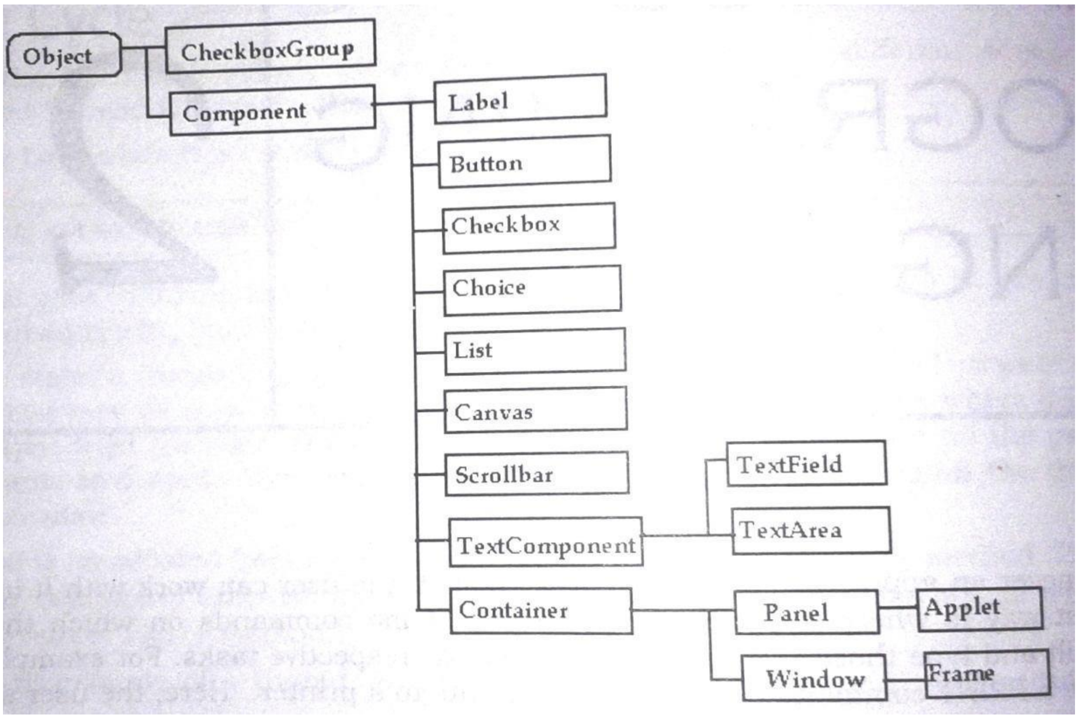
AWT (Abstract Window Toolkit):

AWT represents a class library to develop applications using GUI. The **java.awt** package consists of classes and interfaces to develop GUIs.

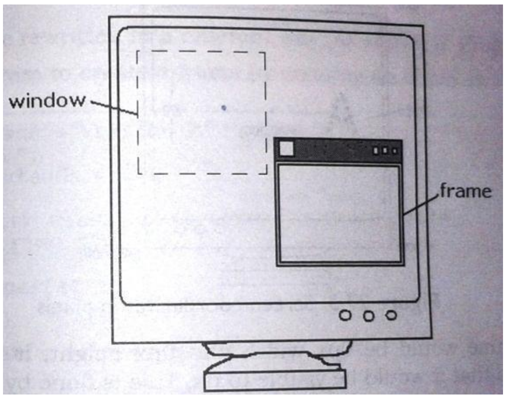


Component: A component represents an object which is displayed pictorially on the screen and interacts with the user.

Ex. Button, TextField, TextArea

Container: A Container is a subclass of Component; it has methods that allow other components to be nested in it. A container is responsible for laying out (that is positioning) any component that it contains. It does this with various layout managers.

Panel: Panel class is a subclass of Container and is a super class of Applet. When screen output is redirected to an applet, it is drawn on the surface of the Panel object. In, essence panel is a window that does not contain a title bar, menu bar or border.

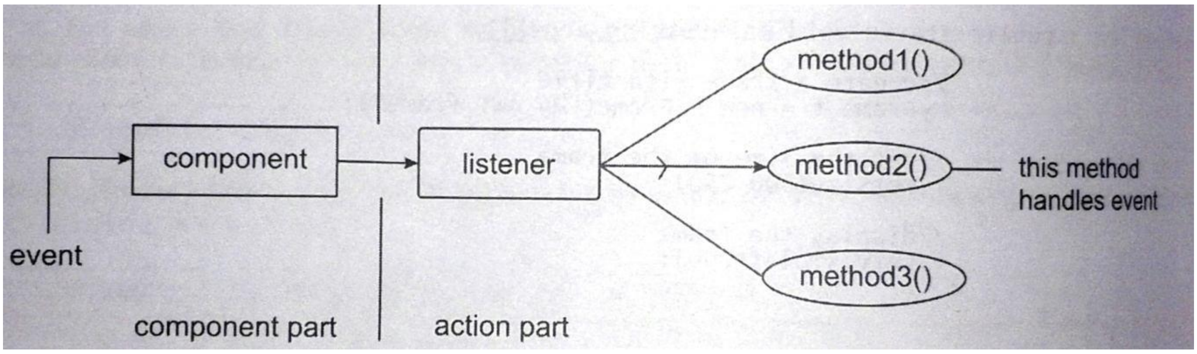


Window: A window represents a rectangular area on the screen without any borders or title bar. The Window class create a top-level window.

Frame: It is a subclass of Window and it has title bar, menu bar, border and resizing windows.

Delegation Event Model:

The modern approach (from version 1.1 onwards) to handle events is based on the delegation event model. Its concept is quite simple: 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. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.

A user interface element is able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

Events: An *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a GUI. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sources: A source is an object that generates an event. Generally sources are components. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void addTypeListener (TypeListener el)

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

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener el)

Event Listeners: A listener is an object that is notified when an event occurs. It has two major requirements.

1.It must have been registered with one or more sources to receive notifications aboutspecific types of events.

2. It must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in *java.awt.event* package.

Sources of Events:

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked;
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Classes and Listener Interfaces:

The java.awt.event package provides many event classes and Listener interfaces for event handling. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the super class for all events. Its one constructor is shown here:

EventObject(Object *src*) - Here, *src* is the object that generates this event.

EventObject contains two methods:

getSource( ) - returns the source of the event.
toString( ) - toString( ) returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

int getID( )

.

The package **java.awt.event** defines many types of events that are generated by various user interface elements

Event Class	Description	Listener Interface
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.	ActionListener
AdjustmentEvent	Generated when a scroll bar is manipulated.	AdjustmentListener
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.	ComponentListener
ContainerEvent	Generated when a component is added to or removed from a container.	ContainerListener
FocusEvent	Generated when a component gains or losses keyboard focus.	FocusListener
InputEvent	Abstract super class for all component input event classes.	
ItemEvent	Generated when a check box or list item is clicked	ItemListener
KeyEvent	Generated when input is received from the keyboard.	KeyListener
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.	MouseListener and MouseMotionListener
TextEvent	Generated when the value of a text area or text field is changed.	TextListener
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.	WindowListener

Useful Methods of Component class:

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

The ActionEvent Class:

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK (Ex. Escape) , and SHIFT_MASK.

ActionEvent has these three constructors:

- ActionEvent(Object src, int type, String cmd)
- ActionEvent(Object src, int type, String cmd, int modifiers)
- ActionEvent(Object src, int type, String cmd, long when, int modifiers)

You can obtain the command name for the invoking ActionEvent object by using the getActionCommand( ) method, shown here:

String getActionCommand( )

The AdjustmentEvent Class:

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events.

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

The ComponentEvent Class:

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**, among others.

The **getComponent( )** method returns the component that generated the event. It is shown here:

Component getComponent( )

The ContainerEvent Class:

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**.

The FocusEvent Class:

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.

The InputEvent Class:

The abstract class **InputEvent** is a subclass of **ComponentEven**t and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the InputEvent class defined the following eight values to represent the modifiers:

ALT_MASK	**ALT_GRAPH_MASK**	**BUTTON2_MASK**	**BUTTON3_MASK**
BUTTON1_MASK	**CTRL_MASK**	**META_MASK**	**SHIFT_MASK**

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	**ALT_GRAPH_DOWN_MASK**	**BUTTON1_DOWN_MASK**
BUTTON2_DOWN_MASK	**BUTTON3_DOWN_MASK**	**CTRL_DOWN_MASK**
META_DOWN_MASK	**SHIFT_DOWN_MASK**	

The KeyEvent Class

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

The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing shift does not generate a character.

There are many other integer constants that are defined by KeyEvent. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters.

The MouseEvent Class:

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse
MOUSE_DRAGGED	The user dragged the mouse
MOUSE_ENTERED	The mouse entered a component
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

```
int getX( )
int getY( )
```

The TextEvent Class:

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. TextEvent defines the integer constant **TEXT_VALUE_CHANGED.**

The WindowEvent Class:

The **WindowEvent** class defines integer constants that can be used to identify different types of events:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window was iconified.
WINDOW_ICONIFIED	The window gained input focus.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.

EventListener Interfaces:

An event listener registers with an event source to receive notifications about the events of a particular type. Various event listener interfaces defined in the `java.awt.event` package are given below:

Interface	Description
ActionListener	Defines the actionPerformed() method to receive and process action events. *void actionPerformed(ActionEvent ae)*
MouseListener	Defines five methods to receive mouse events, such as when a mouse is clicked, pressed, released, enters, or exits a component *void mouseClicked(MouseEvent me)* *void mouseEntered(MouseEvent me)* *void mouseExited(MouseEvent me)* *void mousePressed(MouseEvent me)* *void mouseReleased(MouseEvent me)*
MouseMotionListener	Defines two methods to receive events, such as when a mouse is dragged or moved. *void mouseDragged(MouseEvent me)* *void mouseMoved(MouseEvent me)*
AdjustmentListner	Defines the adjustmentValueChanged() method to receive and process the adjustment events. *void adjustmentValueChanged(AdjustmentEvent ae)*
TextListener	Defines the textValueChanged() method to receive and process an event when the text value changes. *void textValueChanged(TextEvent te)*
WindowListener	Defines seven window methods to receive events. *void windowActivated(WindowEvent we)* *void windowClosed(WindowEvent we)* *void windowClosing(WindowEvent we)* *void windowDeactivated(WindowEvent we)* *void windowDeiconified(WindowEvent we)* *void windowIconified(WindowEvent we)* *void windowOpened(WindowEvent we)*
ItemListener	Defines the itemStateChanged() method when an item has been *void itemStateChanged(ItemEvent ie)*
WindowFocusListener	This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here: *void windowGainedFocus(WindowEvent we)* *void windowLostFocus(WindowEvent we)*
ComponentListener	This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here: *void componentResized(ComponentEvent ce)* *void componentMoved(ComponentEvent ce)* *void componentShown(ComponentEvent ce)* *void componentHidden(ComponentEvent ce)*

ContainerListener	This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here: *void componentAdded(ContainerEvent ce)* *void componentRemoved(ContainerEvent ce)*
FocusListener	This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here: *void focusGained(FocusEvent fe)* *void focusLost(FocusEvent fe)*
KeyListener	This interface defines three methods. *void keyPressed(KeyEvent ke)* *void keyReleased(KeyEvent ke)* *void keyTyped(KeyEvent ke)*

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener
2. Implement the concerned interface

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){}

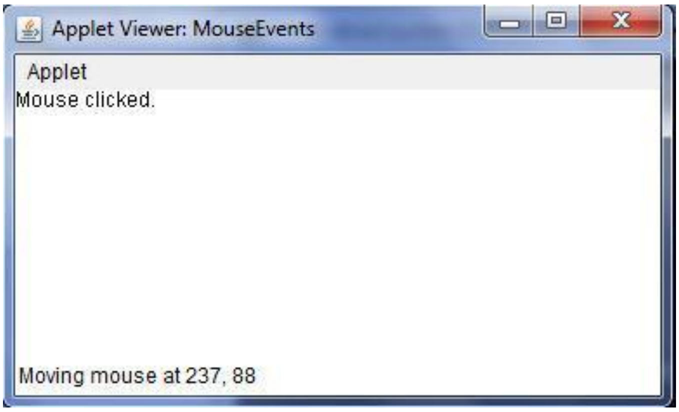
Mouse

- public void addMouseListener(MouseListener a){}

Handling Mouse Events Example Program:

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*; /*

<applet code="MouseEvents" width=300
height=100> </applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener
{
        String msg = "";
        int mouseX = 0, mouseY = 0; // coordinates of mouse
        public void init()
        {
                addMouseListener(this);
                addMouseMotionListener(this);
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {
                // save coordinates
                mouseX = 0;
                mouseY = 10;
                msg = "Mouse
                clicked."; repaint();
        }
        // Handle mouse entered.
        public void mouseEntered(MouseEvent me)
        {
                // save coordinates
                mouseX = 0;
                mouseY = 10;
                msg = "Mouse entered.";
                repaint();
        }
        // Handle mouse exited.
        public void mouseExited(MouseEvent me)
        {
                // save coordinates
                mouseX = 0;
                mouseY = 10;
                msg = "Mouse exited.";
                repaint();
        }
        // Handle button pressed.
        public void mousePressed(MouseEvent me)
        {
                // save coordinates
                mouseX = me.getX();
                mouseY = me.getY();
                msg = "Down";
                repaint();
        }
        // Handle button released.
        public void mouseReleased(MouseEvent me)
        {
                // save coordinates
                mouseX =
                me.getX(); mouseY
                = me.getY(); msg =
                "Up"; repaint();
        }
        // Handle mouse dragged.
        public void mouseDragged(MouseEvent me)
        {
                // save coordinates
                mouseX = me.getX();
                mouseY = me.getY();
                msg = "*";
                showStatus("Dragging mouse at " + mouseX + ", " +
                mouseY); repaint();
        }
        // Handle mouse moved.
        public void mouseMoved(MouseEvent me)
        {
                // show status
                showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
        }
        // Display msg in applet window at current X,Y
        location. public void paint(Graphics g)
        {
                g.drawString(msg, mouseX, mouseY);
        }
}
```

Output:



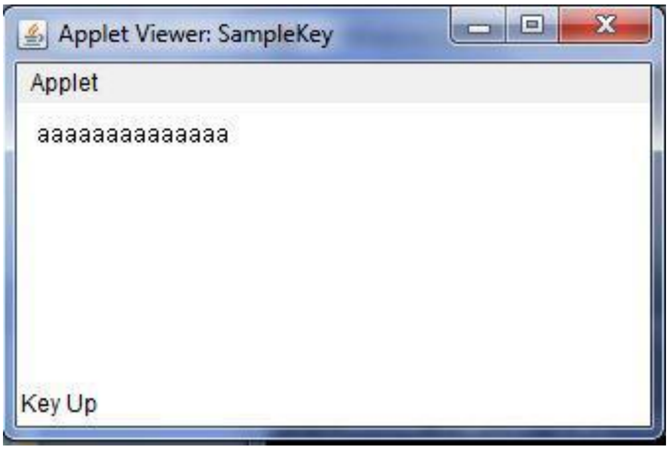
```
Mouse clicked.
```

```
Moving mouse at 237, 88
```

Handling Key Board Events:

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*; /*

<applet code="SampleKey" width=300 height=100>
</applet>
*/
public class SampleKey extends Applet implements KeyListener
{
        String msg = "";

        public void init() {
                addKeyListener(this);
        }
        public void keyPressed(KeyEvent ke) {
                showStatus("Key Down");
        }
        public void keyReleased(KeyEvent ke) {
                showStatus("Key Up");
        }
        public void keyTyped(KeyEvent ke) {
                msg += ke.getKeyChar();
                repaint();
        }
        // Display keystrokes.
        public void paint(Graphics g) {
                g.drawString(msg, 10, 20);
        }
}
```

Output:



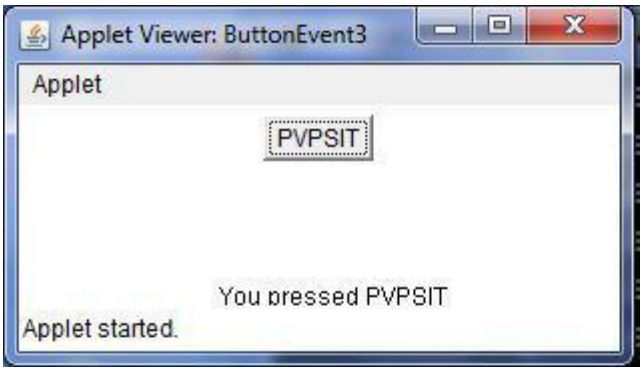
```
aaaaaaaaaaaaaa
```

Key Up

Handling Action Event Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="ButtonEvent3" width=300 height=100>
</applet>
*/
public class ButtonEvent3 extends Applet implements ActionListener
  {
          Button a ;
          String msg;
          public void init()
          {
                    a=new Button("PVPSIT");

                    add(a);

                    a.addActionListener(this);
           }
          public void actionPerformed(ActionEvent ae)
          {
                    String str=ae.getActionCommand();
                    if(str.equals("PVPSIT"))
                    msg="You pressed PVPSIT";
                    repaint();

          }
          public void paint(Graphics g)
          {
                    g.drawString(msg,100,100);

          }
  }
```

Output:



You pressed PVPSIT

Applet started.

Adapter Classes:

Java provides a special feature, called an *adapter class,* that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

For example,

MouseListener	MouseAdapter

void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
	void mouseClicked(MouseEvent me){ }
	void mouseEntered(MouseEvent me) { }
	void mouseExited(MouseEvent me) { }
	void mousePressed(MouseEvent me) { }
	void mouseReleased(MouseEvent me) { }

Table: Commonly used Listener Interfaces implemented by Adapter Classes

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

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet
{
  public void init() {
            addMouseListener(new MyMouseAdapter(this));

  }
}
class MyMouseAdapter extends MouseAdapter
{
  AdapterDemo ad;
  public MyMouseAdapter(AdapterDemo ad)
  {
            this.ad = ad;
```

```
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {
                ad.showStatus("Mouse clicked");
        }
      }
```

Inner Classes:

I*nner class* is a class defined within another class, or even within an expression.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="InnerClassDemo" width=300 height=100>
</applet>
*/
public class InnerClassDemo extends Applet
{
        String msg = "hello";

        public void init() {
                addKeyListener(new MyKeyIn());
        }
        class MyKeyIn extends KeyAdapter
        {
                 public void keyPressed(KeyEvent ke) {
                        showStatus("Key Pressed");
                }
        }

        public void paint(Graphics g) {
                g.drawString(msg, 10, 20);
        }
}
```

Anonymous Inner Classes:

An *anonymous* inner class is one that is not assigned a name.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AInnerClassDemo" width=300 height=100>
</applet> */
```

```
public class AInnerClassDemo extends Applet
{
  String msg = "hello";

  public void init()
  {
          addKeyListener(new KeyAdapter(){
                  public void keyPressed(KeyEvent ke) {
                  showStatus("Key Pressed");
                  }
          });
  }
  // Display keystrokes.
  public void paint(Graphics g) {
          g.drawString(msg, 10, 20);
  }
}
```

Control Fundamentals:

The AWT supports the following types of controls:

Labels
Push buttons
Check boxes
Choice lists
Lists
Scroll bars
Text Editing

These controls are subclasses of **Component**

Adding and Removing Controls: To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add()**, which is defined by **Container**. The General form is:

Component add(Component *compObj*)

Here, *compObj* is an instance of the control that you want to add. A reference to *compObj* is returned.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove( )**. This method is also defined by **Container**. Here is one of its forms:

void remove(Component *obj*)

Here, *obj* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

The HeadlessException:

Most of the AWT controls have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present).

Labels:

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. **Label** defines the following constructors:

Label( ) throws HeadlessException Label(String *str*)
throws HeadlessException Label(String *str*, int *how*)
throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

Using Buttons:

A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String *str*)
String getLabel( )

Here, *str* becomes the new label for the button

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="ButtonEvent1" width=300 height=100>
</applet>
*/
public class ButtonEvent1 extends Applet
  {
    Button b,b1;
    public void init()
    {
            b=new Button("PVPSIT");
            b1=new Button();
            add(b);
            add(b1);
    } }
```

Check Boxes:

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.
Checkbox supports these constructors: Checkbox( ) throws
HeadlessException Checkbox(String *str*) throws
HeadlessException Checkbox(String *str*, boolean *on*)
throws HeadlessException
Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*) throws HeadlessException
Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*) throws HeadlessException

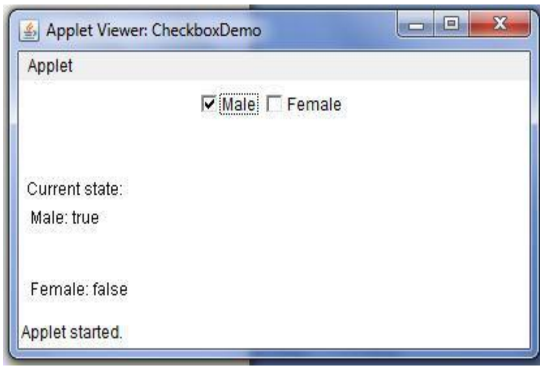
The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. The value of *on* determines the initial state of the check box.

Methods:

boolean getState( ) - To retrieve the current state of a check box
void setState(boolean *on*) - to set the state of a check box String
getLabel( ) – returns the label associated with check box void
setLabel(String *str*) – to set the label

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=240 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener
{
        String msg = "";
        Checkbox m,f;
        public void init()
        {
                m = new Checkbox("Male", true);
                f = new Checkbox("Female");

                add(m );
                add(f);
                m.addItemListener(this);
                f.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g) {
                msg = "Current state: ";
                g.drawString(msg, 6, 80); msg
                = " Male: " + m.getState();
                g.drawString(msg, 6, 100);

                msg = " Female: " + f.getState();
                g.drawString(msg, 6, 150);
        }
}
```



Current state:
Male: true
Female: false
Applet started.

CheckboxGroup:

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons* —only one button can be selected at any one time.

To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**.

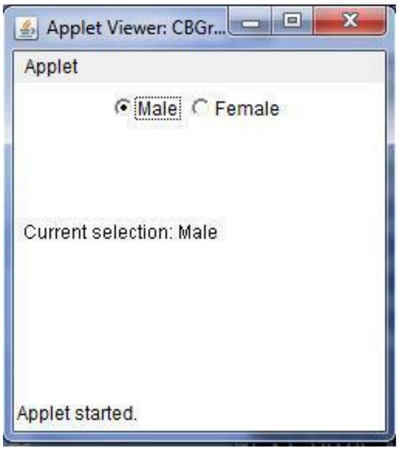
Only the default constructor is defined, which creates an empty group.

Methods:

Checkbox getSelectedCheckbox( ) - which check box in a group is currently selected void
setSelectedCheckbox(Checkbox *which*) - *which* is the check box that you want to
be selected. The previously selected check box will be turned off

Example:

```
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=240 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener
{
        String msg = "";
        Checkbox m,f;
        CheckboxGroup cbg;
        public void init()
        {
                cbg = new CheckboxGroup();
                m = new Checkbox("Male", cbg, true); f
                = new Checkbox("Female", cbg, false);

                add(m);
                add(f);

                m.addItemListener(this);
                f.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g)
        {
                msg = "Current selection: ";
                msg += cbg.getSelectedCheckbox().getLabel();
                g.drawString(msg, 6, 100);
        }
}
```



```
Current selection: Male
```

Applet started.

Choice Controls:

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. **Choice** defines only the default constructor, which creates an empty list. To add a selection to the list, call **add( )**. It has this general form:

void add(String *name*) - *name* is the name of the item being added.

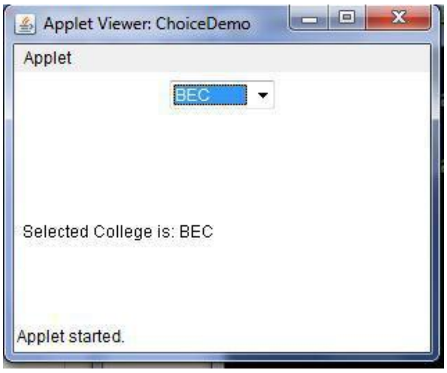
Items are added to the list in the order in which calls to **add( )** occur.

Methods:

String getSelectedItem( ) – returns the item which is currently selected
int getSelectedIndex( ) - returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.
int getItemCount( ) – returns number of items in the list
void select(int *index*) - to set the currently selected item with index void
select(String *name*) - to set the currently selected item with a string
String getItem(int *index*) – returns the name associated with the index

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
        <applet code="ChoiceDemo" width=300
        height=180> </applet>
*/
public class ChoiceDemo extends Applet implements ItemListener
{
        Choice college ;
        String msg = "";
        public void init()
        {
                college = new Choice();

                // add items to os list
                college.add("PVPSIT");
                college.add("BEC");
                college.add("RVR&JC");
                college.add("VRSEC");

                add(college);

                // register to receive item events
                college.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }
        // Display current selections.
        public void paint(Graphics g)
        {
                msg = "Selected College is: ";
                msg += college.getSelectedItem();
                g.drawString(msg, 6, 120);
        }
}
```



Selected College is: BEC

Applet started.

List:

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections.

List provides these constructors:

List( ) throws HeadlessException

List(int *numRows*) throws HeadlessException

List(int *numRows*, boolean *multipleSelect*) throws HeadlessException

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add( )**. It has the following two forms:

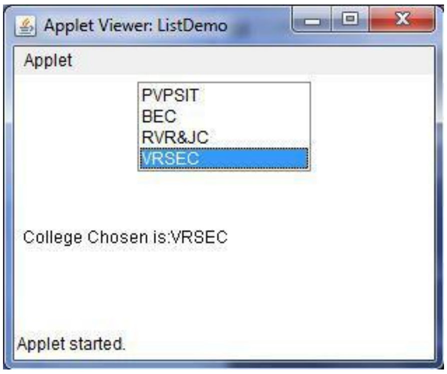
void add(String *name*)

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

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
	<applet code="ListDemo" width=300 height=180>
	</applet>
*/
public class ListDemo extends Applet implements ActionListener
{
	List college;
	String msg = "";
	public void init()
	{
		college = new List(4,true);

		college.add("PVPSIT");
		college.add("BEC");
		college.add("RVR&JC");
		college.add("VRSEC");

		//college.select(1);
		add(college);

		// register to receive action events
		college.addActionListener(this);
		}
	public void actionPerformed(ActionEvent ae) {
		repaint();
	}
```



PVPSIT
BEC
RVR&JC
VRSEC

College Chosen is:VRSEC

Applet started.

```
        // Display current selections.
        public void paint(Graphics g)
        {
                msg="College Chosen is:";
                int ind[];

                ind = college.getSelectedIndexes();
                for(int i=0; i<ind.length; i++)
                        msg += college.getItem(ind[i]) + " ";
                g.drawString(msg, 6, 120);
        }
}
```

TextField:

The **TextField** class implements a single-line text-entry area. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.

TextField is a subclass of **TextComponent**. **TextField** defines the following constructors:

TextField( ) throws HeadlessException TextField(int
numChars) throws HeadlessException TextField(String *str*)
throws HeadlessException TextField(String *str*, int *numChars*)
throws HeadlessException

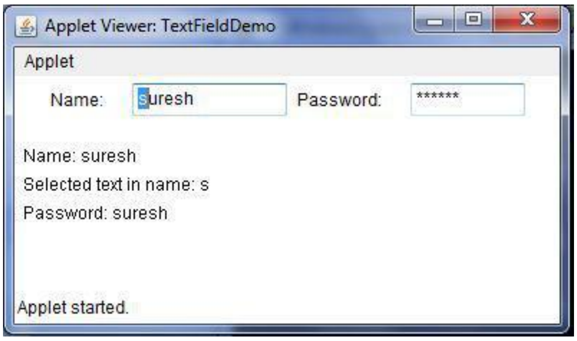
The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width.

Methods:

String getText( ) - To obtain the string currently contained in the text
field void setText(String *str*) - To set the text, here, *str* is the new string.
String getSelectedText( ) - returns currently selected text
void select(int *startIndex*, int *endIndex*) - selects the characters beginning at
startIndex and ending at *endIndex* –1.
boolean isEditable( ) – returns boolean value (true/false)
void setEditable(boolean *canEdit*) - if *canEdit* is **true**, the text may be changed. If it
is **false**, the text cannot be altered.
void setEchoChar(char *ch*) – specified echo character will be displayed in TextField
boolean echoCharIsSet( ) –returns true or false
char getEchoChar( ) – returns the echo character

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet implements TextListener
{
        TextField name, pass;
        public void init()
        {
                Label namep = new Label("Name: " );
                name = new TextField(12);

                Label passp = new Label("Password: " );
                pass = new TextField(8);
                pass.setEchoChar('*');

                add(namep);
                add(name);
                add(passp);
                add(pass);
                // register to receive action events
                name.addTextListener(this);
                pass.addTextListener(this);
        }
        // User pressed Enter.
        public void textValueChanged(TextEvent ae)
        {
                repaint();
        }
        public void paint(Graphics g)
        {
                g.drawString("Name: " + name.getText(), 6, 60);
                g.drawString("Password: " + pass.getText(), 6, 100);
                g.drawString("Selected text in name: "+ name.getSelectedText(), 6, 80);
        }
}
```



Name: suresh Password: ******

```
Name: suresh
Selected text in name: s
Password: suresh
```

Applet started.

TextArea:

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

TextArea( ) throws HeadlessException
TextArea(int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws
HeadlessException

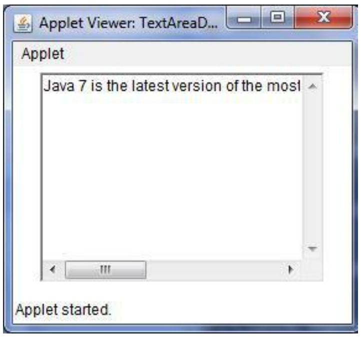
Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

SCROLLBARS_BOTH
SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY
SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described in the preceding section.

TextArea adds the following methods:

void append(String *str*) - appends the string specified by *str* to the end
of the current void insert(String *str*, int *index*) - inserts the string
passed in *str* at the specified index void replaceRange(String *str*, int
startIndex, int *endIndex*) - replaces the characters from *startIndex* to
endIndex–1, with the replacement text passed in *str*



Java 7 is the latest version of the most

Applet started.

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet
{
        public void init()
        {
                String val = "Java 7 is the latest version of the most widely-used computer
                                language for Internet programming.";
                        TextArea text = new TextArea(val, 10, 30);
                        add(text);
        }
}
```

Managing Scroll Bars:

Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

Scroll bars are encapsulated by the **Scrollbar** class. **Scrollbar** defines the following constructors:

Scrollbar( ) throws HeadlessException
Scrollbar(int *style*) throws HeadlessException
Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*) throws HeadlessException

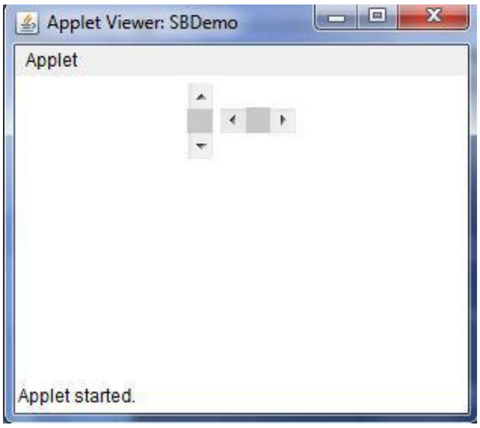
The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

Methods:

void setValues(int *initialValue*, int *thumbSize*, int *min*, int *max*)	If we construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues()**
int getValue( )	To get the current value
void setValue(int *newValue*)	TO set the current value
int getMinimum( )	To get the minimum value
int getMaximum( )	To get the maximum value

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
        <applet code="SBDemo" width=300 height=200>
        </applet>
*/
public class SBDemo extends Applet
{
         Scrollbar vertSB, horzSB;

        public void init()
        {
                vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, 100);
                horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 100);
                add(vertSB);
                add(horzSB);
        }
}
```



Applet started.

Layout Manager

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

Java soft people have created a LayoutManager interface in java.awt package which is implemented in various classes which provide various types of layouts to arrange the components. The following classes represents the layout managers in Java:

1. FlowLayout
2. BorderLayout
3. GridLayout
4. CardLayout
5. GridBagLayout
6. BoxLayout

To set a particular layout, we should first create an object to the layout class and pass the object to setLayout() method. For example, to set FlowLayout to the container:

```
FlowLayout obj=new FlowLayout();
c. setLayout(obj); // assume c is container
```

FlowLayout:

FlowLayout is useful to arrange the components in a line one after the other. When a line is filled with components, they are automatically placed in a next line. This is the default layout in applets.

Constructors:

```
FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

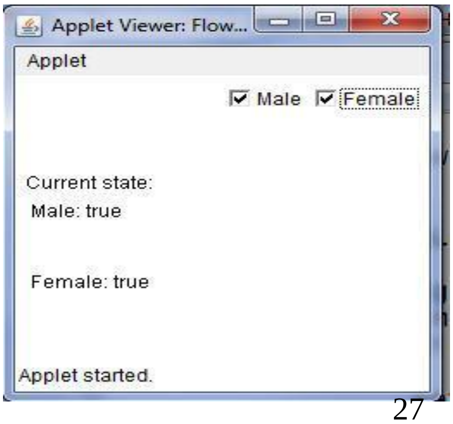
The first form creates the default layout, which centres components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for how are as follows:

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
```

The third constructor allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=240
height=200>
</applet>
*/
```



☑ Male ☑ Female

Current state:
Male: true

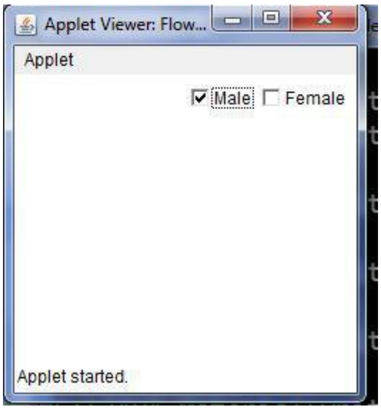
Female: true

Applet started.

```
public class FlowLayoutDemo extends Applet implements ItemListener
{
    String msg="";
    Checkbox m,f;
    public void init()
    {
        setLayout(new FlowLayout(FlowLayout.RIGHT));
        m = new Checkbox("Male", true); f = new
        Checkbox("Female");
        add(m );
        add(f);
        m.addItemListener(this);
        f.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        repaint();
    }
    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80); msg
        = " Male: " + m.getState();
        g.drawString(msg, 6, 100);

        msg = " Female: " + f.getState();
        g.drawString(msg, 6, 150);
    }
}
```

(or)

```
/*
<applet code="FlowLayoutDemo" width=240 height=200>
</applet> */
public class FlowLayoutDemo extends Applet
{
    Checkbox m,f;
    public void init()
    {
        setLayout(new
FlowLayout(FlowLayout.RIGHT));
        m = new Checkbox("Male", true);
        f = new Checkbox("Female");

        add(m );
        add(f);


    }
}
```



BorderLayout:

BorderLayout is useful to arrange the components in the four borders of the frame as well as in the centre. The borders are identified with the names of the directions. The top border is specified as ‘North’, the right side border as ‘East’, the bottom one as ‘South’ and the left one as ‘West’. The centre is represented as ‘Centre’.

Constructors:

BorderLayout( )
BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.
BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER
BorderLayout.SOUTH
BorderLayout.EAST
BorderLayout.WEST
BorderLayout.NORTH

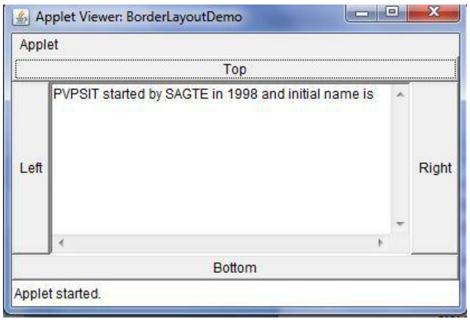
When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

void add(Component *compObj*, Object *region*)

Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

Example:

```
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet
{
        public void init()
        {
                setLayout(new BorderLayout());
                add(new Button("Top"),BorderLayout.NORTH);
                add(new Button("Bottom"),BorderLayout.SOUTH);
                add(new Button("Right"), BorderLayout.EAST);
                add(new Button("Left"), BorderLayout.WEST);
                String msg = "PVPSIT started by SAGTE in 1998.\n";
                add(new TextArea(msg), BorderLayout.CENTER);
        }
}
```



PVPSIT started by SAGTE in 1998 and initial name is

Applet started.

GridLayout:

GridLayout is useful to divide the container into a 2D grid form that contains several rows and columns. The container is divided into equal-sized rectangle; and one component is placed in each rectangle.

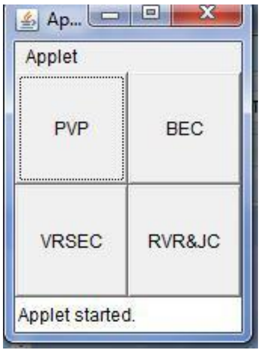
Constructors:

GridLayout( )
GridLayout(int *numRows*, int *numColumns*)
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimitedlength columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Example:

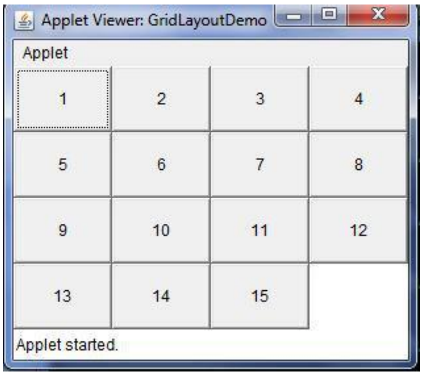
```
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo2" width=150 height=150>
</applet>
*/
public class GridLayoutDemo2 extends Applet
{
        Button b1,b2,b3,b4;
        public void init()
        {
                setLayout(new GridLayout(2, 2));
                b1=new Button("PVP");
                b2=new Button("BEC");
                b3=new Button("VRSEC");
                b4=new Button("RVR&JC");

                add(b1);
                add(b2);
                add(b3);
                add(b4);
        }
}
```



or

```
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
```

```
</applet>
*/
public class GridLayoutDemo extends Applet
{
        static final int n = 4;
        public void init() {
                setLayout(new GridLayout(n, n));

                for(int i = 0; i < n; i++) { for(int
                        j = 0; j < n; j++) {
                                int k = i * n + j;
                                if(k > 0)
                                add(new Button("" + k));
                        }
                }
        }
}
```



Applet started.

CardLayout:

A CardLayout object is a layout manager which treats each component as a card. Only one card is displayed at a time, and the container acts as a stack of cards. The first component added to a CardLayout object is visible component when the container is first displayed.

CardLayout provides these two constructors:

CardLayout( )
CardLayout(int *horz*, int *vert*)

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. Finally, you add this pane to the window.

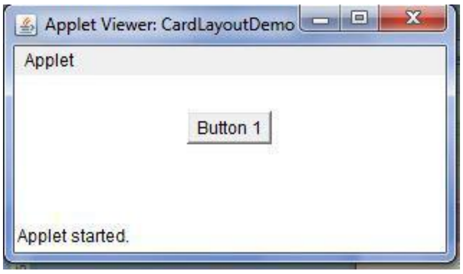
Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add( )** when adding cards to a panel:

void add(Component *panelObj*, Object *name*)
or
void add(Object name, Component *panelObj*)

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelObj*. After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

void first(Container *deck*)
void last(Container *deck*)
void next(Container *deck*)
void previous(Container *deck*)
void show(Container *deck*, String *cardName*)

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutDemo" width=300 height=100>
</applet>
*/
public class CardLayoutDemo extends Applet implements ActionListener
{
        Button b1,b2,b3,b4;
        Panel p;
        CardLayout card;
        public void init()
        {
                b1 = new Button("Button 1");
                b2 = new Button("Button 2");
                b3 = new Button("Button 3");
                b4 = new Button("Button 4");

                p=new Panel();
                card=new CardLayout(20,20);
                p.setLayout(card);

                p.add("First",b1);
                p.add("Second",b2);
                p.add("Third",b3);
                p.add("Fourth",b4);

                b1.addActionListener(this);
                b2.addActionListener(this);
                b3.addActionListener(this);
                b4.addActionListener(this);
                add(p);
        }
        public void actionPerformed(ActionEvent ae)
        {
                card.next(p);
        }
}
```



GridBagLayout:

A GridBagLayout class represents grid bag layout manager where the components are arranged in rows and columns. In this layout the component can span more than one row or column and the size of the component can be adjusted to fit the display area.

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

In order to create GridBagLayout, we first instantiate the GridBagLayout class by using its only no-argument constructor

```
GridBagLayout layout=new GridBagLayout();
setLayout(layout);
```

and defining it as the current layout manager.

To apply constraints on the components, we should first create an object to GridBagConstrinats class, as

```
GridBagConstrinats gbc =new GridBagConstrinats();
```

This will create constraints for the components with default value. The other way to specify the constraints is by directly passing their values while creating the GridBagConstrinats as

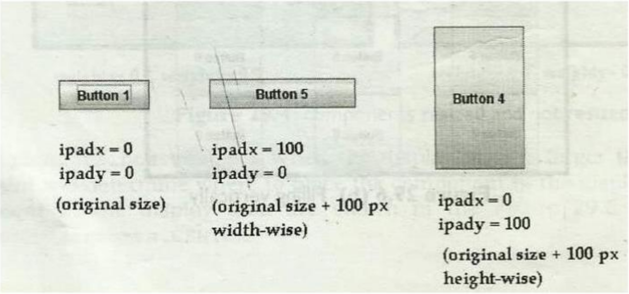
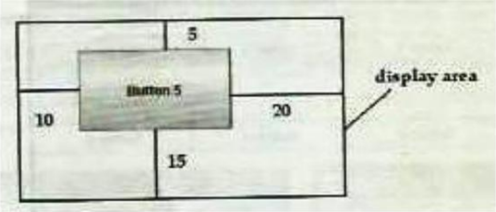
```
GridBagConstrinats gbc= new GridBagConstrinats(
        int gridx, int gridy, int gridwidth, int gridheight, double weightx, double
        weighty, int anchor, int fill, Insets insets, int ipadx, int ipady );
```

To set the constraints use setConstraints() method in GridBagConstrinats class and its prototype

```
void setConstraints(Component comp, GridBagConstraints cons);
```

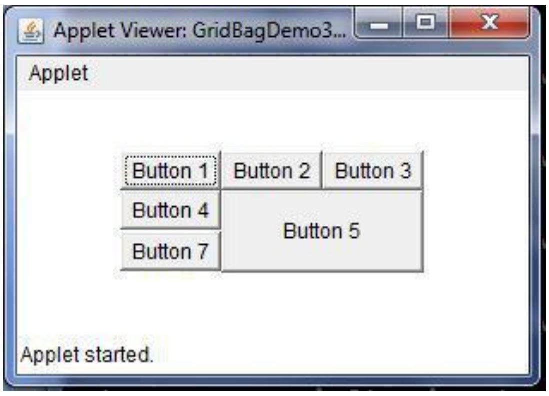
Constraint fields Defined by GridBagConstraints:

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER. Others are GridBagConstraints.EAST GridBagConstraints.WEST GridBagConstraints.SOUTH GridBagConstraints.NORTH GridBagConstraints.NORTHEAST GridBagConstraints.NORTHWEST GridBagConstraints.SOUTHEAST GridBagConstraints.SOUTHWEST
int gridx	Specifies the X coordinate of the cell to which the component will be added.
int gridy	Specifies the Y coordinate of the cell to which the component will be added.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
double weightx	Specifies a weight value that determines the horizontal

	spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.  Button 1 Button 5 Button 4 ipadx = 0 ipady = 0 (original size) ipadx = 100 ipady = 0 (original size + 100 px width-wise) ipadx = 0 ipady = 100 (original size + 100 px height-wise)
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are GridBagConstraints.NONE (the default) GridBagConstraints.HORIZONTAL GridBagConstraints.VERTICAL GridBagConstraints.BOTH.
Insets insets	Small amount of space between the container that holds your components and the window that contains it. Default insets are all zero. Ex. Insets i=new Insets(5,10,20,15); 

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="GridBagDemo" width=200 height=100>
</applet>
*/
public class GridBagDemo extends Applet
{

        Button b1,b2,b3,b4,b5,b6,b7,b8 ;

        public void init() {
                GridBagLayout gbag = new GridBagLayout();
                GridBagConstraints gbc = new GridBagConstraints();
                setLayout(gbag);

                // Define check boxes.
                b1=new Button("Button 1");
                b2=new Button("Button 2");
                b3=new Button("Button 3");
                b4=new Button("Button 4");
                b5=new Button("Button 5");
                b6=new Button("Button 6");
                b7=new Button("Button 7");
                b8=new Button("Button 8");

                 gbc.gridx=0;
                gbc.gridy=0;
                gbag.setConstraints(b1,gbc);
                gbc.gridx=1;
                gbc.gridy=0;
                gbag.setConstraints(b2,gbc);
                gbc.gridx=2;
                gbc.gridy=0;
                gbag.setConstraints(b3,gbc);
                gbc.gridx=0;
                gbc.gridy=1;
                gbag.setConstraints(b4,gbc);
                gbc.gridx=1;
                gbc.gridy=1;
                gbc.gridwidth=2;
                gbc.gridheight=2;
                gbc.ipady=25;
                gbc.ipadx=20;
                gbc.fill=GridBagConstraints.BOTH;
                gbag.setConstraints(b5,gbc);
                gbc.gridx=0;
                gbc.gridy=2;
                gbc.anchor=GridBagConstraints.WEST;
                gbc.ipady= 0;
                gbc.ipadx= 0;
                gbc.fill=GridBagConstraints.NONE;
                gbag.setConstraints(b7,gbc);
                add(b1);
                add(b2);
                add(b3);
                add(b4);
                add(b5);
                add(b7);

        }
}
```



Applet started.