APPLETS:

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **public void init()** – This method is the first method to be called by the browser and it is executed only once. So, the programmer should use this method to initialize any variables, creating components and creating threads, etc. It is called after the param tags inside the applet tag have been processed.
- **public void start()** − This method is called after init() method and each time the applet is revisited by the user. For example, the user minimized and the webpage that contains the applet is moved to another page then this method's execution is stopped. Any calculations and processing of data should be done in this method and result is displayed.
- **public void stop()** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **public void destroy()** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.

Creation of applet:

To create an applet that displays "Hello, World" in the applet frame. We can take the help of paint() method of component class of *java.awt* package.

- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java –

```
import java.applet.*;
import java.awt.*;
public class HelloWorldApplet extends Applet {
   public void paint (Graphics g) {
      g.drawString ("Hello, World", 25, 50);
   }
}
```

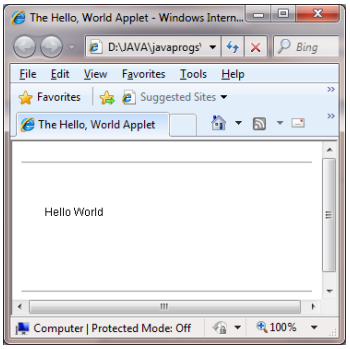
After creation of applet save the program with class name. Because the class is public. Then compile the above program "HelloWorldApplet.class" file created.

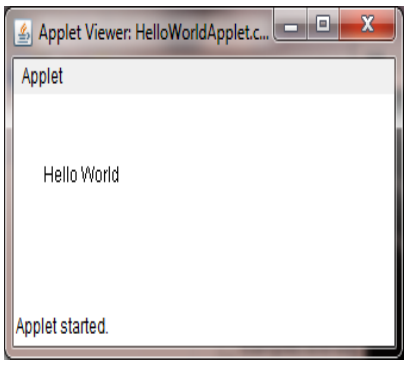
Invoking the applet in web browser:

```
<html>
<head>      <title>Hello World </title> </head>
<body>
<applet code="HelloWorldApplet.class" width="500"  height="400"
></applet>
</body>
</html>
```

Save the above code with "hello.html". This HTML page Contains the applet which can be opened in the browser, or an "appletviewer" can be used to test the applet.

```
> appletviewer hello.html
```

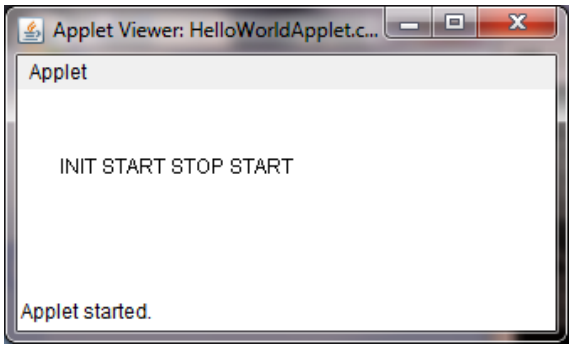




Example 2: Write a java applet program describe about life cycle of applet.

```
import java.applet.*;
import java.awt.*;
public class HelloWorldApplet extends Applet {
      String msg="";
      public void init()
      {
            msg+="INIT ";
      }
      public void start()
      {
            msg+="START ";
```

```
    }
    public void stop()
    {
        msg+="STOP ";
    }
    public void destroy()
    {
        msg+="DESTROY ";
    }
  public void paint (Graphics g) {
      System.out.println(msg);
    g.drawString (msg, 25, 50);
  }
}
```



Uses of applet:

- ✓ Applets are used on internet for creating dynamic web pages. There are two types of web pages: Static and Dynamic. Static web pages provide some information to the user but the user cannot interact with the web page other than viewing the information. Dynamic web pages interact with the user at runtime. For example, a student can type his hall ticket number in a text field and click retrieve button to get back his results from the university server.
- ✓ Another use of applets is for creating animation and games where images can be displayed or, moved giving a visual impression that they are alive.

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser

- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

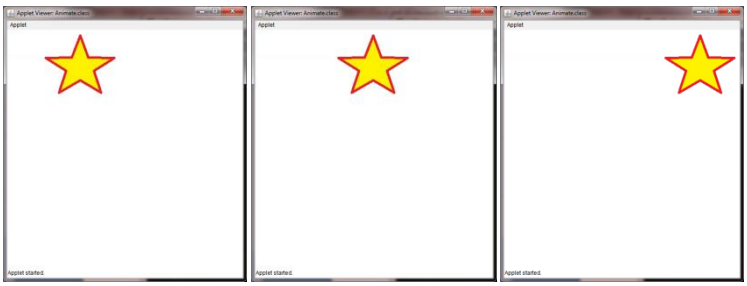
Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

Note:

- The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with an </applet> tag.
- If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.
- Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

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

public class Animate extends Applet {
   public void paint (Graphics g) {
      Image img=getImage(getDocumentBase(),"hello.jpg");
      for(int i=0;i<800;i++)
      {
            g.drawImage(img,i,0,null);
            try{
                  Thread.sleep(20);
            }catch(Exception e){}
      }
   }
}
```



repaint():-

The **repaint()** method is sent to a Component when it needs to be repainted. This happens when a window is moved, or resized, or unhidden. It also happens when a webpage contains an image and the pixels of the image are arriving slowly down the wire.

update(Graphics):-

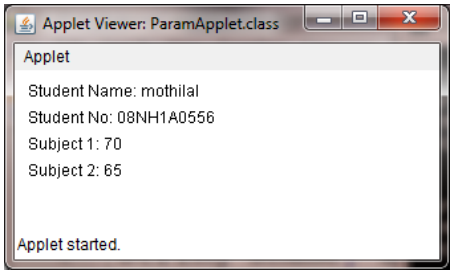
The repaint () method causes then AWT runtime system to execute the update () method of the Component class which clears the window with the background color of the applet and then calls the paint () method. But sometimes erasing the background is undesirable. For example: If user wants to display the x and y coordinates of each location where the mouse is clicked instead of only the currently clicked location then it is not possible using the default version of the update () method. Each time the user clicks in the applets's window, the coordinates of the previously clicked location are erased and only the coordinates of the currently clicked location are displayed. To overcome this problem, it is necessary to override the update () method to invoke paint () directly, thus avoiding the erasure of the component's background.

Passing Parameters to Applets:

We can pass the parameters to the applet. When we create we use those parameters by using getParameter() method. This method returns the parameters in the form of String object.

Ex: Write an applet to take student name, student no, and two subject's marks.

```
import java.applet.*;
import java.awt.*;
public class ParamApplet extends Applet {
String sname, sno, s1, s2;
public void start(){
sname=getParameter("stdname");
sno=getParameter("stdno");
s1=getParameter("sub1");
s2=getParameter("sub2");
}
   public void paint (Graphics g) {
      g.drawString("Student Name: "+sname,10,20);
      g.drawString("Student No: "+sno,10,40);
      g.drawString("Subject 1: "+s1,10,60);
      g.drawString("Subject 2: "+s2,10,80);

   }
}
```




```
         <applet code = "ParamApplet.class" width = "320" height = "120">
         <param name="stdname" value="mothilal">
         <param name="stdno" value="08NH1A0556">
         <param name="sub1" value="70">
         <param name="sub2" value="65">
         </applet>
```

What is an Event?

Change in the state of an object is known as **Event**, i.e., event describes the change in the state of the source. Events are generated as a result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from the list, and scrolling the page are the activities that cause an event to occur.

Types of Event:

The events can be broadly classified into two categories −

- **Foreground Events** − these events require direct interaction of the user. They are generated as consequences of a person interacting with the graphical components in the Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page, etc.
- **Background Events** − These events require the interaction of the end user. Operating system interrupts, hardware or software failure, timer expiration, and operation completion are some examples of background events.

What is Event Handling?

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

Java uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model has the following key participants.

Source − the source is an object on which the event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides us with classes for the source object.

Listener − It is also known as event handler. The listener is responsible for generating a response to an event. From the point of view of Java implementation, the listener is also an object. The listener waits till it receives an event. Once the event is received, the listener processes the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to a separate piece of code.

In this model, the listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listeners who want to receive them.

Steps Involved in Event Handling

Step 1 − The user clicks the button and the event is generated.

Step 2 − The object of concerned event class is created automatically and information about the source and the event get populated within the same object.

Step 3 − Event object is forwarded to the method of the registered listener class.

Step 4 − The method is gets executed and returns.

Points to Remember About the Listener

In order to design a listener class, you have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods, which must be implemented by the listener class.

If you do not implement any of the predefined interfaces, then your class cannot act as a listener class for a source object.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represent an event method. In response to an event, java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener to listen ot its events, the source must register itself to the listener.

Event and Listener (Java Event Handling):

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

Java Event classes and Listener interfaces:

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

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

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

- ***Button***
  - *public void addActionListener(ActionListener a){}*
- ***MenuItem***
  - *public void addActionListener(ActionListener a){}*
- ***TextField***
  - *public void addActionListener(ActionListener a){}*
  - *public void addTextListener(TextListener a){}*
- ***TextArea***
  - *public void addTextListener(TextListener a){}*

- ***Checkbox***
  - *public void addItemListener(ItemListener a){}*
- ***Choice***
  - *public void addItemListener(ItemListener a){}*
- ***List***
  - *public void addActionListener(ActionListener a){}*
  - *public void addItemListener(ItemListener a){}*

Delegation Event Model:

- The Event Delegation Model defines a consistent mechanism to generate and process the events.
- The process is: a source generates an event and sends it to one or more listeners.
- In this scheme, the listener simply waits until it receives an event.
- Once an event is received, the listener processes the event and then returns.

Mouse Event Class:

The MouseEvent class defines 8 types of mouse events. It defines the following integer constants that can be used to identify them:

1. MOUSE_CLICKED : The user clicked the mouse
2. MOUSE_DRAGGED : The user dragged the mouse
3. MOUSE_ENTERED : The mouse entered a component
4. MOUSE_EXITED : The mouse exited from a component
5. MOUSE_MOVED : The mouse moved
6. MOUSE_PRESSED : The mouse was pressed
7. MOUSE_RELEASED : The mouse was released
8. MOUSE_WHEEL : The mouse wheel was moved

Methods of MouseEvent Class:

1. int getX(): returns the X coordinates of the mouse when an event occurred
2. int getY(): returns the Y coordinates of the mouse when an event occurred.
3. Point getPoint(): returns the coordinates of the mouse.

MouseListener Interface:

This interface defines five methods:

1. void mouseClicked(MouseEvent me): it invokes if the mouse is pressed and released at the same point
2. void mouseEntered(MouseEvent me): it invokes when the mouse enters a component.
3. void mouseExited(MouseEvent me): it invokes when the mouse leaves the component.
4. void mousePressed(MouseEvent me): it invokes when the mouse is pressed.
5. void mouseReleased(MouseEvent me): it invokes when the mouse is released.

MouseMotionListener Interface:

This interface defines two methods:

1. void mouseDragged(MouseEvent me): it invokes when multiple times as the mouse is dragged.
2. void mouseMoves(MovesEvent me): it invokes when multiple times as the mouse is moved

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="Mouse.class" width=500 height=500> </applet> */
public class Mouse extends Applet implements MouseListener,
MouseMotionListener
{
	int X = 0, Y = 20;
	String msg = "MouseEvents";
	public void init( )
	{
		addMouseListener(this);
		addMouseMotionListener(this);
		setBackground(Color.black);
		setForeground(Color.red);
	}
	public void mouseEntered(MouseEvent m)
	{
		setBackground(Color.magenta);
		showStatus("Mouse Entered");
		repaint( );
	}
	public void mouseExited(MouseEvent m)
	{
		setBackground(Color.black);
		showStatus("Mouse Exited");
		repaint( );
	}
	public void mousePressed(MouseEvent m)
	{
		X = 10;
		Y = 20;
		msg = "GEC";
		setBackground(Color.green);
		repaint( );
	}
	public void mouseReleased(MouseEvent m)
	{
		X = 10;
		Y = 20;
		msg = "Engineering";
		setBackground(Color.blue);
		repaint( );
	}
	public void mouseMoved(MouseEvent m)
	{
		X = m.getX( );
		Y = m.getY( );
		msg = "College";
		setBackground(Color.white);
		showStatus("Mouse Moved");
		repaint( );
```

```
	}
	public void mouseDragged(MouseEvent m)
	{
		msg = "CSE";
		setBackground(Color.yellow);
		showStatus("MouseMoved" + m.getX( ) + " " + m.getY( ));
		repaint( );
	}
	public void mouseClicked(MouseEvent m)
	{
		msg = "Students";
		setBackground(Color.pink);
		showStatus("Mouse Clicked");
		repaint( );
	}
	public void paint(Graphics g)
	{
		g.drawString(msg, X, Y);
	}
}
```

KeyEvent Class:

A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants:

1. KEY_PRESSED
2. KEY_RELEASED
3. KEY_TYPED

The first two events are generated when any key is pressed or released. The last event occurs when a character is generated.

Methods of KeyEvent Class:

- char getKeyChar() : returns the character that was enterd
- int getKeyCode() : returns the character code.

KeyListener Interface:

The interface defines three methods:

1. void keyPressed(KeyEvent ke): it invokes when a key is pressed.
2. void keyReleased(KeyEvent ke): it invokes when a key is released.
3. void keyTyped(KeyEvent ke): it invokes when a key is typed.

Example:

```
/* <applet code="Key.class" width=4000 height=4000> </applet> */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class Key extends Applet implements KeyListener
{
	String msg = "";
	public void init( )
	{
		setBackground(Color.green);
```

```
            addKeyListener(this);
      }
      public void keyPressed(KeyEvent k)
      {
            showStatus("Key pressed");
            if(k.getKeyCode( ) == KeyEvent.VK_ENTER)
                  showStatus("Enter key is pressed:");
            repaint( );
      }
      public void keyReleased(KeyEvent k)
      {
            showStatus("Key Up or Key Released");
      }
      public void keyTyped(KeyEvent k)
      {
            msg += k.getKeyChar( );
            repaint( );
      }
      public void paint(Graphics g)
      {
            g.setFont(new Font("Arial", Font.BOLD, 30));
            g.drawString("Key Typed is:" + msg, 20, 300);
      }
}
```

Java event handling by implementing ActionListener

```
      import java.awt.*;
      import java.awt.event.*;
      class AEvent extends Frame implements ActionListener
      {
            TextField tf;
            AEvent()
            {
                  //create components
                  tf=new TextField();
                  tf.setBounds(60,50,170,20);
                  Button b=new Button("click me");
                  b.setBounds(100,120,80,30);

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

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

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

Adapter classes:

- An adapter class provides an empty implementation of all methods in an event listener interface.
- Adapter classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface.
- For this, we can define a new class to act as an event listener by extending one of the adapter classes and the MouseMotionAdapter class has two methods, mouseDragged( ) and mouseMoved( ).
- If we are interested in only mouse drag events, then we extend mouseMotionAdapter and implement mouseDragged( ). The empty implementation of mouseMoved( ) would handle the mouse motion events.
- Adapter classes are provided by java.awt.event package. Some of the Adapter classes are

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

Example program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo.class" width=100 height=100>
</applet>
*/
public class AdapterDemo extends Applet
{
      public void init( )
      {
            addMouseListener(new my(this));
```

```
            addMouseMotionListener(new mtadd(this));
      }
}
class My extends MouseAdapter
{
      AdapterDemo a;
      public My(AdapterDemo p) {
            This.a = p;
      }
      public void mouseClicked(MouseEvent me)
      {
            a.showStatus("Mouse Clicked");
      }
}
class MyAdd extends MouseMotionAdapter
{
      AdapterDemo a;
      public MyAdd(AdapterDemo p)
      {
            this.p;
      }
      public void mouseDragged(MouseEvent me)
      {
            a.showStatus("Mouse Dragged");
      }
}
```

Java AWT:

Introduction to AWT:

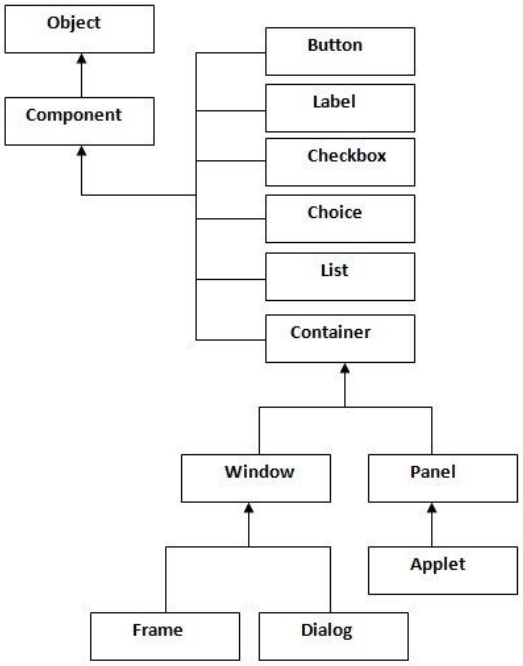
- Java AWT (Abstract Window Toolkit) is an API to develop GUI or window based application in java.
- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- AWT is heavyweight i.e. its components uses the resources of system.
- The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Why AWT is platform independent?

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

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

Java AWT Hierarchy



Components

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

Container

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

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

Types of containers:

There are four types of containers in Java AWT:

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

Window

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

Panel

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

Frame

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

Methods of Component Class

Method	Description
public void add(Component c)	Inserts a component on this component.
public void setSize(int width,int height)	Sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	Defines the layout manager for the component.
public void setVisible(boolean status)	Changes the visibility of the component, by default false.

User Interface Components

- The java.awt package provides an integrated set of classes to manage user interface-components.
- The simplest form of Java AWT component is the basic User Interface Component.
- We can create and add these to your applet as it is an AWT container.
- We can put other AWT components, such as UI components or other containers, in it.
- The following table gives a list of all the Controls in Java AWT and their respective functions.

In java.awt package, the following components are available:

1. **Label:** This component of java AWT creates a multi-line descriptive string that is shown on the graphical user interface.
2. **Button:** This is used to create a button on the user interface with a specified label. We can design code to execute some logic on the click event of a button using listeners.
3. **Text Field:** This component of java AWT creates a text box of a single line to enter text data.
4. **Choice:** This AWT component represents a pop-up menu having multiple choices. The option which the user selects is displayed on top of the menu.
5. **Scroll Bar:** This is used for providing horizontal or vertical scrolling feature on the GUI.
6. **List:** This component can hold a list of text items. This component allows a user to choose one or more options from all available options in the list.
7. **Checkbox:** This component is used to create a checkbox of GUI whose state can be either checked or unchecked.

1. Label:

- A label is an object of type Label, and it contains a string, which it displays.
- Labels are passive controls that do not support any interaction with the user.

AWT Label Fields

The java.awt.Component class has following fields:

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

Constructors

Sr. no.	Constructor	Description
1.	`Label()`	It constructs an empty label.
2.	`Label(String text)`	It constructs a label with the given string (left justified by default).
3.	`Label(String text, int alignement)`	It constructs a label with the specified string and the specified alignment.

Methods Specified

Sr. no.	Method name	Description
1.	`void setText(String text)`	It sets the texts for label with the specified text.
2.	`void setAlignment(int alignment)`	It sets the alignment for label with the specified alignment.
3.	`String getText()`	It gets the text of the label
4.	`int getAlignment()`	It gets the current alignment of the label.
5.	`void addNotify()`	It creates the peer for the label.
6.	`AccessibleContext getAccessibleContext()`	It gets the Accessible Context associated with the label.
7.	`protected String paramString()`	It returns the string the state of the label.

Example:

```
import java.awt.*;
public class LabelExample {
public static void main(String args[]){
    Frame f = new Frame ("Label example");
    Label l1, l2;

    l1 = new Label ("First Label.");
    l2 = new Label ("Second Label.");

    l1.setBounds(50, 100, 100, 30);
    l2.setBounds(50, 150, 100, 30);

    f.add(l1);
    f.add(l2);

    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

2. Button:

Constructors

Following table shows the types of Button class constructors

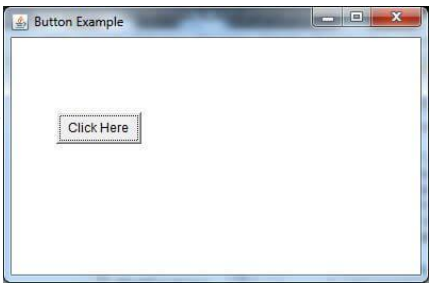
Sr. no.	Constructor	Description
1.	*Button( )*	It constructs a new button with an empty string i.e. it has no label.
2.	*Button (String text)*	It constructs a new button with given string as its label.

Methods

Sr. no.	Method	Description
1.	*void setText (String text)*	It sets the string message on the button
2.	*String getText()*	It fetches the String message on the button.
3.	*void setLabel (String label)*	It sets the label of button with the specified string.
4.	*String getLabel()*	It fetches the label of the button.
5.	*void addNotify()*	It creates the peer of the button.
6.	*AccessibleContext getAccessibleContext()*	It fetched the accessible context associated with the button.
7.	*void addActionListener(ActionListener l)*	It adds the specified action listener to get the action events from the button.
8.	*String getActionCommand()*	It returns the command name of the action event fired by the button.
9.	*ActionListener[ ] getActionListeners()*	It returns an array of all the action listeners registered on the button.
10.	*T[ ] getListeners(Class listenerType)*	It returns an array of all the objects currently registered as FooListeners upon this Button.
11.	*protected String paramString()*	It returns the string which represents the state of button.
12.	*protected void processActionEvent (ActionEvent e)*	It process the action events on the button by dispatching them to a registered ActionListener object.
13.	*protected void processEvent (AWTEvent e)*	It process the events on the button
14.	*void removeActionListener (ActionListener l)*	It removes the specified action listener so that it no longer receives action events from the button.
15.	*void setActionCommand(String command)*	It sets the command name for the action event given by the button.

Example:

```
import java.awt.*;
public class ButtonExample {
public static void main (String[] args) {

    // create instance of frame with the label
    Frame f = new Frame("Button Example");

    // create instance of button with label
    Button b = new Button("Click Here");

    // set the position for the button in frame
    b.setBounds(50,100,80,30);

    // add button to the frame
    f.add(b);
    // set size, layout and visibility of frame
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```



3. TextField:

Constructors

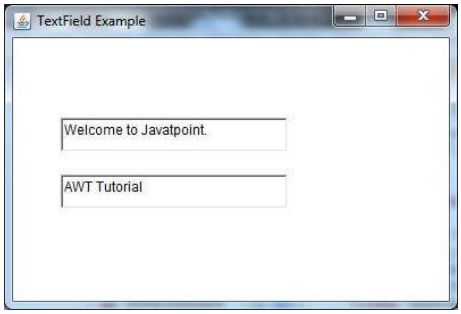
Sr. no.	Constructor	Description
1.	`TextField()`	It constructs a new text field component.
2.	`TextField(String text)`	It constructs a new text field initialized with the given string text to be displayed.
3.	`TextField(int columns)`	It constructs a new textfield (empty) with given number of columns.
4.	`TextField(String text, int columns)`	It constructs a new text field with the given text and given number of columns (width).

Methods

Sr. no.	Method name	Description
1.	`void addNotify()`	It creates the peer of text field.
2.	`boolean echoCharIsSet()`	It tells whether text field has character set for echoing or not.
3.	`void addActionListener(ActionListener l)`	It adds the specified action listener to receive action events from the text field.
4.	`ActionListener[] getActionListeners()`	It returns array of all action listeners registered on text field.
5.	`AccessibleContext getAccessibleContext()`	It fetches the accessible context related to the text field.

6.	*int getColumns()*	It fetches the number of columns in text field.
7.	*char getEchoChar()*	It fetches the character that is used for echoing.
8.	*protected String paramString()*	It returns a string representing state of the text field.
9.	*protected void processActionEvent(ActionEvent e)*	It processes action events occurring in the text field by dispatching them to a registered ActionListener object.
10.	*protected void processEvent(AWTEvent e)*	It processes the event on text field.
11.	*void removeActionListener(ActionListener l)*	It removes specified action listener so that it doesn't receive action events anymore.
12.	*void setColumns(int columns)*	It sets the number of columns in text field.
13.	*void setEchoChar(char c)*	It sets the echo character for text field.
14.	*void setText(String t)*	It sets the text presented by this text component to the specified text.

Example:




```
import java.awt.*;
public class TextFieldExample1 {
    // main method
    public static void main(String args[]) {
    // creating a frame
    Frame f = new Frame("TextField Example");

    // creating objects of textfield
    TextField t1, t2;
    // instantiating the textfield objects
    // setting the location of those objects in the frame
    t1 = new TextField("Welcome to Javatpoint.");
    t1.setBounds(50, 100, 200, 30);
    t2 = new TextField("AWT Tutorial");
    t2.setBounds(50, 150, 200, 30);
    // adding the components to frame
    f.add(t1);
    f.add(t2);
    // setting size, layout and visibility of frame
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

4. Choice:

Constructor

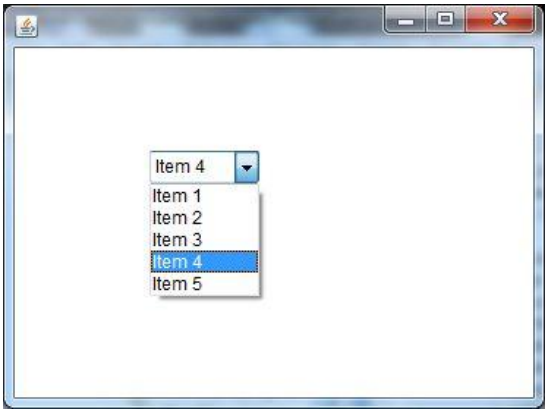
Sr. no.	Constructor	Description
1.	*Choice()*	It constructs a new choice menu.

Methods

Sr. no.	Method name	Description
1.	*void add(String item)*	It adds an item to the choice menu.
2.	*void addItemListener(ItemListener l)*	It adds the item listener that receives item events from the choice menu.
3.	*void addNotify()*	It creates the peer of choice.
4.	*AccessibleContext getAccessibleContext()*	It gets the accessbile context related to the choice.
5.	*String getItem(int index)*	It gets the item (string) at the given index position in the choice menu.
6.	*int getItemCount()*	It returns the number of items of the choice menu.
7.	*ItemListener[] getItemListeners()*	It returns an array of all item listeners registered on choice.
8.	*T[] getListeners(Class listenerType)*	Returns an array of all the objects currently registered as FooListeners upon this Choice.
9.	*int getSelectedIndex()*	Returns the index of the currently selected item.
10.	*String getSelectedItem()*	Gets a representation of the current choice as a string.
11.	*Object[] getSelectedObjects()*	Returns an array (length 1) containing the currently selected item.
12.	*void insert(String item, int index)*	Inserts the item into this choice at the specified position.
13.	*void remove(int position)*	It removes an item from the choice menu at the given index position.
14.	*void remove(String item)*	It removes the first occurrence of the item from choice menu.
15.	*void removeAll()*	It removes all the items from the choice menu.
16.	*void removeItemListener (ItemListener l)*	It removes the mentioned item listener. Thus is doesn't receive item events from the choice menu anymore.
17.	*void select(int pos)*	It changes / sets the selected item in the choice menu to the item at given index position.
18.	*void select(String str)*	It changes / sets the selected item in the choice menu to the item whose string value is equal to string specified in the argument.

Example:

```
// importing awt class
import java.awt.*;
public class ChoiceExample1 {

         // class constructor
        ChoiceExample1() {

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

        // creating a choice component
        Choice c = new Choice();

        // setting the bounds of choice menu
        c.setBounds(100, 100, 75, 75);

        // adding items to the choice menu
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");

        // adding choice menu to frame
        f.add(c);

        // setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
     }
// main method
public static void main(String args[])
{
   new ChoiceExample1();
}
}
```



5. Scroll Bar:

Constructors

Sr. no.	Constructor	Description
1	`Scrollbar()`	Constructs a new vertical scroll bar.
2	`Scrollbar(int orientation)`	Constructs a new scroll bar with the specified orientation.
3	`Scrollbar(int orientation, int value, int visible, int minimum, int maximum)`	Constructs a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum values.

Where the parameters,

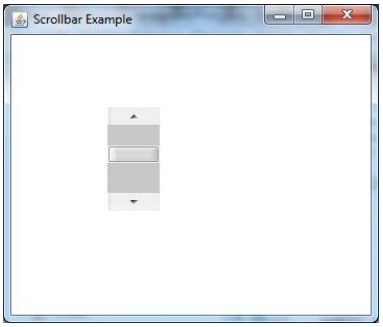
- **orientation**: specify whether the scrollbar will be horizontal or vertical.
- **Value**: specify the starting position of the knob of Scrollbar on its track.
- **Minimum**: specify the minimum width of track on which scrollbar is moving.
- **Maximum**: specify the maximum width of track on which scrollbar is moving.

Methods

Sr. no.	Method name	Description
1.	*void addAdjustmentListener (AdjustmentListener l)*	It adds the given adjustment listener to receive instances of AdjustmentEvent from the scroll bar.
2.	*void addNotify()*	It creates the peer of scroll bar.
3.	*int getBlockIncrement()*	It gets the block increment of the scroll bar.
4.	*int getMaximum()*	It gets the maximum value of the scroll bar.
5.	*int getMinimum()*	It gets the minimum value of the scroll bar.
6.	*int getOrientation()*	It returns the orientation of scroll bar.
7.	*int getUnitIncrement()*	It fetches the unit increment of the scroll bar.
8.	*int getValue()*	It fetches the current value of scroll bar.
9.	*void setBlockIncrement(int v)*	It sets the block increment from scroll bar.
10.	*void setMaximum (int newMaximum)*	It sets the maximum value of the scroll bar.
11.	*void setMinimum (int newMinimum)*	It sets the minimum value of the scroll bar.
12.	*void setOrientation (int orientation)*	It sets the orientation for the scroll bar.
13.	*void setUnitIncrement(int v)*	It sets the unit increment for the scroll bar.
14.	*void setValue (int newValue)*	It sets the value of scroll bar with the given argument value.
15.	*void setValueIsAdjusting (boolean b)*	It sets the valueIsAdjusting property to scroll bar.
16.	*void setValues (int value, int visible, int minimum, int maximum)*	It sets the values of four properties for scroll bar: value, visible amount, minimum and maximum.

Example:

```
// importing awt package
import java.awt.*;

public class ScrollbarExample1 {

// class constructor
ScrollbarExample1() {

        // creating a frame
             Frame f = new Frame("Scrollbar Example");
        // creating a scroll bar
             Scrollbar s = new Scrollbar();

        // setting the position of scroll bar
             s.setBounds (100, 100, 50, 100);

        // adding scroll bar to the frame
             f.add(s);

        // setting size, layout and visibility of frame
             f.setSize(400, 400);
             f.setLayout(null);
             f.setVisible(true);
}

// main method
public static void main(String args[]) {
        new ScrollbarExample1();
}
}
```



6. List:

Constructors

Sr. no.	Constructor	Description
1.	`List()`	It constructs a new scrolling list.
2.	`List(int row_num)`	It constructs a new scrolling list initialized with the given number of rows visible.
3.	`List(int row_num, Boolean multipleMode)`	It constructs a new scrolling list initialized which displays the given number of rows.

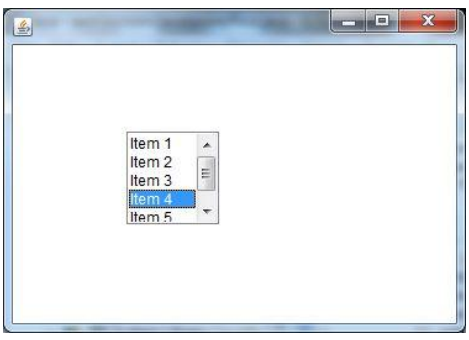
Methods

Sr. no.	Method name	Description
1.	`void add(String item)`	It adds the specified item into the end of scrolling list.
2.	`void add(String item, int index)`	It adds the specified item into list at the

		given index position.
3.	*void addActionListener(ActionListener l)*	It adds the specified action listener to receive action events from list.
4.	*void addItemListener(ItemListener l)*	It adds specified item listener to receive item events from list.
5.	*void deselect(int index)*	It deselects the item at given index position.
6.	*String getItem(int index)*	It fetches the item related to given index position.
7.	*int getItemCount()*	It gets the count/number of items in the list.
8.	*ItemListener[] getItemListeners()*	It returns an array of item listeners registered on the list.
9.	*String[] getItems()*	It fetched the items from the list.
10.	*int getRows()*	It fetches the count of visible rows in the list.
11.	*int getSelectedIndex()*	It fetches the index of selected item of list.
12.	*int[] getSelectedIndexes()*	It gets the selected indices of the list.
13.	*String getSelectedItem()*	It gets the selected item on the list.
14.	*String[] getSelectedItems()*	It gets the selected items on the list.
15.	*int getVisibleIndex()*	It gets the index of an item which was made visible by method makeVisible()
16.	*void makeVisible(int index)*	It makes the item at given index visible.
17.	*boolean isIndexSelected(int index)*	It returns true if given item in the list is selected.
18.	*boolean isMultipleMode()*	It returns the true if list allows multiple selections.
19.	*void remove(int position)*	It removes the item at given index position from the list.
20.	*void remove(String item)*	It removes the first occurrence of an item from list.
21.	*void removeAll()*	It removes all the items from the list.
22.	*void replaceItem(String newVal, int index)*	It replaces the item at the given index in list with the new string specified.
23.	*void select(int index)*	It selects the item at given index in the list.
24.	*void setMultipleMode(boolean b)*	It sets the flag which determines whether the list will allow multiple selection or not.

Example:

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

public class ListExample1
{
    // class constructor
     ListExample1() {
     // creating the frame
        Frame f = new Frame();
```



```
        // creating the list of 5 rows
         List l1 = new List(5);

         // setting the position of list component
         l1.setBounds(100, 100, 75, 75);

         // adding list items into the list
         l1.add("Item 1");
         l1.add("Item 2");
         l1.add("Item 3");
         l1.add("Item 4");
         l1.add("Item 5");

         // adding the list to frame
         f.add(l1);

         // setting size, layout and visibility of frame
         f.setSize(400, 400);
         f.setLayout(null);
         f.setVisible(true);
     }

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

7. Checkbox:

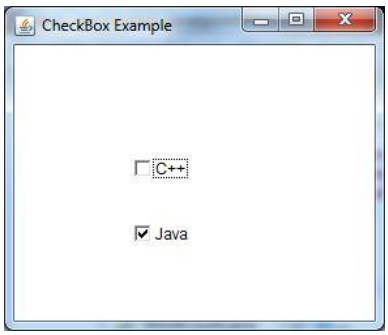
Constructors

Sr. no.	Constructor	Description
1.	`Checkbox()`	It constructs a checkbox with no string as the label.
2.	`Checkbox(String label)`	It constructs a checkbox with the given label.
3.	`Checkbox(String label, boolean state)`	It constructs a checkbox with the given label and sets the given state.
4.	`Checkbox(String label, boolean state, CheckboxGroup group)`	It constructs a checkbox with the given label, set the given state in the specified checkbox group.
5.	`Checkbox(String label, CheckboxGroup group, boolean state)`	It constructs a checkbox with the given label, in the given checkbox group and set to the specified state.

Methods

Sr. no.	Method name	Description
1.	*void addItemListener(ItemListener IL)*	It adds the given item listener to get the item events from the checkbox.
2.	*void addNotify()*	It creates the peer of checkbox.
3.	*CheckboxGroup getCheckboxGroup()*	It determines the group of checkbox.
4.	*String getLabel()*	It fetched the label of checkbox.
5.	*boolean getState()*	It returns true if the checkbox is on, else returns off.
7.	*void setCheckboxGroup(CheckboxGroup g)*	It sets the checkbox's group to the given checkbox.
8.	*void setLabel(String label)*	It sets the checkbox's label to the string argument.
9.	*void setState(boolean state)*	It sets the state of checkbox to the specified state.

Example:




```
// importing AWT class
import java.awt.*;
public class CheckboxExample1
{
     CheckboxExample1() {

       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);
     }
       // main method
       public static void main (String args[])
       {
           new CheckboxExample1();
       }
}
```

LayoutManagers:

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represent the layout managers:

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

BorderLayout:

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

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

Constructors of BorderLayout class:

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

Example of BorderLayout class:

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

public class Border {
JFrame f;
Border(){
    f=new JFrame();

    JButton b1=new JButton("NORTH");;
    JButton b2=new JButton("SOUTH");;
    JButton b3=new JButton("EAST");;
    JButton b4=new JButton("WEST");;
    JButton b5=new JButton("CENTER");;

    f.add(b1,BorderLayout.NORTH);
    f.add(b2,BorderLayout.SOUTH);
    f.add(b3,BorderLayout.EAST);
    f.add(b4,BorderLayout.WEST);
    f.add(b5,BorderLayout.CENTER);
```

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

GridLayout

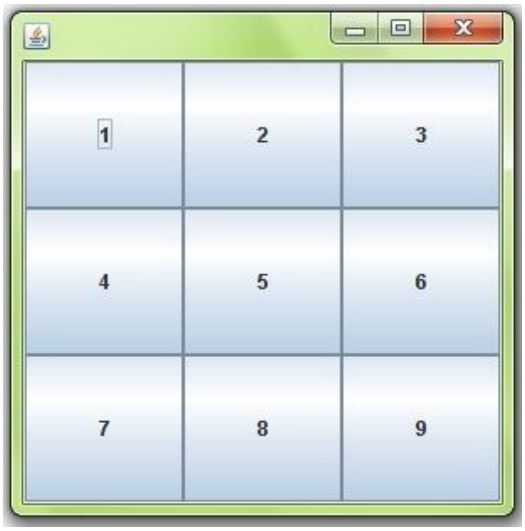
The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class:

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

Example of GridLayout class:

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

public class MyGridLayout{
JFrame f;
MyGridLayout(){
    f=new JFrame();

    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");
        JButton b6=new JButton("6");
        JButton b7=new JButton("7");
    JButton b8=new JButton("8");
        JButton b9=new JButton("9");

    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.add(b6);f.add(b7);f.add(b8);f.add(b9);

    f.setLayout(new GridLayout(3,3));
    //setting grid layout of 3 rows and 3 columns

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



FlowLayout

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

Fields of FlowLayout class:

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

Constructors of FlowLayout class:

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

Example of FlowLayout class:

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

public class MyFlowLayout{
JFrame f;
MyFlowLayout(){
    f=new JFrame();

    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");

    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);

    f.setLayout(new FlowLayout(FlowLayout.RIGHT));
    //setting flow layout of right alignment

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

BoxLayout class:

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

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

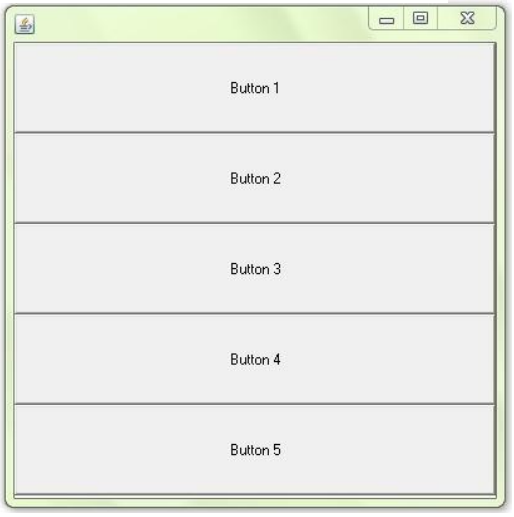
Fields of BoxLayout class:

1. **public static final int X_AXIS**
2. **public static final int Y_AXIS**
3. **public static final int LINE_AXIS**
4. **public static final int PAGE_AXIS**

Constructor of BoxLayout class:

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

Example of BoxLayout class with Y-AXIS:




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

public class BoxLayoutExample1 extends Frame {
 Button buttons[];

 public BoxLayoutExample1 () {
   buttons = new Button [5];

   for (int i = 0;i<5;i++) {
      buttons[i] = new Button ("Button " + (i + 1));
      add (buttons[i]);
    }

setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
setSize(400,400);
setVisible(true);
}

public static void main(String args[]){
BoxLayoutExample1 b=new BoxLayoutExample1();
}
}
```

CardLayout class

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

Constructors of CardLayout class:

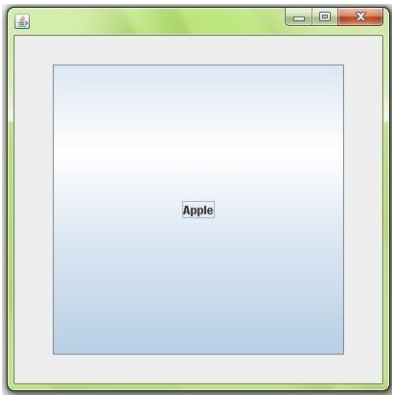
1. **CardLayout():** creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

Commonly used methods of CardLayout class:

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

Example of CardLayout class:

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

import javax.swing.*;

public class CardLayoutExample extends JFrame implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
    CardLayoutExample(){

        c=getContentPane();
        card=new CardLayout(40,30);
        c.setLayout(card);

        b1=new JButton("Apple");
        b2=new JButton("Boy");
        b3=new JButton("Cat");
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);

        c.add("a",b1);c.add("b",b2);c.add("c",b3);

    }
```



```
    public void actionPerformed(ActionEvent e) {
    card.next(c);
    }

    public static void main(String[] args) {
        CardLayoutExample cl=new CardLayoutExample();
        cl.setSize(400,400);
        cl.setVisible(true);
        cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

Swings:

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

What is JFC

The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

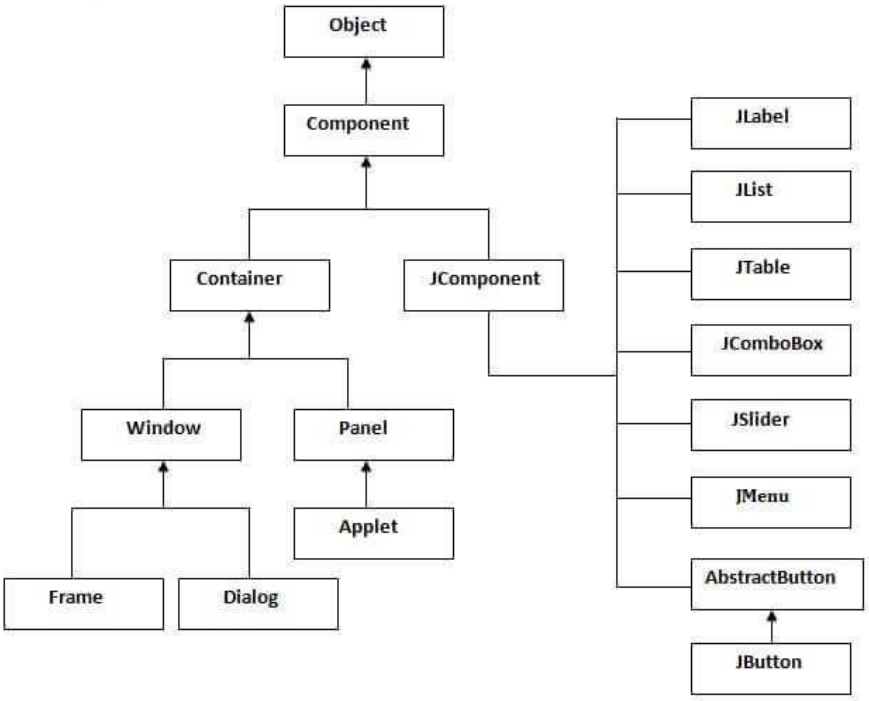
Do You Know

- How to create runnable jar file in java?
- How to display image on a button in swing?
- How to change the component color by choosing a color from ColorChooser ?
- How to display the digital watch in swing tutorial ?
- How to create a notepad in swing?

- How to create puzzle game and pic puzzle game in swing ?
- How to create tic tac toe game in swing ?

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Methods of Component class

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

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