListener{
   WindowExample(){
      addWindowListener(this);

      setSize(400,400);
      setLayout(null);
      setVisible(true);
   }

public static void main(String[] args) {
   new WindowExample();
}
public void windowActivated(WindowEvent arg0) {
   System.out.println("activated");
}
public void windowClosed(WindowEvent arg0) {
    System.out.println("closed");
}
public void windowClosing(WindowEvent arg0) {
    System.out.println("closing");
    dispose();
}
public void windowDeactivated(WindowEvent arg0) {
    System.out.println("deactivated");
}
public void windowDeiconified(WindowEvent arg0) {
    System.out.println("deiconified");
}
public void windowIconified(WindowEvent arg0) {
    System.out.println("iconified");
}
public void windowOpened(WindowEvent arg0) {
    System.out.println("opened");
}
}
```

Output:





```
C:\batch4we>javac WindowExample.java

C:\batch4we>java WindowExample
activated
opened
iconified
deactivated
deiconified
activated
closing
deactivated
closed

C:\batch4we>_
```

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*. 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

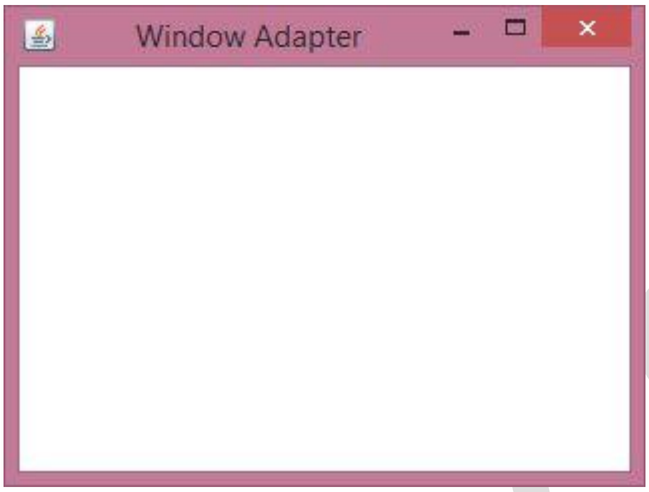
javax.swing.event Adapter classes

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Java WindowAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class AdapterExample{
   Frame f;
   AdapterExample(){
      f=new Frame("Window Adapter");
      f.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
            f.dispose();
         }
      });

      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
public static void main(String[] args) {
    new AdapterExample();
}
}
```

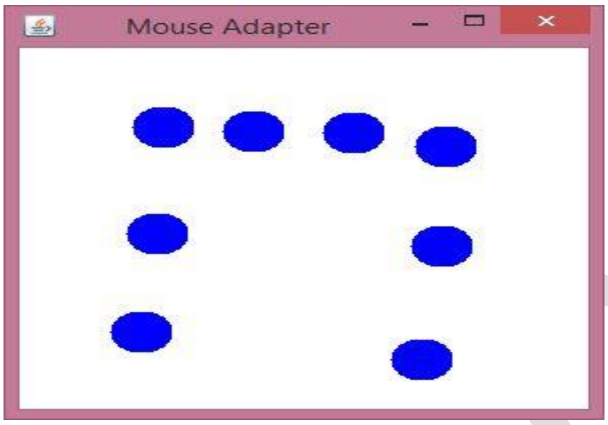
Output:



Java MouseAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseAdapterExample extends MouseAdapter{
    Frame f;
    MouseAdapterExample(){
        f=new Frame("Mouse Adapter");
        f.addMouseListener(this);

        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {
        Graphics g=f.getGraphics();
        g.setColor(Color.BLUE);
        g.fillOval(e.getX(),e.getY(),30,30);
    }

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

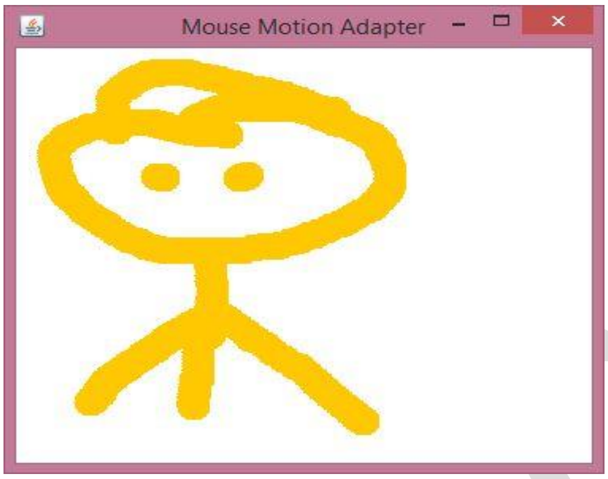
Output:



Java MouseMotionAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseMotionAdapterExample extends MouseMotionAdapter{
    Frame f;
    MouseMotionAdapterExample(){
        f=new Frame("Mouse Motion Adapter");
        f.addMouseMotionListener(this);

        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
public void mouseDragged(MouseEvent e) {
    Graphics g=f.getGraphics();
    g.setColor(Color.ORANGE);
    g.fillOval(e.getX(),e.getY(),20,20);
}
public static void main(String[] args) {
    new MouseMotionAdapterExample();
}
}
```

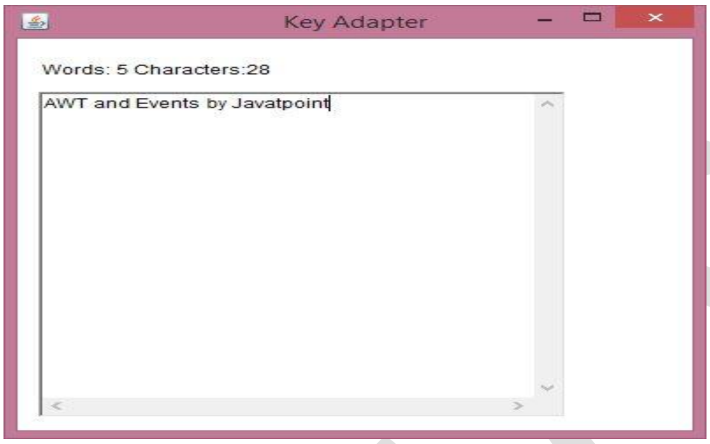
Output:



Java KeyAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class KeyAdapterExample extends KeyAdapter{
    Label l;
    TextArea area;
    Frame f;
    KeyAdapterExample(){
        f=new Frame("Key Adapter");
        l=new Label();
        l.setBounds(20,50,200,20);
        area=new TextArea();
        area.setBounds(20,80,300, 300);
        area.addKeyListener(this);
        f.add(l);f.add(area);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);      }
    public void keyReleased(KeyEvent e) {
        String text=area.getText();
        String words[]=text.split("\\s");
        l.setText("Words: "+words.length+" Characters:"+text.length());      }
   public static void main(String[] args) {
        new KeyAdapterExample();      }  }
```

Output:



```
Words: 5 Characters:28
AWT and Events by Javatpoint
```

Event Adapters

The listener classes that you define can extend adapter classes and override only the methods that you need.

Example:

```
import java.awt.* ;
import java.awt.event.* ;
public class MouseClickHandler extends MouseAdapter {
  public void mouseClicked( MouseEvent e) {
  // do stuff with the mouse click...
  }
}
```

Event Handling Using Anonymous Classes

You can include an entire class definition within the scope of an expression. This approach defines what is called an anonymous inner class and creates the instance all at once.

Example1

```
import java.awt.* ;
import java.awt.event.* ;
public class TestAnonymous {
    private Frame f ;
    private TextField tf;
    public TestAnonymous() {
        f = new Frame("Anonymous class example") ;
        tf = new TextField(30) ;
    }
    public void launchFrame() {
        Label label = new Label("Click and drag the mouse") ;
        f.add(label, BorderLayout.NORTH) ;
        f.add(tf, BorderLayout.SOUTH) ;
        f.addMouseMotionListener(new MouseMotionAdapter()
        {
                public void mouseDragged( MouseEvent e){
                String s = "Mouse dragging: X= " + e.getX() + "Y= " + e.getY() ;
                tf.setText(s) ;
        }
    }) ;
    f.addMouseListener( new MouseClickHandler(tf) ) ;
    f.setSize(300,200) ;
    f.setVisible(true) ;
}
    public static void main(String[] args)
    {
        TestAnonymous obj = new TestAnonymous() ;
        obj.launchFrame();
    }
}
```

Example2

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

```
public class MouseClickHandler extends MouseAdapter {
        private TextField tf ;
        public static int count = 0 ;
        public MouseClickHandler(TextField tf) {
        this.tf = tf ;
        }
        public void mouseClicked( MouseEvent e) {
        count++;
        String s = "Mouse has been clicked " + count + " times so far." ;
        tf.setText(s) ;
        }
}
```

Event Handling Using Inner Classes

You can implement event handlers as inner class. This allows access to the private data of the outer class.

Example:

```
import java.awt.* ;
import java.awt.event.* ;
public class TestInner {
private Frame f ;
private TextField tf;
   public TestInner() {
        f = new Frame("Inner classes example") ;
        tf = new TextField(30) ;
   }
   public void launchFrame() {
        Label label = new Label("Click and drag the mouse") ;
        f.add(label, BorderLayout.NORTH) ;
        f.add(tf, BorderLayout.SOUTH) ;
        f.addMouseMotionListener(new MyMouseMotionListener()) ;
        f.addMouseListener(new MouseClickHandler(tf)) ;
        f.setSize(300,200) ;
        f.setVisible(true) ;
    }
    class MyMouseMotionListener extends MouseMotionAdapter {
    public void mouseDragged( MouseEvent e ) {
        String s = "Mouse dragging: X= " + e.getX() + "Y= " + e.getY() ;
        tf.setText(s) ;
    }
public static void main(String[] args) {
        TestInner obj = new TestInner() ;
        obj.launchFrame();
}
}
```

AWT EVENT HIERARCHY

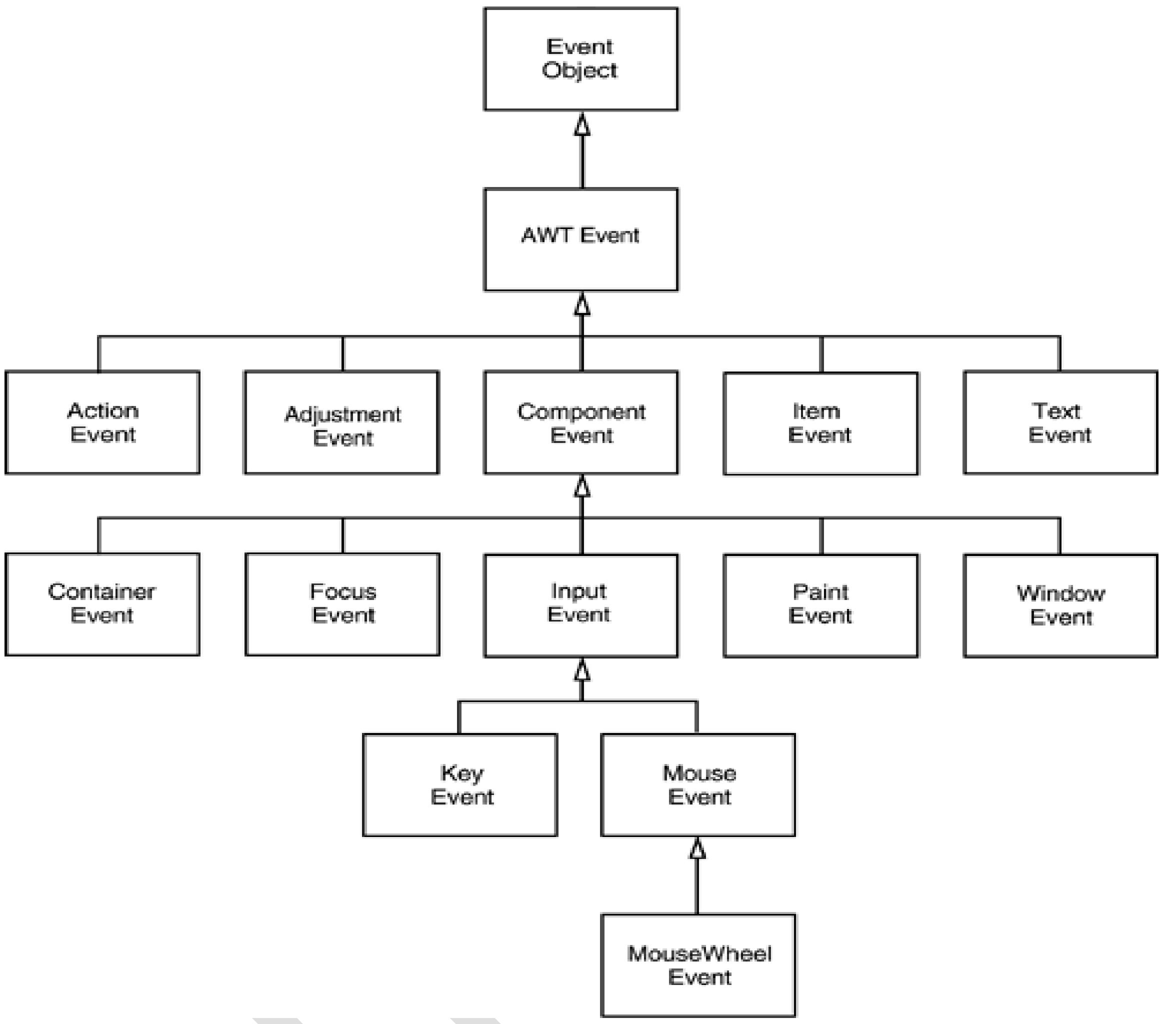
Event handling in Java is object oriented, with all events descending from the EventObject class in the java .util package. (The common superclass is not called Event because that is the name of the event class in the old event model. Although the old model is now deprecated, its classes are still a part of the Java library.)

Some of the Swing components generate event objects of yet more event types; these directly extend Event Object, not AWT Event.

The event objects encapsulate information about the event that the event source communicates to its listeners. When necessary, you can then analyze the event objects that passed to the listener object, as we did in the button example with the getSource and get Action Command methods.

Some of the AWT event classes are of no practical use for the Java programmer. For example, the AWT inserts Paint Event objects into the event queue, but these objects are not delivered to listeners. Java programmers don’t listen to paint events; they override the paint Component method to control repainting.

The Event Object class has a subclass AWTEvent, which is the parent of all AWT event classes.



Here are the most commonly used semantic event classes in the java.awt.event package:

- ActionEvent (for a button click, a menu selection, selecting a list item, or ENTER typedin a text field)
- AdjustmentEvent (the user adjusted a scrollbar)
- ItemEvent (the user made a selection from a set of checkbox or list items)

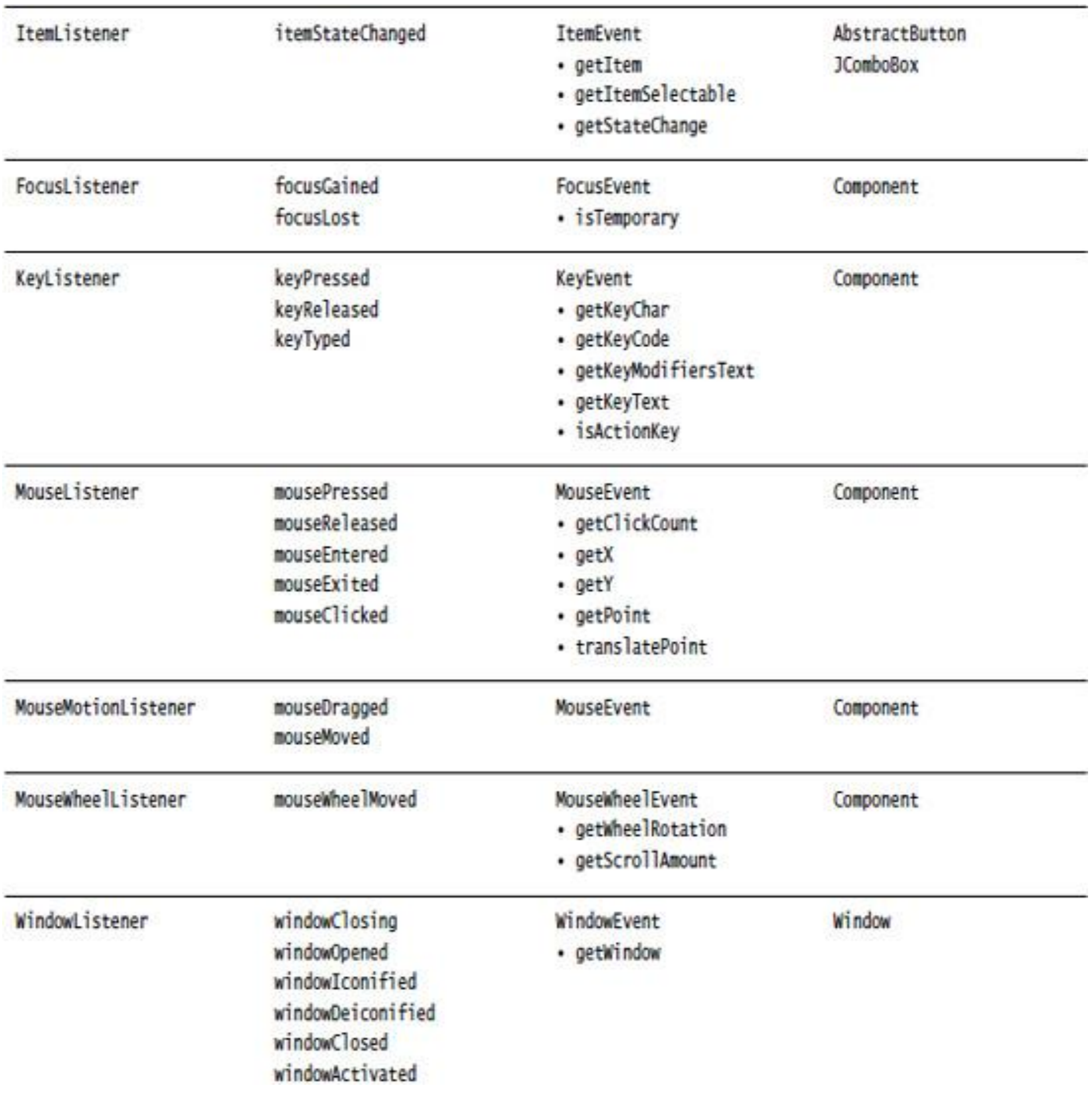
Five low-level event classes are commonly used:

- KeyEvent (a key was pressed or released)
- MouseEvent (the mouse button was pressed, released, moved, or dragged)
- MouseWheelEvent (the mouse wheel was rotated)
- FocusEvent (a component got focus or lost focus)
- WindowEvent (the window state changed)

The following interfaces listen to these events:

- ActionListener MouseMotionListener
- AdjustmentListener MouseWheelListener
- FocusListener WindowListener
- ItemListener WindowFocusListener
- KeyListener WindowStateListener
- MouseListener

Event Handling Summary



ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated	WindowEvent • getWindow	Window

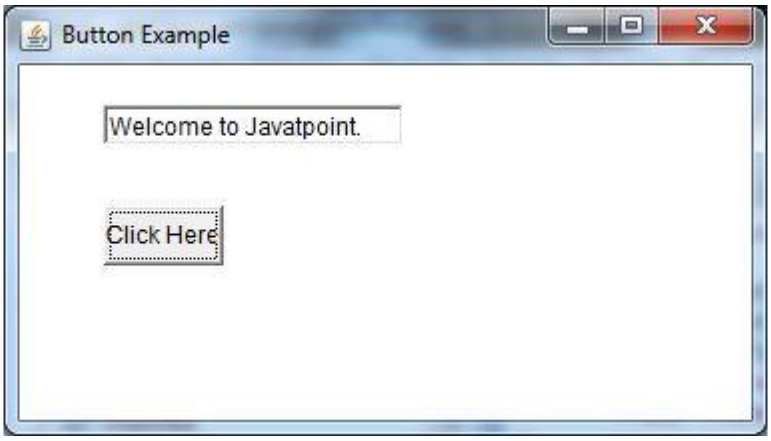
EVENT HANDLING FOR JAVA AWT COMPONENTS EXAMPLES

AWT Component	Events Generated
Button	ActionEvent
CheckBox	ItemEvent
ScrollBar	AdjustmentEvent

Java AWT Button Example with ActionListener

```
import java.awt.*;
import java.awt.event.*;
public class ButtonExample {
public static void main(String[] args) {
   Frame f=new Frame("Button Example");
   final TextField tf=new TextField();
   tf.setBounds(50,50, 150,20);
   Button b=new Button("Click Here");
   b.setBounds(50,100,60,30);
   b.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
         tf.setText("Welcome to Javatpoint.");
      }
   });
   f.add(b);f.add(tf);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
}
}
```

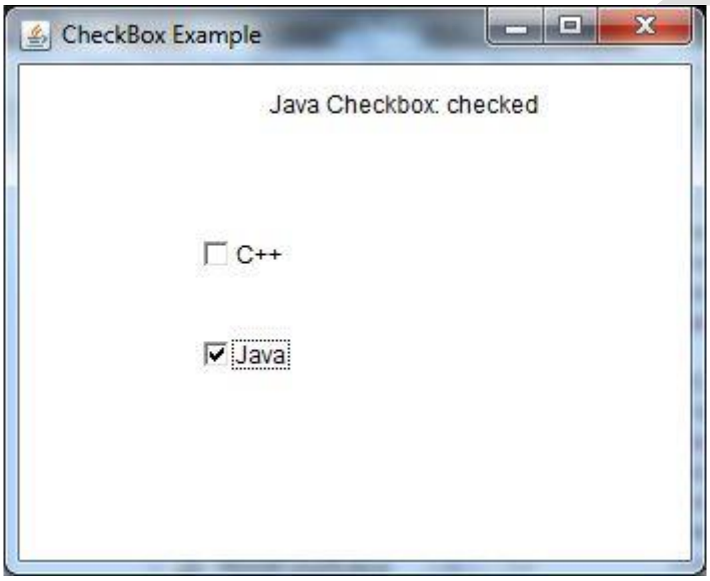
Output:



Java AWT Checkbox Example with ItemListener

```
import java.awt.*;
import java.awt.event.*;
public class CheckboxExample
{
        CheckboxExample(){
        Frame f= new Frame("CheckBox Example");
        final Label label = new Label();
        label.setAlignment(Label.CENTER);
        label.setSize(400,100);
        Checkbox checkbox1 = new Checkbox("C++");
        checkbox1.setBounds(100,100, 50,50);
        Checkbox checkbox2 = new Checkbox("Java");
        checkbox2.setBounds(100,150, 50,50);
        f.add(checkbox1); f.add(checkbox2); f.add(label);
        checkbox1.addItemListener(new ItemListener() {
          public void itemStateChanged(ItemEvent e) {
             label.setText("C++ Checkbox: "+ (e.getStateChange()==1?"checked":"unchecked"));
               }
          });
        checkbox2.addItemListener(new ItemListener() {
          public void itemStateChanged(ItemEvent e) {
            label.setText("Java Checkbox: "+ (e.getStateChange()==1?"checked":"unchecked"));
              }
          });
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
   new CheckboxExample();
}
}
```

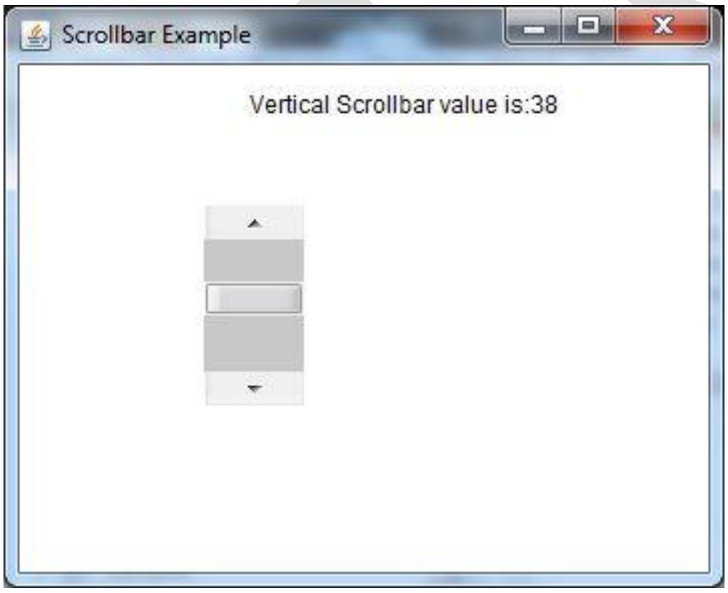
Output:



Java AWT Scrollbar Example with AdjustmentListener

```
import java.awt.*;
import java.awt.event.*;
class ScrollbarExample{
     ScrollbarExample(){
            Frame f= new Frame("Scrollbar Example");
            final Label label = new Label();
            label.setAlignment(Label.CENTER);
            label.setSize(400,100);
            final Scrollbar s=new Scrollbar();
            s.setBounds(100,100, 50,100);
            f.add(s);f.add(label);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
            s.addAdjustmentListener(new AdjustmentListener() {
               public void adjustmentValueChanged(AdjustmentEvent e) {
                  label.setText("Vertical Scrollbar value is:"+ s.getValue());
               }
            });
         }
public static void main(String args[]){
new ScrollbarExample();
}
}
```

Output:



Vertical Scrollbar value is:38

Introduction to Swing – layout management - Swing Components – Text Fields , Text Areas – Buttons- Check Boxes – Radio Buttons – Lists- choices- Scrollbars – Windows – Menus – Dialog Boxes

INTRODUCTION TO SWING

- The original Java GUI subsystem was the Abstract Window Toolkit (AWT).
- AWT translates it visual components into platform-specific equivalents (peers).
- Under AWT, the look and feel of a component was defined by the platform.
- AWT components are referred to as **heavyweight**.
- Swing was introduced in 1997 to fix the problems with AWT.
- Swing offers two key features:
  1. Swing components are **lightweight** and don't rely on peers.
  2. Swing supports a pluggable look and feel. The three PLAFs available to all users are Metal (default), Windows, and Motif.
- Swing is built on AWT.

Java Foundation Classes (JFC)

The Java Foundation Classes (JFC) are a suite of libraries designed to assist programmers in creating enterprise applications with Java. The Swing API is only one of five libraries that make up the JFC. The Java Foundation Classes also consist of the Abstract Window Toolkit (AWT), the Accessibility API, the 2D API, and enhanced support for drag-and-drop capabilities.

AWT

The Abstract Window Toolkit is the basic GUI toolkit shipped with all versions of the Java Development Kit. While Swing does not reuse any of the older AWT components, it does build off of the lightweight component facilities introduced in AWT 1.1.

Accessibility

The accessibility package provides assistance to users who have trouble with traditional user interfaces. Accessibility tools can be used in conjunction with devices such as audible text readers or braille keyboards to allow direct access to the Swing components. Accessibility is split into two parts: the Accessibility API, which is shipped with the Swing distribution, and the Accessibility Utilities API, distributed separately. All Swing components contain support for accessibility.

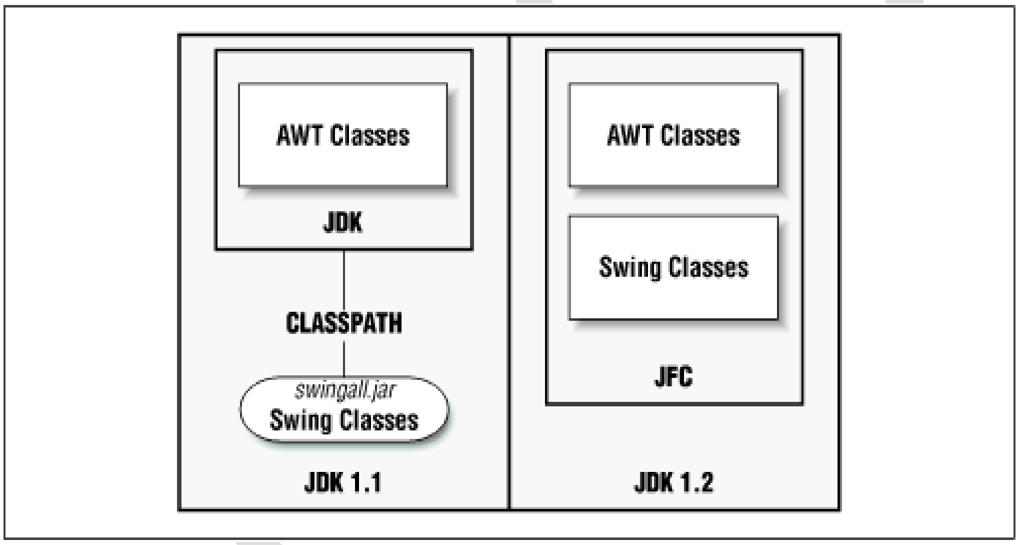
2D *API*

The 2D API contains classes for implementing various painting styles, complex shapes, fonts, and colors. This Java package is loosely based on APIs that were licensed from IBM's Taligent division.

Drag and Drop

Drag and drop is one of the more common metaphors used in graphical interfaces today. The user is allowed to click and "hold" a GUI object, moving it to another window or frame in the desktop with predictable results. The Drag and Drop API allows users to implement droppable elements that transfer information between Java applications and native applications.

Relationship between Swing and AWT



Swing Features

Swing provides many new features for those planning to write large-scale applications in Java. Here is an overview of some of the more popular features.

1. Pluggable Look-and-Feels

One of the most exciting aspects of the Swing classes is the ability to dictate the *look-and-feel* (L&F) of each of the components, even resetting the look-and-feel at runtime. Look-and-feels have become an important issue in GUI development over the past five years.

2. Lightweight Components

Most Swing components are *lightweight* . In the purest sense, this means that components are not dependent on native peers to render themselves. Instead, they use simplified graphics primitives to paint themselves on the screen and can even allow portions to be transparent. The ability to create lightweight components first emerged in JDK 1.1, although the majority of AWT components did not take advantage of it. Prior to that, Java programmers had no choice but to subclass java.awt.Canvas or java.awt.Panel if they wished to create their own components.

3. Additional Features

Several other features distinguish Swing from the older AWT components:

- A wide variety of new components, such as tables, trees, sliders, progress bars, internal frames, and text components.
- Swing components contain support for replacing their insets with an arbitrary number of concentric borders.
- Swing components can have *tooltips* placed over them. A tooltip is a textual popup that momentarily appears when the mouse cursor rests inside the component's painting region. Tooltips can be used to give more information about the component in question.
- You can arbitrarily bind keyboard events to components, defining how they will react to various keystrokes under given conditions.
- There is additional debugging support for the rendering of your own lightweight Swing components.

Swing Packages and Classes

javax.accessibility - Contains classes and interfaces that can be used to allow *assistive technologies* to interact with Swing components.

javax.swing - Contains the core Swing components, including most of the model interfaces and support classes.

javax.swing.border - Contains the definitions for the abstract border class as well as eight predefined borders. Borders are not components; instead, they are special graphical elements that Swing treats as properties and places around components in place of their insets.

javax.swing.colorchooser - Contains support for the JColorChooser component

javax.swing.filechooser - Contains support for the JFileChooser component

javax.swing.pending - Contains an assortment of components that aren't ready for prime time, but may be in the future.

javax.swing.plaf - Defines the unique elements that make up the pluggable look-and-feel for each Swing component. Its various subpackages are devoted to rendering the individual look-and-feels for each component on a platform-by-platform basis.

javax.swing.table - Provides models and views for the table component. The table component allows you to arrange various information in a grid-based format with an appearance similar to a spreadsheet. Using the lower-level classes, you can manipulate how tables are viewed and selected, as well as how they display their information in each cell.

javax.swing.text - Provides scores of text-based classes and interfaces supporting a common design known as *document/view*.

javax.swing.text.html - Used specifically for reading and formatting HTML text through an ancillary editor kit.

javax.swing.text.html.parser - Contains support for parsing HTML.

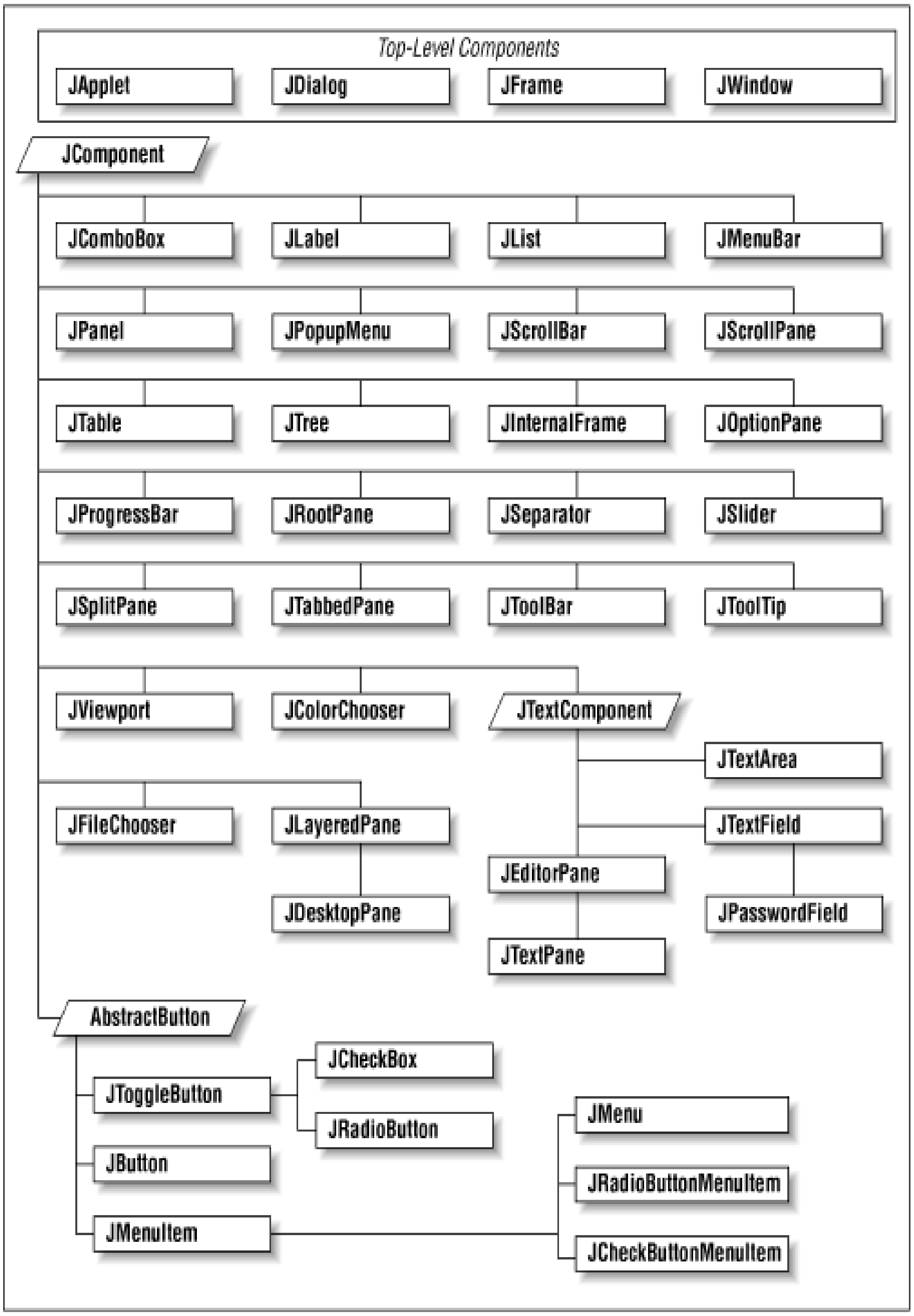
javax.swing.text.rtf - Used specifically for reading and formatting the Rich Text Format (RTF) text through an ancillary editor kit.

javax.swing.tree - Defines models and views for a hierarchal tree component, such as you might see representing a file structure or a series of properties.

javax.swing.undo - Contains the necessary functionality for implementing undoable functions.

Class Hierarchy

Figure below shows a detailed overview of the Swing class hierarchy as it appears in the 1.2 JDK. At first glance, the class hierarchy looks very similar to AWT. Each Swing component with an AWT equivalent shares the same name, except that the Swing class is preceded by a capital "J". In most cases, if a Swing component supersedes an AWT component, it can be used as a drop-in replacement.



THE MODEL-VIEW-CONTROLLER ARCHITECTURE

Swing uses the *model-view-controller architecture* (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.

Model

The model encompasses the state data for each component. There are different models for different types of components. For example, the model of a scrollbar component might contain information about the current position of its adjustable "thumb," its minimum and maximum values, and the thumb's width (relative to the range of values). A menu, on the other hand, may simply contain a list of the menu items the user can select from. Note that this information remains the same no matter how the component is painted on the screen; model data always exists independent of the component's visual representation.

View

The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms. Almost all window frames will have a titlebar spanning the top of the window. However, the titlebar may have a close box on the left side (like the older MacOS platform), or it may have the close box on the right side (as in the Windows 95 platform). These are examples of different types of views for the same window object.

Controller

The controller is the portion of the user interface that dictates how the component interacts with events. Events come in many forms — a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component will react to the event—if it reacts at all.

Figure below shows how the model, view, and controller work together to create a scrollbar component. The scrollbar uses the information in the model to determine how far into the scrollbar to render the thumb and how wide the thumb should be. Note that the model specifies this information relative to the minimum and the maximum. It does not give the position or width of the thumb in screen pixels—the view calculates that. The view determines exactly where and how to draw the scrollbar, given the proportions offered by the model. The view knows whether it is a horizontal or vertical scrollbar, and it knows exactly how to shadow the end buttons and the thumb. Finally, the controller is responsible for handling mouse events on the component. The controller knows, for example, that dragging the thumb is a legitimate action for a scroll bar, and pushing on the end buttons is acceptable as well. The result is a fully functional MVC scrollbar.

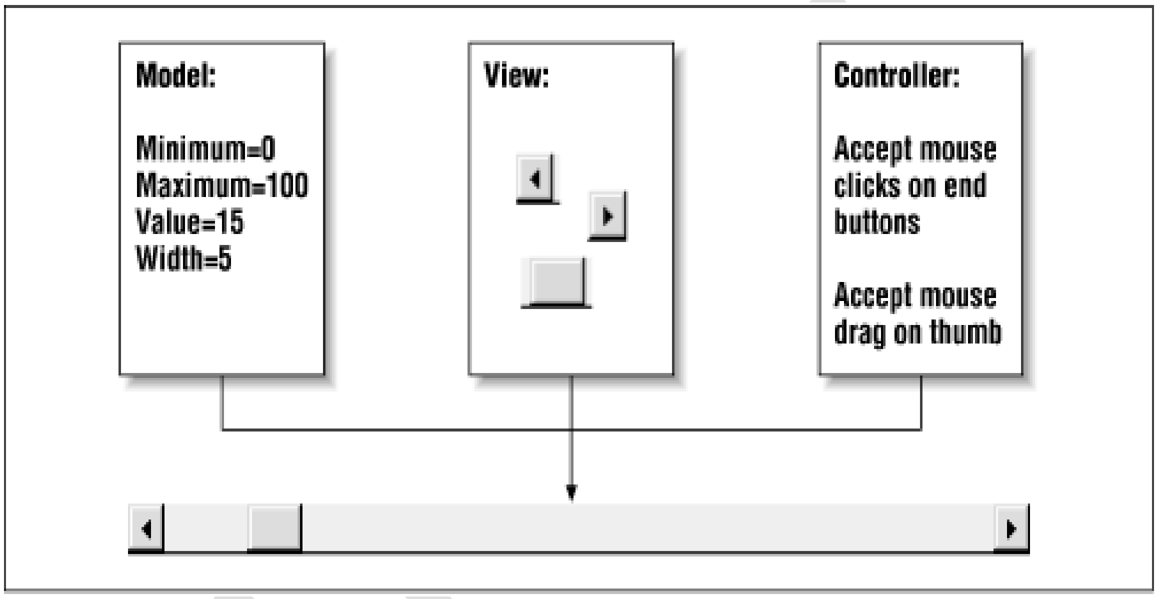

Model:
Minimum=0
Maximum=100
Value=15
Width=5
View:
Controller:
Accept mouse clicks on end buttons
Accept mouse drag on thumb

Figure. *The three elements of a model-view-controller architecture*

MVC Interaction

With MVC, each of the three elements—the model, the view, and the controller—requires the services of another element to keep itself continually updated.

We already know that the view cannot render the scrollbar correctly without obtaining information from the model first. In this case, the scrollbar will not know where to draw its "thumb" unless it can obtain its current position and width relative to the minimum and maximum. Likewise, the view determines if the component is the recipient of user events, such as mouse clicks. (For example, the view knows the exact width of the thumb; it can tell whether a click occurred over the thumb or just outside of it.) The view passes these events on to the controller, which decides how to handle them best. Based on the controller's decisions, the values in the model may need to be altered. If the user drags the scrollbar thumb, the controller will react by incrementing the thumb's position in the model. At that point, the whole cycle can repeat. The three elements, therefore, communicate their data.

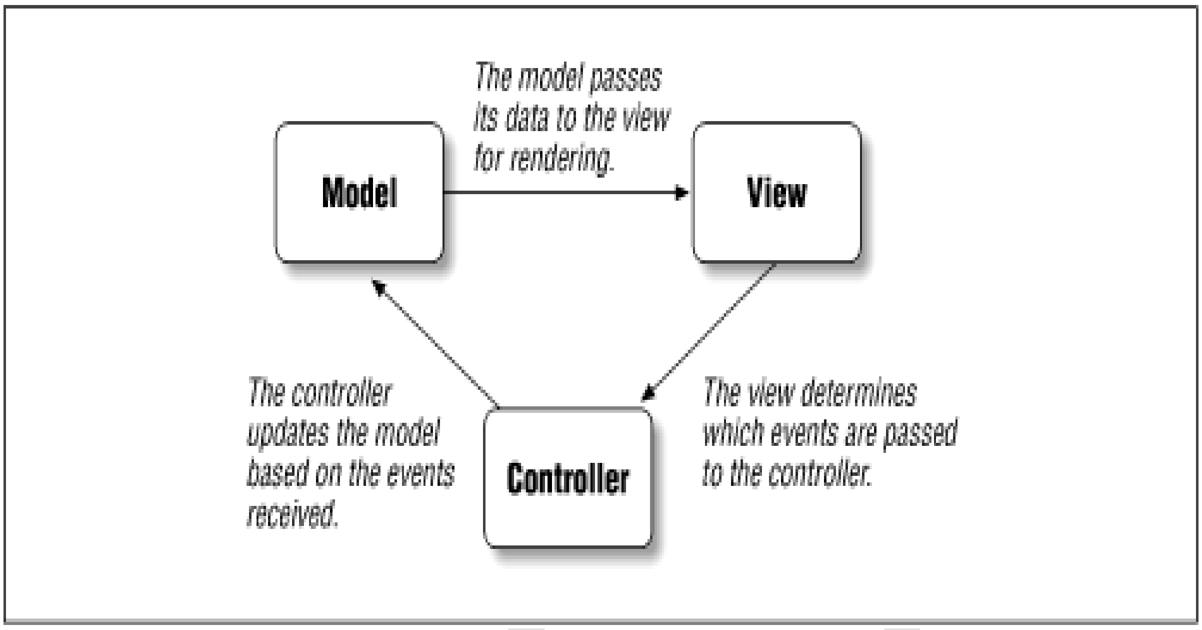

The model passes its data to the view for rendering.
Model
View
The controller updates the model based on the events received.
Controller
The view determines which events are passed to the controller.

Figure. Communication through model-view-controller architecture

MVC in Swing

Swing actually makes use of a simplified variant of the MVC design called the *model-delegate*. This design combines the view and the controller object into a single element that draws the component to the screen and handles GUI events known as the *UI delegate*.

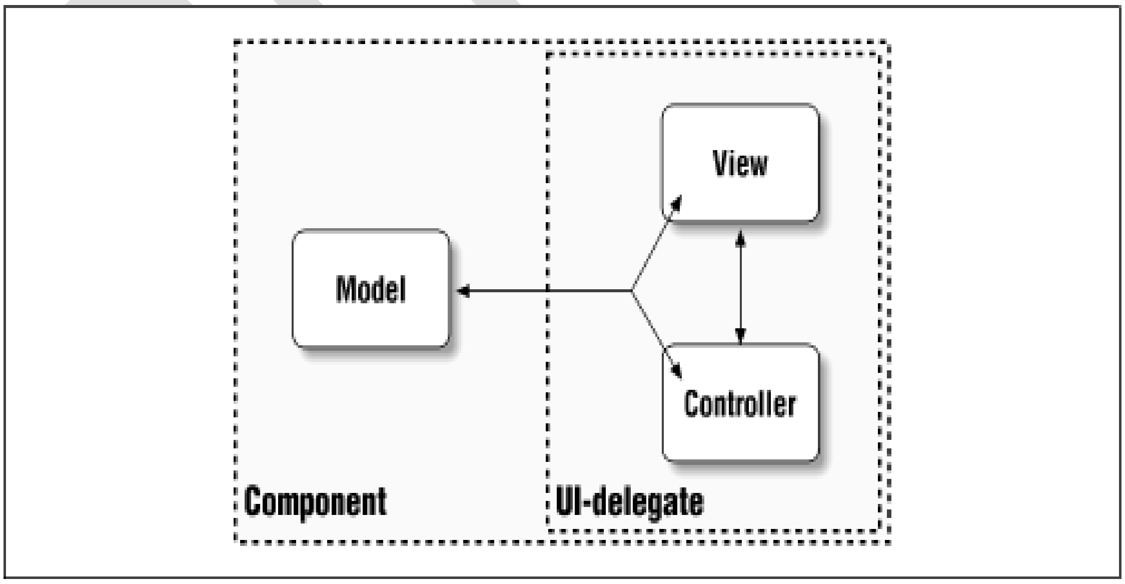


Figure. *With Swing, the view and the controller are combined into a UI-delegate*

LAYOUT MANAGEMENT

Swing has plenty of layout managers available — both built-in and third-party. However, most of the managers are not suitable in modern UI creation.

There are three layout managers that can do the job properly:

- MigLayout
- GroupLayout
- FormLayout

MigLayout, GroupLayout, and FormLayout are powerful, flexible layout managers that can cope with most layout requirements.

The following layout managers are obsolete:

- FlowLayout
- GridLayout
- CardLayout
- BoxLayout
- GridBagLayout

Problems with obsolete managers

Obsolete managers are either too simple (FlowLayout, GridLayout) or unnecessary complex (GridBagLayout). All these managers have a fundamental design error: *they use fixed gaps between components*.

Obsolete managers try to fix their weaknesses by a technique called nesting. In nesting, developers use several different layout managers in multiple panels. While it is possible to create an UI with nesting, it brings additional unnecessary complexity to the code.

Obsolete managers

FlowLayout manager

This is the simplest layout manager in the Java Swing toolkit. It is the default layout manager for the JPanel component. It is so simple that it cannot be used for any real layout.

FlowLayout()

FlowLayout(int align)

FlowLayout(int align, int hgap, int vgap)

There are three constructors available for the FlowLayout manager. The first one creates a manager with implicit values. Centered with 5px horizontal and vertical spaces. The others allow to specify those parameters.

Example: FlowLayoutEx.java

```
import java.awt.Dimension;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTree;
import javax.swing.SwingUtilities;
public class FlowLayoutEx extends JFrame {
public FlowLayoutEx() {
initUI();
}
private void initUI() {
JPanel panel = new JPanel();
JButton button = new JButton("button");
panel.add(button);
JTree tree = new JTree();
panel.add(tree);
JTextArea area = new JTextArea("text area");
area.setPreferredSize(new Dimension(100, 100));
panel.add(area);
add(panel);
pack();
setTitle("FlowLayout Example");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
}
public static void main(String[] args) {
```

```
SwingUtilities.invokeLater(() -> {
FlowLayoutEx ex = new FlowLayoutEx();
ex.setVisible(true);
});
}
}
```

The example shows a button, a tree component, and a text area component in the window. If we create a empty tree component, there are some default values inside the component.

```
        JPanel panel = new JPanel();
```

The implicit layout manager of the JPanel component is a flow layout manager. We do not have to set it manually.

```
        JTextArea area = new JTextArea("text area");
        area.setPreferredSize(new Dimension(100, 100));
```

The flow layout manager sets a preferred size for its components. This means, that in our case, the area component will have 100x100px. If we didn't set the preferred size, the component would have a size of its text. The component will grow and shrink accordingly.

```
        panel.add(area);
```

To put a component inside a container, we simply call the add() method.

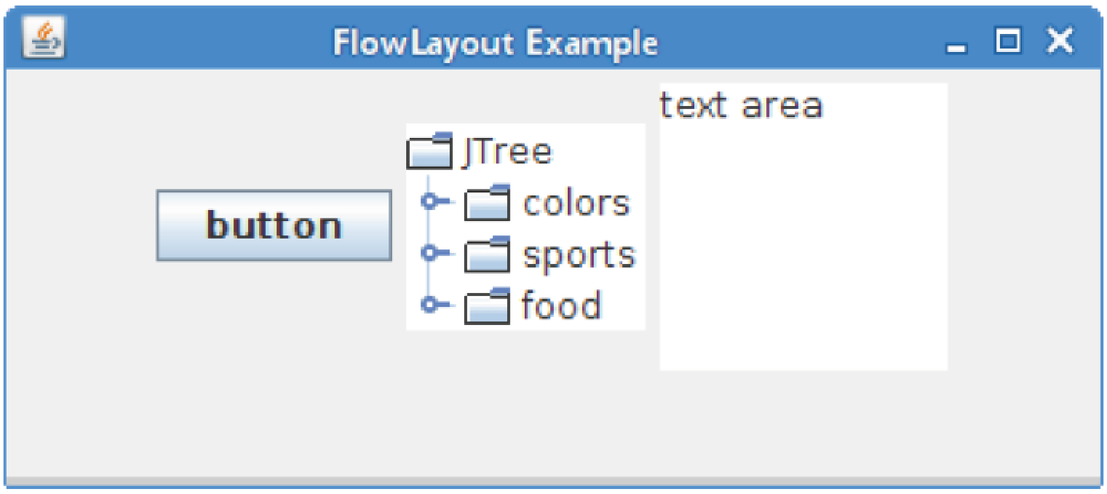


Figure: FlowLayout

GridLayout

The GridLayout layout manager lays out components in a rectangular grid. The container is divided into equally sized rectangles. One component is placed in each rectangle. GridLayout is very simple and cannot be used for any real layout.

Example: GridLayoutEx.java

```
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
public class GridLayoutEx extends JFrame {
public GridLayoutEx() {
initUI();
}
private void initUI() {
JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
panel.setLayout(new GridLayout(5, 4, 5, 5));
String[] buttons = {
"Cls", "Bck", "", "Close", "7", "8", "9", "/", "4",
"5", "6", "*", "1", "2", "3", "-", "0", ".", "=", "+"
};
for (int i = 0; i < buttons.length; i++) {
if (i == 2) {
panel.add(new JLabel(buttons[i]));
} else {
panel.add(new JButton(buttons[i]));
}
}
add(panel);
setTitle("GridLayout");
setSize(350, 300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
}
public static void main(String[] args) {
SwingUtilities.invokeLater(() -> {
GridLayoutEx ex = new GridLayoutEx();
ex.setVisible(true);
});
}
}
```

The example shows a skeleton of a simple calculator tool. We put nineteen buttons and one label into the manager. Notice that each button is of the same size.

```
        panel.setLayout(new GridLayout(5, 4, 5, 5));
```

Here we set the grid layout manager for the panel component. The layout manager takes four parameters. The number of rows, the number of columns and the horizontal and vertical gaps between components.

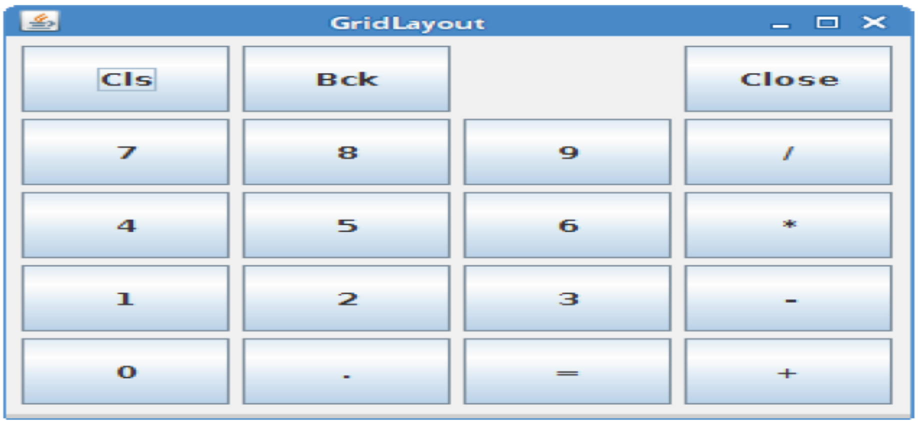


Figure: GridLayout

BorderLayout

BorderLayout is a simple layout manager that can be handy in certain layouts. It is a default layout manager for JFrame, JWindow, JDialog, JInternalFrame, and JApplet. It has a serious limitiation — it sets the gaps between its children in pixels, thus creating rigid layouts. This leads to non-portable UI, and therefore, its usage is not recommended.

BorderLayout divides the space into five regions: north, west, south, east, and centre. Each region can have only one component. If we need to put more components into a region, we have to put a panel there with a manager of our choice. The components in N, W, S, E regions get their preferred size. The component in the centre takes up the whole space left.

Example: BorderEx.java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Insets;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.border.EmptyBorder;
public class BorderEx extends JFrame {
public BorderEx() {
initUI();
}
private void initUI() {
JPanel bottomPanel = new JPanel(new BorderLayout());
JPanel topPanel = new JPanel();
topPanel.setBackground(Color.gray);
topPanel.setPreferredSize(new Dimension(250, 150));
bottomPanel.add(topPanel);
bottomPanel.setBorder(new EmptyBorder(new Insets(20, 20, 20, 20)));
add(bottomPanel);
pack();
setTitle("Border Example");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
}
public static void main(String[] args) {
SwingUtilities.invokeLater(() -> {
BorderEx ex = new BorderEx();
ex.setVisible(true);
});
}
}
```

The example will display a gray panel and border around it.

```
JPanel bottomPanel = new JPanel(new BorderLayout());
JPanel topPanel = new JPanel();
```

We place a panel into a panel. The bottom panel has a BorderLayout manager.

```
bottomPanel.add(topPanel);
```

Here we placed the top panel into the bottom panel component. More precisely, we placed it into the center area of its BorderLayout manager.

```
bottomPanel.setBorder(new EmptyBorder(new Insets(20, 20, 20, 20)));
```

Here we created a 20px border around the bottom panel. The border values are as follows: top, left, bottom and right. Note that creating fixed insets (spaces) is not portable.

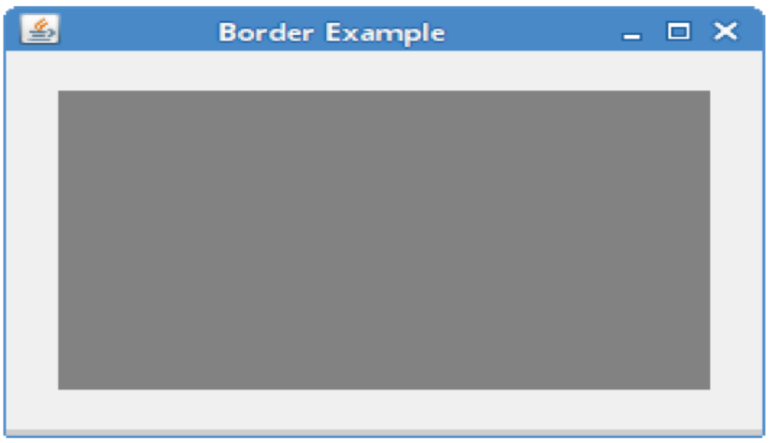


Figure: Border example

BoxLayout

BoxLayout manager is a simple layout manager that organizes components in a column or a row. It can create quite sophisticated layouts with nesting. However, this raises the complexity of the layout creation and uses additional resources, notably many other JPanel components. BoxLayout is only able to create fixed spaces; therefore, its layouts are not portable.

BoxLayout has the following constructor:

```
BoxLayout(Container target, int axis)
```

The constructor creates a layout manager that will lay out components along the given axis. Unlike other layout managers, BoxLayout takes a container instance as the first parameter in the constructor. The second parameter determines the orientation of the manager. To create a horizontal box, we can use the LINE_AXIS constant. To create a vertical box, we can use the PAGE_AXIS constant. The box layout manager is often used with the Box class. This class creates several invisible components which affect the final layout.

glue

strut

rigid area

Let's say we want to put two buttons into the right bottom corner of the window.

Example: BoxLayoutButtonsEx.java

```
import java.awt.Dimension;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
public class BoxLayoutButtonsEx extends JFrame {
public BoxLayoutButtonsEx() {
initUI();
}
private void initUI() {
JPanel basic = new JPanel();
basic.setLayout(new BoxLayout(basic, BoxLayout.Y_AXIS));
add(basic);
basic.add(Box.createVerticalGlue());
JPanel bottom = new JPanel();
bottom.setAlignmentX(1f);
bottom.setLayout(new BoxLayout(bottom, BoxLayout.X_AXIS));
JButton ok = new JButton("OK");
JButton close = new JButton("Close");
bottom.add(ok);
bottom.add(Box.createRigidArea(new Dimension(5, 0)));
bottom.add(close);
bottom.add(Box.createRigidArea(new Dimension(15, 0)));
basic.add(bottom);
basic.add(Box.createRigidArea(new Dimension(0, 15)));
setTitle("Two Buttons");
setSize(300, 150);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
}
public static void main(String[] args) {
SwingUtilities.invokeLater(() -> {
BoxLayoutButtonsEx ex = new BoxLayoutButtonsEx();
ex.setVisible(true);
});
}
}
```

The following drawing illustrates the example.

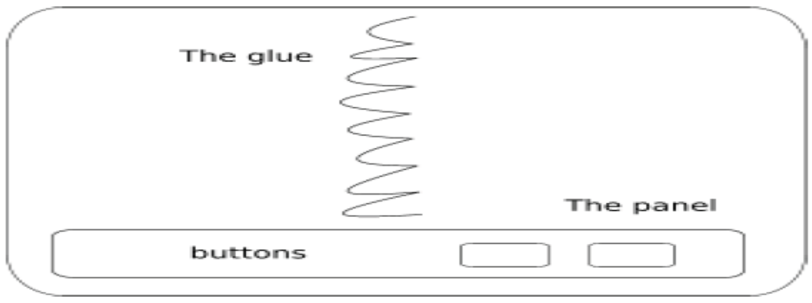


Figure: Two buttons

We create two panels. The basic panel has a vertical box layout. The bottom panel has a horizontal one. We will put a bottom panel into the basic panel. We right align the bottom panel. The space between the top of the window and the bottom panel is expandable. It is done by the vertical glue.

```
    basic.setLayout(new BoxLayout(basic, BoxLayout.Y_AXIS));
```

Here we create a basic panel with the vertical BoxLayout.

```
    JPanel bottom = new JPanel();
    bottom.setAlignmentX(1f);
```

```
bottom.setLayout(new BoxLayout(bottom, BoxLayout.X_AXIS));
```

The bottom panel is right aligned. This is done by the setAlignmentX() method. The panel has a horizontal layout.

```
bottom.add(Box.createRigidArea(new Dimension(5, 0)));
```

We put some rigid space between the buttons.

```
basic.add(bottom);
```

Here we put the bottom panel with a horizontal box layout to the vertical basic panel.

```
basic.add(Box.createRigidArea(new Dimension(0, 15)));
```

We also put some space between the bottom panel and the border of the window.

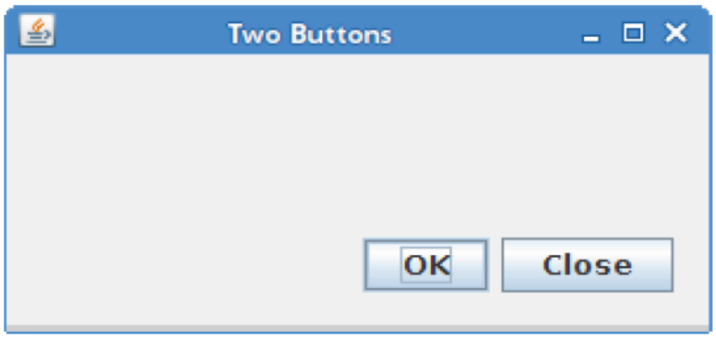


Figure: BoxLayout buttons example

No manager

It is possible to go without a layout manager. Without layout manager, we position components using absolute values.

Example: AbsoluteLayoutEx.java

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
public class AbsoluteLayoutEx extends JFrame {
public AbsoluteLayoutEx() {
initUI();
}
private void initUI() {
setLayout(null);
JButton okBtn = new JButton("OK");
okBtn.setBounds(50, 50, 80, 25);
JButton clsBtn = new JButton("Close");
clsBtn.setBounds(150, 50, 80, 25);
add(okBtn);
add(clsBtn);
setTitle("Absolute positioning");
setSize(300, 250);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
}
public static void main(String[] args) {
SwingUtilities.invokeLater(() -> {
AbsoluteLayoutEx ex = new AbsoluteLayoutEx();
ex.setVisible(true);
});
}
}
```

This simple example shows two buttons.

```
        setLayout(null);
```

We use absolute positioning by providing null to the setLayout() method. (The JFrame component has a default layout manager, the BorderLayout.)

```
        okBtn.setBounds(50, 50, 80, 25);
```

The setBounds() method positions the Ok button. The parameters are the x and y coordinates and the width and height of the component.

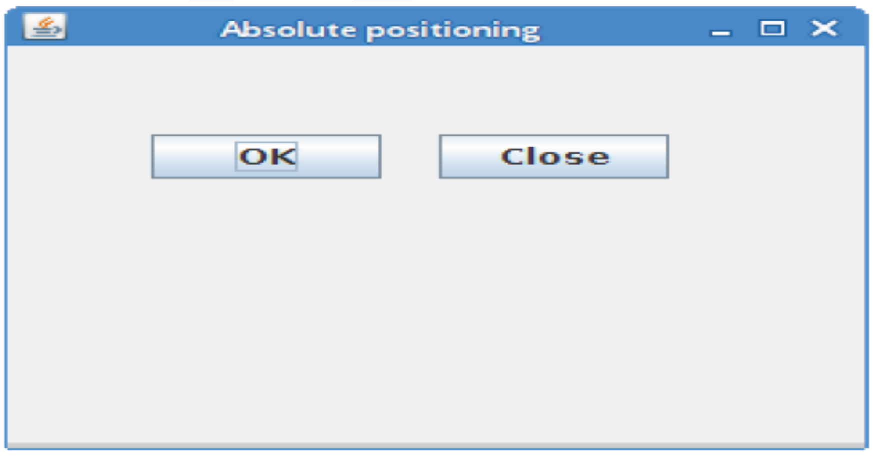


Figure: Absolute layout

SWING COMPONENTS

A component is an independent visual control. Swing Framework contains a large set of components which provide rich functionalities and allow high level of customization. They all are derived from JComponent class. All these components are lightweight components. This class provides some common functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.

A container holds a group of components. It provides a space where a component can be managed and displayed. Containers are of two types,

- 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

JButton class provides functionality of a button. JButton class has three constuctors,

- JButton(Icon ic)
- JButton(String str)
- JButton(String str, Icon ic)

JTextField

JTextField is used for taking input of single line of text. It is most widely used text component. It has three constructors,

- JTextField(int cols)
- JTextField(String str, int cols)

- JTextField(String str)

cols represent the number of columns in text field.

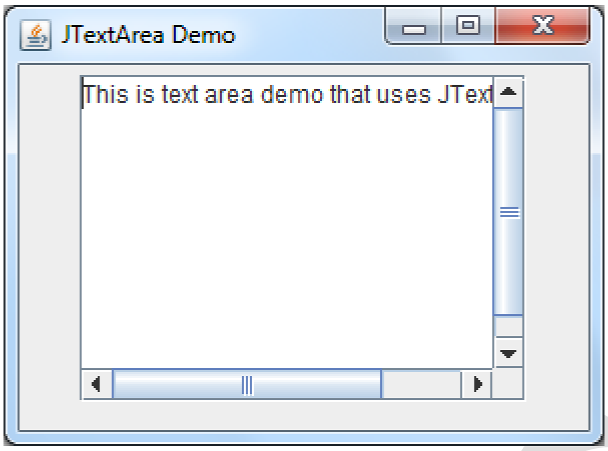
Example using JTextField

```
import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

public class MyTextField extends JFrame

{

 public MyTextField()

 {

  JTextField jtf = new JTextField(20);        //creating JTextField.

  add(jtf);                                   //adding JTextField to frame.

  setLayout(new FlowLayout());

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  setSize(400, 400);

  setVisible(true);

 }

 public static void main(String[] args)

 {

  new MyTextField();

 }

}
```

JTextArea

we create a new text area using the *JTextArea* class.

Here is the screenshot of the demo application:



JButton

```
package jtextareademo;

import javax.swing.*;

import java.awt.*;

public class Main {

    public static void main(String[] args) {

        final JFrame frame = new JFrame("JTextArea Demo");

        JTextArea ta = new JTextArea(10, 20);

         JScrollPane sp = new JScrollPane(ta);

         frame.setLayout(new FlowLayout());

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setSize(300, 220);

        frame.getContentPane().add(sp);
        frame.setVisible(true);
   }}
```

Example using JButton

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class testswing extends JFrame
{
 testswing() {
  JButton bt1 = new JButton("Yes");            //Creating a Yes Button.
  JButton bt2 = new JButton("No");             //Creating a No Button.
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)     //setting close operation.
  setLayout(new FlowLayout());                 //setting layout using FlowLayout object
  setSize(400, 400);                   //setting size of Jframe
  add(bt1);              //adding Yes button to frame.
  add(bt2);              //adding No button to frame.
  setVisible(true); }
 public static void main(String[] args) {
  new testswing(); }}
```

JCheckBox

JCheckBox class is used to create checkboxes in frame. Following is constructor for JCheckBox,

JCheckBox(String str)

Example using JCheckBox

```
import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

public class Test extends JFrame{

 public Test() {

  JCheckBox jcb = new JCheckBox("yes");   //creating JCheckBox.

  add(jcb);                              //adding JCheckBox to frame.

  jcb = new JCheckBox("no");                       //creating JCheckBox.

  add(jcb);                              //adding JCheckBox to frame.

  jcb = new JCheckBox("maybe");                    //creating JCheckBox.

  add(jcb);                              //adding JCheckBox to frame.

  setLayout(new FlowLayout());

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  setSize(400, 400);

  setVisible(true); }

 public static void main(String[] args) {

  new Test(); }}
```

Objects for the above components can be created by invoking its constructor . Let we see an example program

JRadioButton

Radio button is a group of related button in which only one can be selected. JRadioButton class is used to create a radio button in Frames. Following is the constructor for JRadioButton,

JRadioButton(String str)

Example using JRadioButton

```
import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

public class Test extends JFrame

{

 public Test()

 {

  JRadioButton jcb = new JRadioButton("A");          //creating JRadioButton.

  add(jcb);                                          //adding JRadioButton to frame.

  jcb = new JRadioButton("B");                       //creating JRadioButton.

  add(jcb);                                          //adding JRadioButton to frame.

  jcb = new JRadioButton("C");                       //creating JRadioButton.

  add(jcb);                                          //adding JRadioButton to frame.

  jcb = new JRadioButton("none");

  add(jcb);

  setLayout(new FlowLayout());

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  setSize(400, 400);

  setVisible(true);

 }

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

JList

A list is a widget that allows user to choose either a single selection or multiple selections. To create a list widget you use ***JList*** class. The *JList* class itself does not support scrollbar. In order to add scrollbar, you have to use *JScrollPane* class together with JList class. The JScrollPane then manages a scrollbar automatically.

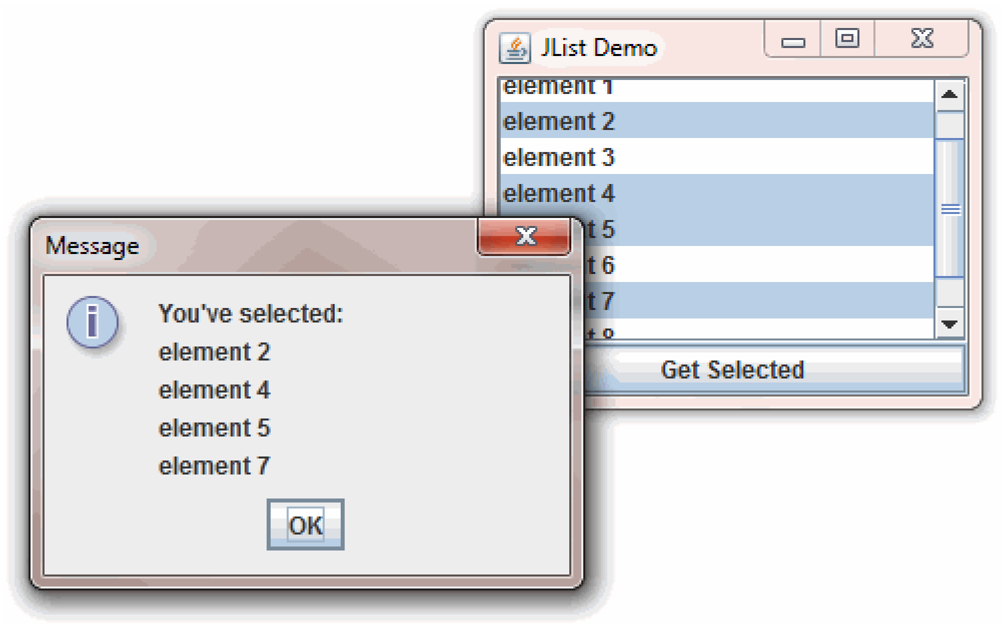
Here are the constructors of JList class:

JList Constructors	Meaning
public JList()	Creates an empty List.
public JList(ListModel listModel)	Creates a list from a given model
public JList(Object[] arr)	Creates a list that displays elements of a given array.
public JList(Vector<?> vector)	Creates a list that displays elements of a given vector.

Example

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

public class Main {

    public static void main(String[] args) {
        final int MAX = 10;
        // initialize list elements
        String[] listElems = new String[MAX];
        for (int i = 0; i < MAX; i++) {
            listElems[i] = "element " + i;
        }

        final JList list = new JList(listElems);
        final JScrollPane pane = new JScrollPane(list);
        final JFrame frame = new JFrame("JList Demo");

        // create a button and add action listener
        final JButton btnGet = new JButton("Get Selected");
        btnGet.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String selectedElem = "";
                int selectedIndices[] = list.getSelectedIndices();
                for (int j = 0; j < selectedIndices.length; j++) {
                    String elem = (String) list.getModel().getElementAt(selectedIndices[j]);
                    selectedElem += "\n" + elem;

                }
                JOptionPane.showMessageDialog(frame,
                        "You've selected:" + selectedElem);
            }// end actionPerformed
        });

        frame.setLayout(new BorderLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(pane, BorderLayout.CENTER);
        frame.getContentPane().add(btnGet, BorderLayout.SOUTH);
        frame.setSize(250, 200);
        frame.setVisible(true);
    }
}
```

Output



You've selected:
element 2
element 4
element 5
element 7

JComboBox

ComboBox is a swing widget that displays a drop down list. A ComboBox gives users options that they can select one and only one item at a time. A ComboBox can be *editable* or *read-only*. To create a ComboBox, you use JComboBox class.

Here are the most common constructor of JComboBox class:

JComboBox Constructors	**Meaning**
public **JComboBox** ()	Creates a JComboBox instance with default data model.
public**JComboBox**(ComboBoxModel model)	Creates a JComboBox instance with elements from a specified ComboBoxModel instance.
public **JComboBox**(Object[] arr)	Creates a JComboBox instance with elements from a given array.
public **JComboBox**(Vector<?> vector)	Creates a JComboBox instance that displays elements from a given vector.

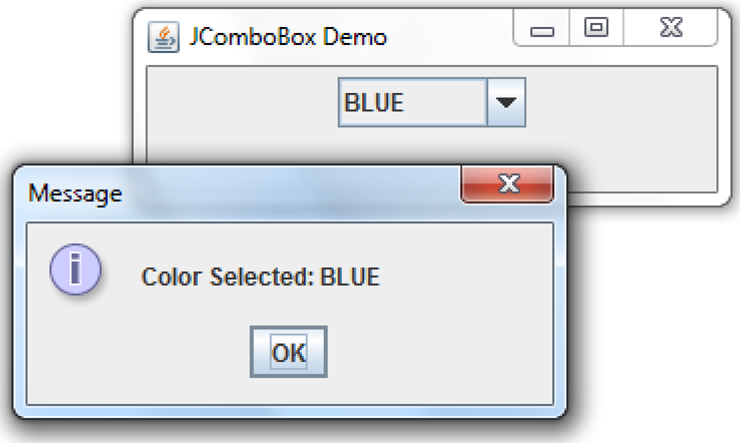
You can create a JComboBox instance from an array or vector. Most of the time, you will use ComboBoxModel to manipulate ComboBox's elements.

To set the ComboBox editable you use the method *setEditable()*. The default editor will be displayed as an input field(JTextField) which accepts user's input.

<u>Example</u>

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

public class Main {

    public static void main(String[] args) {
        final JFrame frame = new JFrame("JComboBox Demo");

        String[] colorList = {"RED",
            "GREEN",
            "BLUE",
            "CYAN",
            "DARK GRAY",
            "MAGENTA",
            "ORANGE",
            "PINK"};
        final JComboBox cbColor = new JComboBox(colorList);
        cbColor.addActionListener(
                new ActionListener() {

                    public void actionPerformed(ActionEvent e) {
                        JOptionPane.showMessageDialog(frame,
                                "Color Selected: " +
                                cbColor.getSelectedItem().toString());
                    }
                });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        Container cont = frame.getContentPane();
        cont.setLayout(new FlowLayout());
        cont.add(cbColor);
        frame.setVisible(true);
    }
}
```

Output



Color Selected: BLUE

SCROLLBARS

A scrollbar consists of a rectangular tab called *slider* or *thumb* located between two *arrow buttons*. Two arrow buttons control the position of the slider by increasing or decreasing a number of units, one unit by default. The area between the slider and the arrow buttons is known as paging area. If user clicks on the paging area, the slider will move one block, normally 10 units. The slider's position of scrollbar can be changed by:

- Dragging the slider up and down or left and right.
- Pushing on either of two arrow buttons.
- Clicking the paging area.

In addition, user can use scroll wheel on computer mouse to control the scrollbar if it is available.

To create a scrollbar in swing, you use JScrollBar class. You can create either vertical or horizontal scrollbar.

Here are the JScrollBar’s constructors.

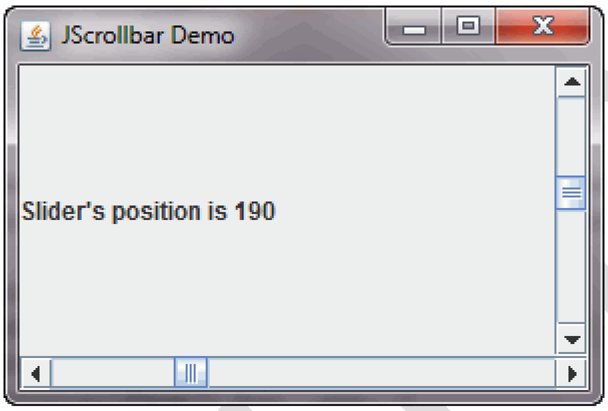
Constructors	Descriptions
JScrollBar()	Creates a vertical scrollbar.
JScrollBar(int orientation)	Creates a scrollbar with a given orientation.
JScrollBar(int orientation, int value, int extent, int min, int max)	Creates a scrollbar with a given orientation and initialize the following scrollbar’s properties: value, extent, minimum, and maximum.

Example

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

public class Main {

  public static void main(String[] args) {
    final JFrame frame = new JFrame("JScrollbar Demo");
    final JLabel label = new JLabel( );

    JScrollBar hbar=new JScrollBar(JScrollBar.HORIZONTAL, 30, 20, 0, 500);
    JScrollBar vbar=new JScrollBar(JScrollBar.VERTICAL, 30, 40, 0, 500);

    class MyAdjustmentListener implements AdjustmentListener {
      public void adjustmentValueChanged(AdjustmentEvent e) {
        label.setText("Slider's position is " + e.getValue());
        frame.repaint();
      }
    }

    hbar.addAdjustmentListener(new MyAdjustmentListener( ));
    vbar.addAdjustmentListener(new MyAdjustmentListener( ));
```

```
        frame.setLayout(new BorderLayout( ));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,200);
        frame.getContentPane().add(label);
        frame.getContentPane().add(hbar, BorderLayout.SOUTH);
        frame.getContentPane().add(vbar, BorderLayout.EAST);
        frame.getContentPane().add(label, BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

Output



```
Slider's position is 190
```

WINDOWS

A Window object is a top-level window with no borders and no menubar. The default layout for a window is BorderLayout. A window must have either a frame, dialog, or another window defined as its owner when it's constructed.

In a multi-screen environment, you can create a Window on a different screen device by constructing the Window with Window(Window, GraphicsConfiguration).

In a virtual device multi-screen environment in which the desktop area could span multiple physical screen devices, the bounds of all configurations are relative to the virtual device coordinate system. The origin of the virtual-coordinate system is at the upper left-hand corner of the primary physical screen. Depending on the location of the primary screen in the virtual device, negative coordinates are possible

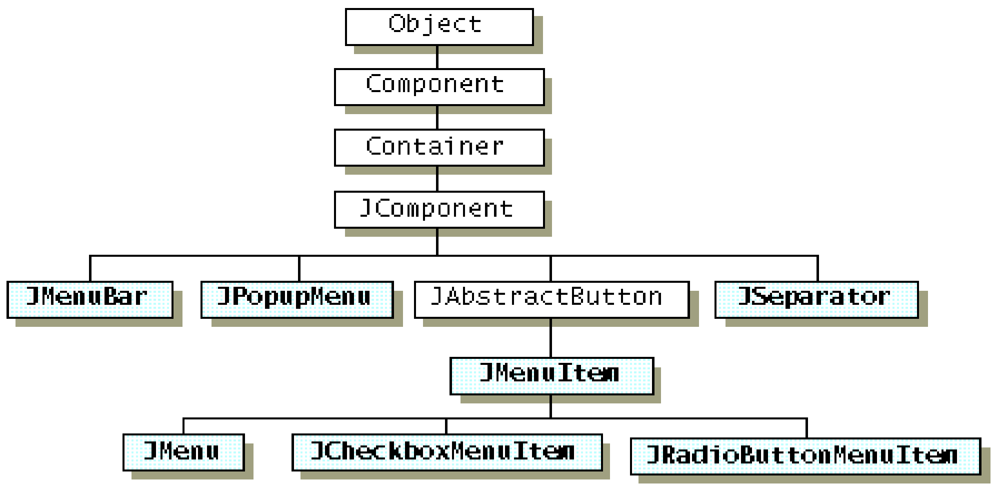
JMenu

Menu components in Swing are subclasses of JComponent so they have all functionality of a normal Swing component.

Here are some remarkable characteristics of the Java Swing menu:

- Menu items can have both icons and text.
- Menu items can be checkboxes or radio buttons.
- Each menu item can have one keyboard accelerator that display after the menu item text.
- Almost standard Swing components can be used as menu items.

Here is the class hierarchy of Swing menu system.



Create Menu Items with JMenuItem

In order to create menu items in Swing, you need to create new instances of *JMenuItem* and set different properties for them. You can create menu item with both text and icon.

Here are the constructors of *JMenuItem:*

Constructors	**Description**
JMenuItem()	Creates a JMenuItem instance without icon or text.
JMenuItem(Icon icon)	Creates a JMenuItem instance with a given icon.
JMenuItem(String text)	Creates a JMenuItem instance with a given text.
JMenuItem(String text, Icon icon)	Creates a JMenuItem instance with a given text and icon

Constructors	Description
JMenuItem(String text, int mnemonic)	Creates a JMenuItem instance with the given text and keyboard mnemonic.
JMenuItem(Action a)	Creates a JMenuItem instance whose properties are taken from the a given Action.

Here is the code snippet to create a menu item:

```
menuItem = new JMenuItem("New Project...",  new ImageIcon("images/newproject.png"));
menuItem.setMnemonic(KeyEvent.VK_P);
menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1,
ActionEvent.ALT_MASK));
menuItem.getAccessibleContext().setAccessibleDescription( "New Project");
```

Create Menu with JMenu

After having menu items, you need to add them to a menu. In order to create a menu you need to use JMenu class. JMenu class represents the menu which can attach to a menu bar or another menu. Menu directly attached to a menu bar is known as top-level menu. If the menu is attached to a menu, it is called sub-menu.

Here are the constructors of JMenu class:

Constructors	Description
JMenu()	Creates an instance of JMenu without text.
JMenu(String s)	Creates an instance of JMenu a given text.
JMenu(String s, boolean tearOffMenu)	Creates an instance of JMenu a given text. and specify the menu as a tear-off menu or not.
JMenu(Action a)	Creates an instance of JMenu whose properties are taken from the specified Action.

Here is the code snippet to create a menu and add a menu item to that menu.

```
//Build the File Menu.
```

```
 menu = new JMenu("File");
 menu.setMnemonic(KeyEvent.VK_F);
 menu.getAccessibleContext().setAccessibleDescription("Dealing with Files");
// create menu item and add it to the menu
 menuItem = new JMenuItem("New File...", new ImageIcon("images/newfile.png"));
 menuItem.setMnemonic(KeyEvent.VK_F);

 menu.add(menuItem);
```

Create menu bar and attach it to a frame

JMenuBar class represents menu bar and can attach it to a frame. One or more menu can be added to a menu bar by using *add()* method. To attach menu bar to a frame you use method *setJMenuBar()* of frame.

Example

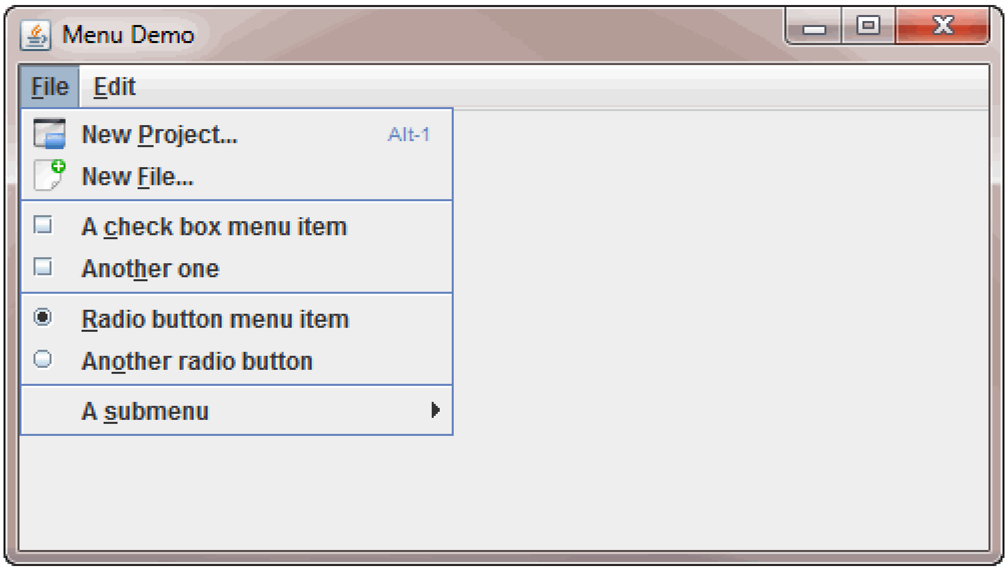
```
import java.awt.event.*;

public class Main {

    public static JMenuBar createMenuBar() {

        JMenuBar menuBar;
        JMenu menu, submenu;
        JMenuItem menuItem;
        JRadioButtonMenuItem rdmi;
        JCheckBoxMenuItem cbmi;

        //Create the menu bar.
        menuBar = new JMenuBar();

        //Build the File Menu.
        menu = new JMenu("File");
        menu.setMnemonic(KeyEvent.VK_F);
        menu.getAccessibleContext().setAccessibleDescription("Dealing with Files");
```

```
menuBar.add(menu);

//a group of JMenuItems
menuItem = new JMenuItem("New Project...",
        new ImageIcon("images/newproject.png"));
menuItem.setMnemonic(KeyEvent.VK_P);
menuItem.setAccelerator(KeyStroke.getKeyStroke(
        KeyEvent.VK_1, ActionEvent.ALT_MASK));
menuItem.getAccessibleContext().setAccessibleDescription(
        "New Project");
menu.add(menuItem);

menuItem = new JMenuItem("New File...",
        new ImageIcon("images/newfile.png"));
menuItem.setMnemonic(KeyEvent.VK_F);
menu.add(menuItem);

//a group of check box menu items
menu.addSeparator();
cbmi = new JCheckBoxMenuItem("A check box menu item");
cbmi.setMnemonic(KeyEvent.VK_C);
menu.add(cbmi);

cbmi = new JCheckBoxMenuItem("Another one");
cbmi.setMnemonic(KeyEvent.VK_H);
menu.add(cbmi);

//a group of radio button menu items
menu.addSeparator();
ButtonGroup group = new ButtonGroup();
rdmi = new JRadioButtonMenuItem("Radio button menu item");
rdmi.setSelected(true);
rdmi.setMnemonic(KeyEvent.VK_R);
group.add(rdmi);
```

```
    menu.add(rdmi);

    rdmi = new JRadioButtonMenuItem("Another radio button");
    rdmi.setMnemonic(KeyEvent.VK_O);
    group.add(rdmi);
    menu.add(rdmi);

    //a submenu
    menu.addSeparator();
    submenu = new JMenu("A submenu");
    submenu.setMnemonic(KeyEvent.VK_S);

    menuItem = new JMenuItem("Menu item in the submenu");
    menuItem.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_2, ActionEvent.ALT_MASK));
    submenu.add(menuItem);

    menuItem = new JMenuItem("Another menu item");
    submenu.add(menuItem);
    menu.add(submenu);

    //Build Edit menu in the menu bar.
    menu = new JMenu("Edit");
    menu.setMnemonic(KeyEvent.VK_E);
    menu.getAccessibleContext().setAccessibleDescription(
            "Edit Menu");
    menuBar.add(menu);
    return menuBar;

  }

  public static void main(String[] args) {
    final JFrame frame = new JFrame("Menu Demo");
    frame.setJMenuBar(createMenuBar());
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(500, 300);
        frame.setVisible(true);
    }
}
```

Output



JDialog

JDialog is a part Java swing package. The main purpose of the dialog is to add components to it. JDialog can be customized according to user need .

Constructor of the class are:

1. **JDialog()** : creates an empty dialog without any title or any specified owner
2. **JDialog(Frame o)** :creates an empty dialog with a specified frame as its owner
3. **JDialog(Frame o, String s)** : creates an empty dialog with a specified frame as its owner
   and a specified title
4. **JDialog(Window o)** : creates an empty dialog with a specified window as its owner
5. **JDialog(Window o, String t)** : creates an empty dialog with a specified window as its owner and specified title.
6. **JDialog(Dialog o)** :creates an empty dialog with a specified dialog as its owner

7. **JDialog(Dialog o, String s)** : creates an empty dialog with a specified dialog as its owner and specified title.

Commonly used methods

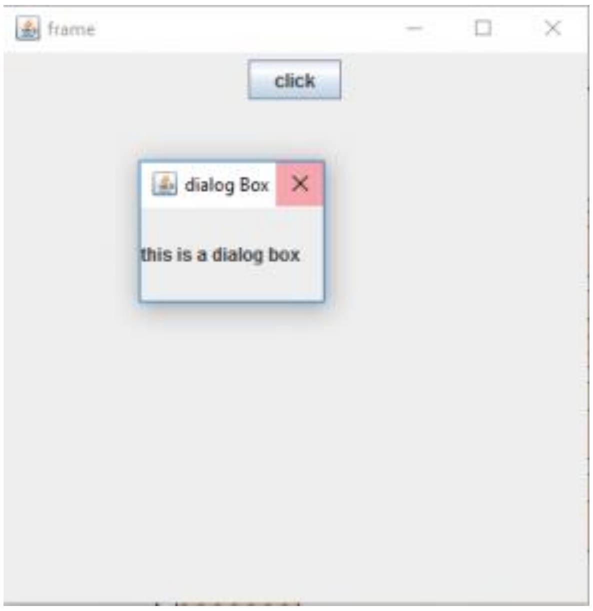
1. **setLayout(LayoutManager m)** : sets the layout of the dialog to specified layout manager
2. **setJMenuBar(JMenuBar m)** : sets the menubar of the dialog to specified menubar
3. **add(Component c)**: adds component to the dialog
4. **isVisible(boolean b)**: sets the visibility of the dialog, if value of the boolean is true then visible else invisible
5. **update(Graphics g)** : calls the paint(g) function
6. **remove(Component c)** : removes the component c
7. **getGraphics()** : returns the graphics context of the component.
8. **getLayeredPane()** : returns the layered pane for the dialog
9. **setContentPane(Container c)** :sets the content pane for the dialog
10. **setLayeredPane(JLayeredPane l)** : set the layered pane for the dialog

Example1

```
// java Program to create a simple JDialog
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class solve extends JFrame implements ActionListener {

    // frame
    static JFrame f;

    // main class
    public static void main(String[] args)
    {
        // create a new frame
        f = new JFrame("frame");

        // create a object
        solve s = new solve();

        // create a panel
        JPanel p = new JPanel();

        JButton b = new JButton("click");

        // add actionlistener to button
        b.addActionListener(s);

        // add button to panel
        p.add(b);

        f.add(p);

        // set the size of frame
        f.setSize(400, 400);

        f.show();
    }
    public void actionPerformed(ActionEvent e)
    {
        String s = e.getActionCommand();
        if (s.equals("click")) {
            // create a dialog Box
            JDialog d = new JDialog(f, "dialog Box");

            // create a label
            JLabel l = new JLabel("this is a dialog box");

            d.add(l);

            // setsize of dialog
            d.setSize(100, 100);
```

```
            // set visibility of dialog
            d.setVisible(true);
        }
    }
}
```

Output



Example2

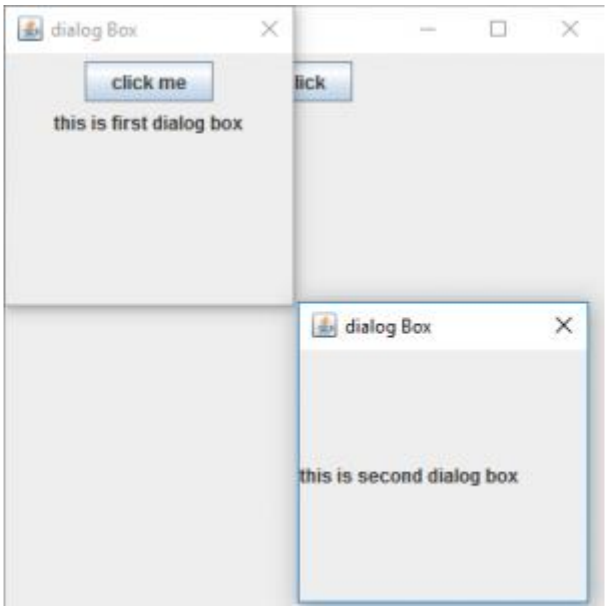
// java Program to create a dialog within a dialog

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class solve extends JFrame implements ActionListener {

    // frame
    static JFrame f;

    // dialog
    static JDialog d, d1;
```

```
// main class
public static void main(String[] args)
{
    // create a new frame
    f = new JFrame("frame");

    // create a object
    solve s = new solve();

    // create a panel
    JPanel p = new JPanel();

    JButton b = new JButton("click");

    // add actionlistener to button
    b.addActionListener(s);

    // add button to panel
    p.add(b);

    f.add(p);

    // set the size of frame
    f.setSize(400, 400);

    f.show();
}
public void actionPerformed(ActionEvent e)
{
    String s = e.getActionCommand();
    if (s.equals("click")) {
        // create a dialog Box
        d = new JDialog(f, "dialog Box");
        // create a label
        JLabel l = new JLabel("this is first dialog box");

        // create a button
        JButton b = new JButton("click me");

        // add Action Listener
        b.addActionListener(this);

        // create a panel
        JPanel p = new JPanel();

        p.add(b);
        p.add(l);

        // add panel to dialog
        d.add(p);

        // setsize of dialog
        d.setSize(200, 200);

        // set visibility of dialog
        d.setVisible(true);
    }
    else { // create a dialog Box
        d1 = new JDialog(d, "dialog Box");

        // create a label
        JLabel l = new JLabel("this is second dialog box");

        d1.add(l);

        // setsize of dialog
        d1.setSize(200, 200);
```

```
            // set loaction of dialog
            d1.setLocation(200, 200);

            // set visibility of dialog
            d1.setVisible(true);
        }
    }
}
```

Output



this is first dialog box

this is second dialog box

<u>JTable</u>

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.

Constructors in JTable:

1. **JTable():** A table is created with empty cells.
2. **JTable(int rows, int cols):** Creates a table of size rows * cols.
3. **JTable(Object[][] data, Object []Column):** A table is created with the specified name where []Column defines the column names.

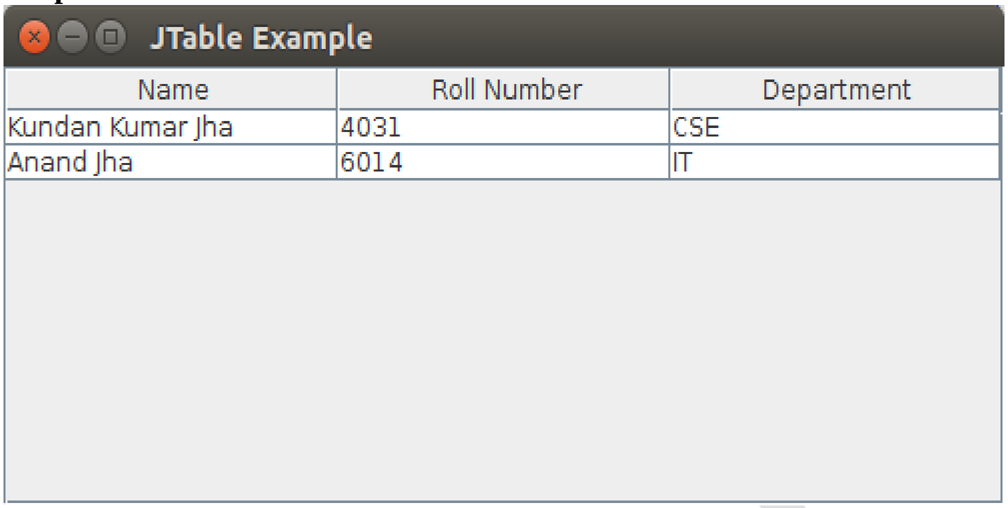
Functions in JTable:

1. **addColumn(TableColumn []column) :** adds a column at the end of the JTable.
2. **clearSelection() :** Selects all the selected rows and columns.
3. **editCellAt(int row, int col) :** edits the intersecting cell of the column number col and row number row programmatically, if the given indices are valid and the corresponding cell is editable.
4. **setValueAt(Object value, int row, int col) :** Sets the cell value as ‘value’ for the position row, col in the JTable.

Example

```
// Packages to import
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class JTableExamples {
    // frame
    JFrame f;
    // Table
    JTable j;

    // Constructor
    JTableExamples()
    {
        // Frame initiallization
        f = new JFrame();

        // Frame Title
        f.setTitle("JTable Example");

        // 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
        j = new JTable(data, columnNames);
        j.setBounds(30, 40, 200, 300);

        // adding it to JScrollPane
        JScrollPane sp = new JScrollPane(j);
        f.add(sp);
        // Frame Size
        f.setSize(500, 200);
        // Frame Visible = true
        f.setVisible(true);
    }

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

Output:


JTable Example

Name	Roll Number	Department
Kundan Kumar Jha	4031	CSE
Anand Jha	6014	IT

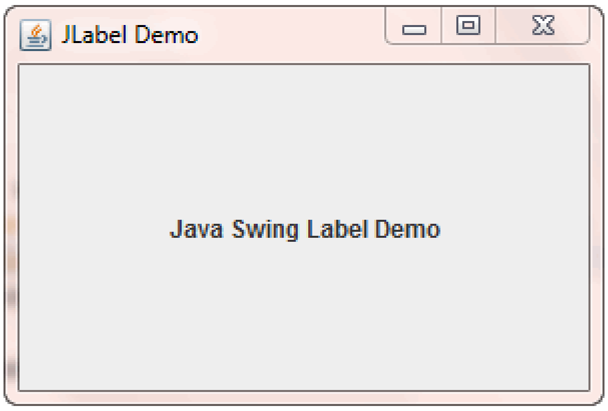
JLabel

The label is the simplest component in the Swing toolkit. The label can contain text, icon or both. To create a simple and non-interactive label, you use ***JLabel*** class.

Example

```
import javax.swing.JLabel;
import javax.swing.JFrame;

public class Main {
    public static void main(String[] args) {
        JLabel label = new JLabel("Java Swing Label Demo",JLabel.CENTER);
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400,300);
        frame.getContentPane().add(label);
        frame.setVisible(true);
    }
}
```

Output


JLabel Demo
Java Swing Label Demo