# Event Handling

## The Delegation Event Model:

The *delegation event model*, which defines standard and consistent mechanisms to generate and process events. 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 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.

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.

The following sections define events and describe the roles of sources and listeners.

## Events:

In the delegation model, 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 graphical user interface. 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. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface.For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

## Event Sources:

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. 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 add*Type*Listener(*Type*Listener *el*)**

Here, *Type* is the name of the event and *el* is a reference to the event listener.

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 remove*Type*Listener(*Type*Listener *el*)**

Here, *Type* is the name of the event and *el* is a reference to the event listener.

## Event Listeners:

A *listener* is an object that is notified when an event occurs. It has two major requirements.First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, 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**.

## Event Classes:

| Event Class | Description |
|---|---|
| ActionEvent | Generated when a button is pressed, a list item is double-clicked, or a menu item is selected. |
| AdjustmentEvent | Generated when a scroll bar is manipulated. |
| ComponentEvent | Generated when a component is hidden, moved, resized, or becomes visible. |
| ContainerEvent | Generated when a component is added to or removed from a container. |
| FocusEvent | Generated when a component gains or loses keyboard focus. |
| InputEvent | Abstract super class for all component input event classes. |
| ItemEvent | Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected. |
| KeyEvent | Generated when input is received from the keyboard. |
| MouseEvent | Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component. |
| MouseWheelEvent | Generated when the mouse wheel is moved. (Added by Java 2, version 1.4) |
| TextEvent | Generated when the value of a text area or text field is changed. |
| WindowEvent | Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**Table 20-1.** *Main Event Classes in* java.awt.event (continued)

## 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**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

**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*)**

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred. The third constructor was added by Java 2, version 1.4.
You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

**String getActionCommand( )**

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

## The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this tutorial.) There are two types of item events, which are identified by the following integer constants:

DESELECTED The user deselected an item.
SELECTED The user selected an item.

In addition, **ItemEvent** defines one integer constant, **ITEM_STATE_CHANGED**,that signifies a change of state.
**ItemEvent** has this constructor:

**ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)**

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.The **getItem( )** method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

**Object getItem( )**

The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

**ItemSelectable getItemSelectable( )**

Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.The **getStateChange( )** method returns the state change (i.e., **SELECTED** or**DESELECTED**) for the event. It is shown here: **int getStateChange( )**

## 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 key presses result in characters. For example, pressing the SHIFT key 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. Here are some others:

**VK_ENTER VK_ESCAPE VK_CANCEL VK_UP**
**VK_DOWN VK_LEFT VK_RIGHT VK_PAGE_DOWN**
**VK_PAGE_UP VK_SHIFT VK_ALT VK_CONTROL**

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt. **KeyEvent** is a subclass of **InputEvent**. Here are two of its constructors:

**KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*)**
**KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)**

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when this key event occurred.The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**. The **KeyEvent** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

**char getKeyChar( )**
**int getKeyCode( )**

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

## 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_PRESSED The mouse was pressed.**
**MOUSE_RELEASED The mouse was released.**
**MOUSE_WHEEL The mouse wheel was moved (Java 2, v1.4).**

**MouseEvent** is a subclass of **InputEvent**. Here is one of its constructors

.

**MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,int *x*, int *y*, int *clicks*, boolean *triggersPopup*)**

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. Java 2, version 1.4 adds a second constructor which also allows the button that caused the event to be specified. The most commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse 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 one constructor for this class is shown here:

**TextEvent(Object *src*, int *type*)**

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the **TextEvent** class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

## The WindowEvent Class:

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

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 gained input focus.
WINDOW_ICONIFIED The window was iconified.
WINDOW_LOST_FOCUS The window lost input focus.
WINDOW_OPENED The window was opened.
WINDOW_STATE_CHANGED The state of the window changed.

**WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors.The first is

**WindowEvent(Window *src*, int *type*)**

Here, *src* is a reference to the object that generated this event. The type of the event is *type.*
Java 2, version 1.4 adds the next three constructors.
WindowEvent(Window *src*, int *type*, Window *other*)
WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Here, *other* specifies the opposite window when a focus event occurs. The *fromState* specifies the prior state of the window and *toState* specifies the new state that the window will have when a window state change occurs.

## Sources of Events:

some of the user interface components that can generate the events are listed below

| Event Source | Description |
|---|---|
| Button | Generates action events when the button is pressed. |
| Checkbox | 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; generates item events when an item is selected or deselected. |
| Menu Item | Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected. |
| Scrollbar | 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. |

**Table 20-2.** *Event Source Examples*

## Event Listener Interfaces:

The delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

| Interface | Description |
|---|---|
| ActionListener | Defines one method to receive action events. |
| AdjustmentListener | Defines one method to receive adjustment events. |
| ComponentListener | Defines four methods to recognize when a component is hidden, moved, resized, or shown. |
| ContainerListener | Defines two methods to recognize when a component is added to or removed from a container. |
| FocusListener | Defines two methods to recognize when a component gains or loses keyboard focus. |
| ItemListener | Defines one method to recognize when the state of an item changes. |
| KeyListener | Defines three methods to recognize when a key is pressed, released, or typed. |
| MouseListener | Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released. |
| MouseMotionListener | Defines two methods to recognize when the mouse is dragged or moved. |
| MouseWheelListener | Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4) |
| TextListener | Defines one method to recognize when a text value changes. |
| WindowFocusListener | Defines two methods to recognize when a window gains or loses input focus. (Added by Java 2, version 1.4) |
| WindowListener | Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**Table 20-3.** *Commonly Used Event Listener Interfaces*

## The ActionListener Interface:

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

**void actionPerformed(ActionEvent *ae*)**

## The AdjustmentListener Interface:

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

**void adjustmentValueChanged(AdjustmentEvent *ae*)**

## The ItemListener Interface:

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

## The KeyListener Interface:

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released. The general forms of these methods are shown here:

**void keyPressed(KeyEvent *ke*)**
**void keyReleased(KeyEvent *ke*)**
**void keyTyped(KeyEvent *ke*)**

## The MouseListener Interface:

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.The general forms of these methods are shown here:

**void mouseClicked(MouseEvent *me*)**
**void mouseEntered(MouseEvent *me*)**
**void mouseExited(MouseEvent *me*)**
**void mousePressed(MouseEvent *me*)**
**void mouseReleased(MouseEvent *me*)**

## The MouseMotionListener Interface:

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

**void mouseDragged(MouseEvent *me*)**
**void mouseMoved(MouseEvent *me*)**

## The WindowListener Interface:

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified,the **windowDeiconified( )** method is called. When a window is opened or closed,the

**windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

**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*)**

## The TextListener Interface:

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

**void textChanged(TextEvent *te*)**

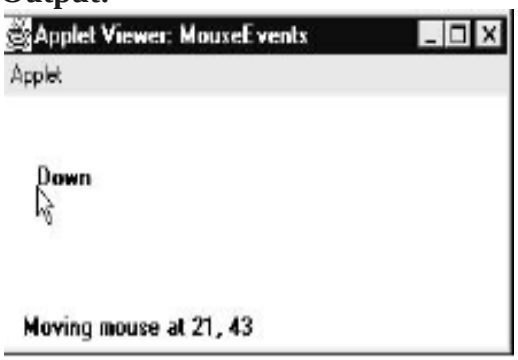
## Handling Mouse Events:

```
// 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);
  }
}//end
```

**Output:**




**Explanation:**

Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events.This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets. Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here:

**void addMouseListener(MouseListener *ml*)**
**void addMouseMotionListener(MouseMotionListener *mml*)**

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both.The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

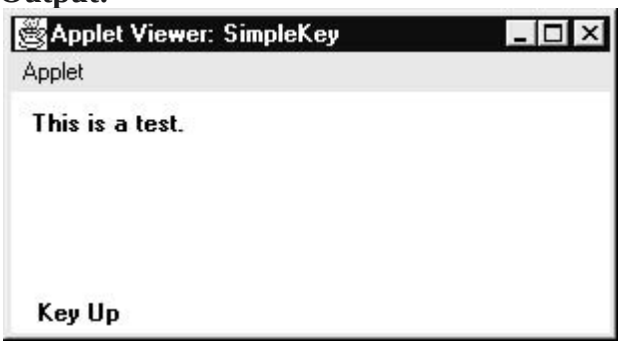
## Handling Keyboard Events:

Type of event generated from keyboard is **KeyEvent.** Listerner to be registered to receive notification is **addKeyListener(ref).**There is one other requirement that your program must meet before it can process keyboard events: it must request input focus. To do this, call requestFocus( ), which is defined by Component. To process the KeyEvent(i.e handle the

event )the interface to be implemented is **KeyListener .**This interface defines three methods. The keyPressed( ) and keyReleased( ) methods are invoked when a key is pressed and released, respectively. The keyTyped( ) method is invoked when a character has been entered.
For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released.

```
// Demonstrate the key event handlers.
 import java.awt.*;
 import java.awt.event.*;
 import java.applet.*;
 /*
  <applet code="SimpleKey" width=300 height=100>
  </applet>
 */
 public class SimpleKey extends Applet
 implements KeyListener
 {
   String msg = "";
   int X = 10, Y = 20; // output coordinates
   public void init()
   {
     addKeyListener(this);
     requestFocus(); // request input focus
   }
   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, X, Y);
   }
}
```

**Output:**




## 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. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

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

**Table 20-4.** *Commonly Used Listener Interfaces Implemented by Adapter Classes*

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**. The signatures of these empty methods are exactly as defined in the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and implement **mouseDragged( )**.

```
// Demonstrate an adapter class.
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));
addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus("Mouse clicked");
}
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
adapterDemo.showStatus("Mouse dragged");
}
}
```

# AWT(Abstract Window Toolkit)

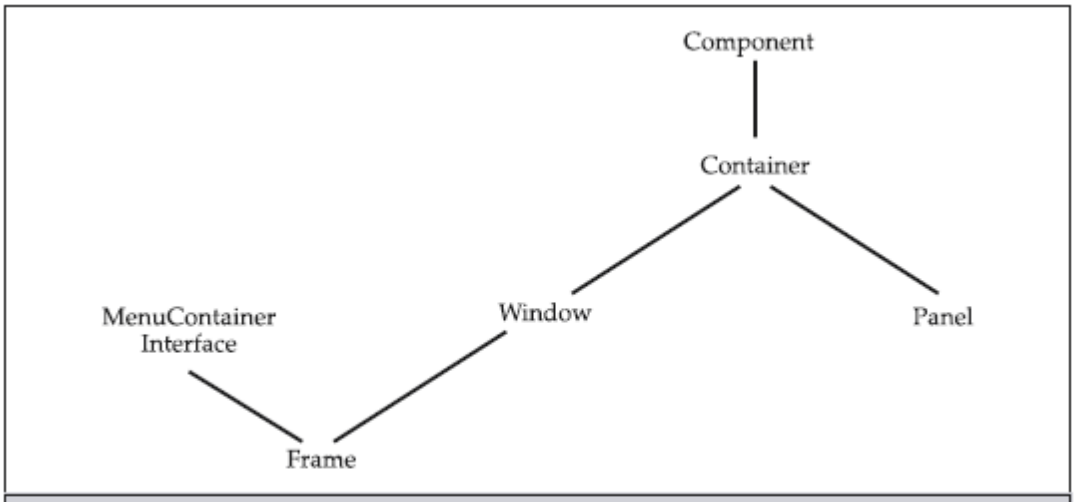



**Figure 21-1.** *The class hierarchy for* Panel *and* Frame

## Component:

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**.

It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

## Container:

The **Container** class is a subclass of **Component**. Other **Container** objects can be stored inside of a **Container**. This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

## Panel:

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. **Panel** is the superclass for **Applet**. A **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**).

## Window:

The **Window** class creates a top-level window. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**.

## Frame:

**Frame** encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners.

## Canvas:

Although it is not part of the hierarchy for applet or frame windows. **Canvas** encapsulates a blank window upon which you can draw.

# Using AWT Controls,Layout Managers,and Menus

*Controls* are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

## 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 **add( )** method has several forms. The following form is the one that is used for the first part of this chapter:

**Component add(Component *compObj*)**

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

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**.It has this general form:

**void remove(Component *obj*)**

Here, *obj* is a reference to the control you want to remove.

# Labels:

The easiest control to use is a 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. **Label** defines the following constructors:

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

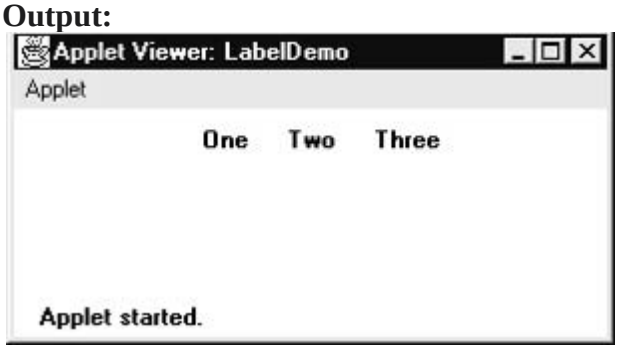
The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one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

You can set or change the text in a label by using the **setText( )** method. You can obtain the current label by calling **getText( )**. These methods are shown here:

**void setText(String *str*)**
**String getText( )**

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet
{
     public void init()
    {
          Label one = new Label("One");
          Label two = new Label("Two");
          Label three = new Label("Three");
          // add labels to applet window
         add(one);
         add(two);
         add(three);
   }
}
```

**Output:**



## Using Buttons

The most widely used control is the push button. 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( )
Button(String *str*)

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.

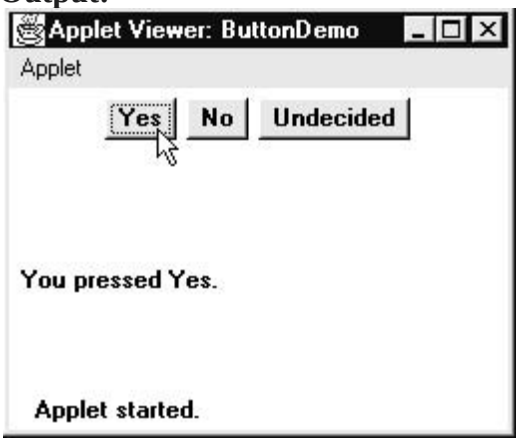
## Handling Buttons:

Each time a button is pressed,an **ActionEvent** is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed( )** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method.

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener
{
  String msg = "";
  Button yes, no, maybe;
  public void init()
{
   yes = new Button("Yes");
   no = new Button("No");
   maybe = new Button("Undecided");
   add(yes);
   add(no);
   add(maybe);
   yes.addActionListener(this);
   no.addActionListener(this);
   maybe.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
```

```
   String str = ae.getActionCommand();
    if(str.equals("Yes"))
   {
     msg = "You pressed Yes.";
  }
  else if(str.equals("No"))
 {
    msg = "You pressed No.";
  }
  else
 {
     msg = "You pressed Undecided.";
  }
      repaint();
}
   public void paint(Graphics g)
{
    g.drawString(msg, 6, 100);
  }
}
```

**Output:**




## 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 presents.

**Checkbox** supports these constructors:
Checkbox( )
Checkbox(String *str*)
Checkbox(String *str*, boolean *on*)
Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*)
Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*)

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**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. You can obtain the current label associated with a check box by calling **getLabel( )**. To set the label, call **setLabel( )**. These methods are as follows:

```
boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)
```

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

## Handling Check Boxes:

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener
{
  String msg = "";
  Checkbox Win98, winNT, solaris, mac;
  public void init()
  {
    Win98 = new Checkbox("Windows 98/XP", null, true);
    winNT = new Checkbox("Windows NT/2000");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");
    add(Win98);
    add(winNT);
    add(solaris);
    add(mac);
    Win98.addItemListener(this);
    winNT.addItemListener(this);
    solaris.addItemListener(this);
```

```
    mac.addItemListener(this);
  }
  public void itemStateChanged(ItemEvent ie)
  {
    repaint();
  }

  public void paint(Graphics g)
  {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows 98/XP: " + Win98.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows NT/2000: " + winNT.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " MacOS: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

**Output:**


## 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.*

Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.You can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. You can set a check box by calling **setSelectedCheckbox( )**. These methods are as follows:

**Checkbox getSelectedCheckbox( )**

**void setSelectedCheckbox(Checkbox *which*)**

Here, *which* is the check box that you want to be selected.

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener
{
  String msg = "";
  Checkbox Win98, winNT, solaris, mac;
  CheckboxGroup cbg;
  public void init()
  {
    setBackground(Color.gray);
    cbg = new CheckboxGroup();
    Win98 = new Checkbox("Windows 98/XP", cbg, true);
    winNT = new Checkbox("Windows NT/2000", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("MacOS", cbg, false);
    add(Win98);
    add(winNT);
    add(solaris);
    add(mac);
    Win98.addItemListener(this);
    winNT.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
 }
 public void itemStateChanged(ItemEvent ie)
 {
  repaint();
 }

 public void paint(Graphics g)
 {
  msg = "Current selection: ";
  msg += cbg.getSelectedCheckbox().getLabel();
  g.drawString(msg, 6, 100);
 }
}
```

**Output:**

# Choice Controls:

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. **Choice** only defines 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*)**

Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add( )** occur.

To determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

**String getSelectedItem( )**
**int getSelectedIndex( )**

The **getSelectedItem( )** method returns a string containing the name of the item.**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. To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item using the **select( )** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

**int getItemCount( )**
**void select(int *index*)**
**void select(String *name*)**

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

**String getItem(int *index*)**

Here, *index* specifies the index of the desired item.

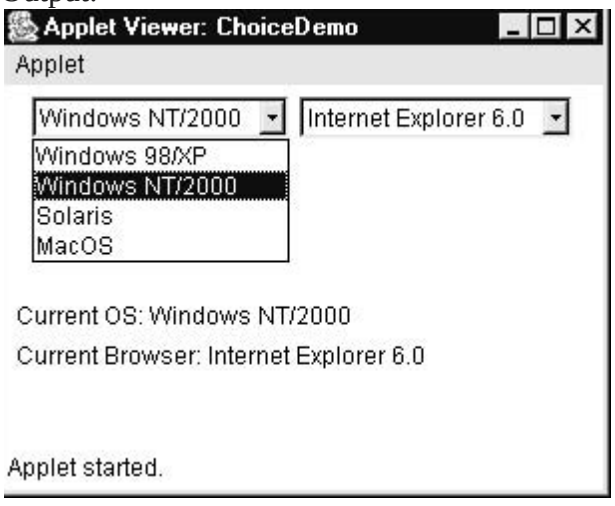
## Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method.

```
// Demonstrate Choice lists.
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 os, browser;
  String msg = "";
  public void init()
  {
    os = new Choice();
    browser = new Choice();
    // add items to os list
```

```
    os.add("Windows 98/XP");
    os.add("Windows NT/2000");
    os.add("Solaris");
    os.add("MacOS");
   // add items to browser list
   browser.add("Netscape 3.x");
   browser.add("Netscape 4.x");
   browser.add("Netscape 5.x");
   browser.add("Netscape 6.x");
   browser.add("Internet Explorer 4.0");
   browser.add("Internet Explorer 5.0");
   browser.add("Internet Explorer 6.0");
   browser.add("Lynx 2.4");
   browser.select("Netscape 4.x");
   // add choice lists to window
   add(os);
   add(browser);
  // register to receive item events
  os.addItemListener(this);
  browser.addItemListener(this);
 }
 public void itemStateChanged(ItemEvent ie)
 {
  repaint();
 }
 // Display current selections.
 public void paint(Graphics g)
 {
  msg = "Current OS: ";
  msg += os.getSelectedItem();
  g.drawString(msg, 6, 120);
  msg = "Current Browser: ";
  msg += browser.getSelectedItem();
  g.drawString(msg, 6, 140);
 }
}
```

Output:



# Using Lists:

The **List** class provides a compact, multiple-choice, scrolling selection list. It can also be created to allow multiple selections. **List** provides these constructors:

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

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*)**

Here, *name* is the name of the item added to the list. This form adds items to the end of the list.

you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

**String getSelectedItem( )**
**int getSelectedIndex( )**

The **getSelectedItem( )** method returns a string containing the name of the item. **getSelectedIndex( )** returns the index of the item.

For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes( )**, shown here, to determine the current selections:

**String[ ] getSelectedItems( )**
**int[ ] getSelectedIndexes( )**

**getSelectedItems( )** returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods are shown here:

**void select(int *index*)**

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

**String getItem(int *index*)**

Here, *index* specifies the index of the desired item.

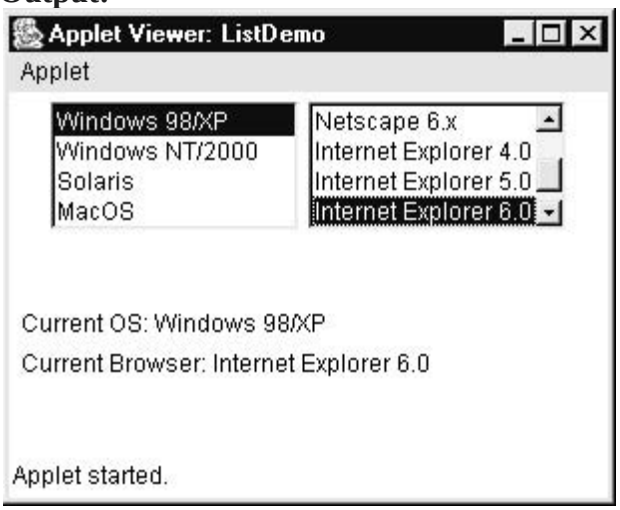
# Handling Lists:

To process list events, you will need to implement the **ActionListener** interface.Each time a **List** item is double-clicked, an **ActionEvent** object is generated. Its **getActionCommand( )** method can be used to retrieve the name of the newly selected item.

```
// Demonstrate Lists.
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 os, browser;
  String msg = "";
  public void init()
  {
    os = new List(4, true);
    browser = new List(4, false);
    // add items to os list
    os.add("Windows 98/XP");
    os.add("Windows NT/2000");
    os.add("Solaris");
    os.add("MacOS");
    // add items to browser list
    browser.add("Netscape 3.x");
    browser.add("Netscape 4.x");
    browser.add("Netscape 5.x");
    browser.add("Netscape 6.x");
    browser.add("Internet Explorer 4.0");
    browser.add("Internet Explorer 5.0");
    browser.add("Internet Explorer 6.0");
    browser.add("Lynx 2.4");
    browser.select(1);
    // add lists to window
    add(os);
    add(browser);
    // register to receive action events
    os.addActionListener(this);
    browser.addActionListener(this);
 }
 public void actionPerformed(ActionEvent ae)
 {
    repaint();
 }
 // Display current selections.
 public void paint(Graphics g)
 {
  int idx[];
  msg = "Current OS: ";
  idx = os.getSelectedIndexes();
  for(int i=0; i<idx.length; i++)
  {
    msg += os.getItem(idx[i]) + " ";
  }
  g.drawString(msg, 6, 120);
  msg = "Current Browser: ";
  msg += browser.getSelectedItem();
  g.drawString(msg, 6, 140);
 }
}
```

**Output:**




## Managing Scroll Bars:

*Scroll bars* are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. Scrollbar defines the following constructors:

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

style- Scrollbar.VERTICAL a vertical scroll. style- Scrollbar.HORIZONTAL, the scroll bar is horizontal. initialValue-initail value of scroll bar. thumbSize-height of thumb. min and max-minimum and maximum values for the scroll bar.

Methods in Scrollbar:

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues( )**, shown here, before it can be used:

**void setValues(int initialValue, int thumbSize, int min, int max)**

To obtain the current value of the scroll bar, call **getValue( )**. It returns the current setting. To set the current value, call **setValue( )**. These methods are as follows:

**int getValue( )**
**void setValue(int newValue)**

Here, *newValue* specifies the new value for the scroll bar.

## Handling Scroll Bars:

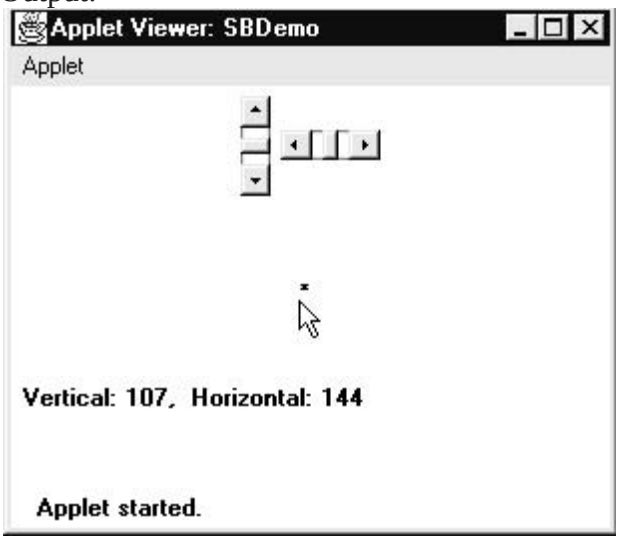
Type of event generated is AdjustmentEvent.To processor or Handle the AdjustmentEvent implement AdjustmentListener interface ,it has method ,ites general from is as follows.

**public void adjustmentValueChanged(AdjustmentEvent ae)**

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
```

```
</applet>
*/
public class SBDemo extends Applet implements AdjustmentListener, MouseMotionListener
{

  String msg = "";
  Scrollbar vertSB, horzSB;
  public void init()
  {
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));
    vertSB = new Scrollbar(Scrollbar.VERTICAL,0, 1, 0, height);
    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,0,1, 0, width);
    System.out.println("width:"+width);
    System.out.println("height:"+height);
    add(vertSB);
    add(horzSB);
    // register to receive adjustment events
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);
    addMouseMotionListener(this);
  }
  public void adjustmentValueChanged(AdjustmentEvent ae)
  {
     repaint();
  }
  // Update scroll bars to reflect mouse dragging.
  public void mouseDragged(MouseEvent me)
  {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
  }
  // Necessary for MouseMotionListener
 public void mouseMoved(MouseEvent me)
 {
 }
 // Display current value of scroll bars.
 public void paint(Graphics g)
 {
   msg = "Vertical: " + vertSB.getValue();
   msg += ", Horizontal: " + horzSB.getValue();
   g.drawString(msg, 6, 160);
   // show current mouse drag position
   g.drawString("*", horzSB.getValue(),
   vertSB.getValue());
 }
}//end
```

Output:




## Using a TextField:

The **TextField** class implements a single-line text-entry area, usually called an *edit control*. **TextField** defines the following constructors:

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

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.

To obtain the string currently contained in the text field, call **getText( )**. To set the text, call **setText( )**. These methods are as follows:

**String getText( )**
**void setText(String *str*)**

Here, *str* is the new string.

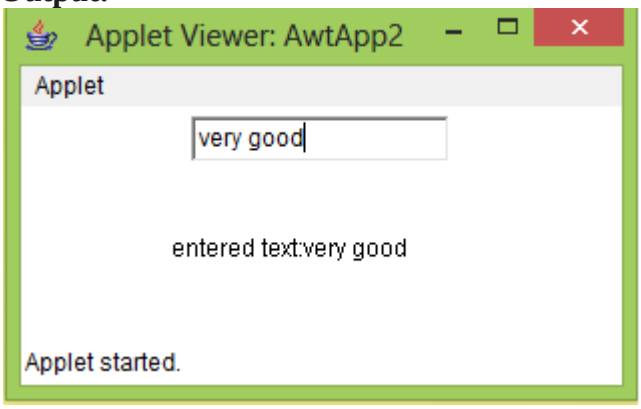
## Handling TextField:

The following example demonstrates you how to process or handle ActionEvent which is generated from TextField.Press ENTER key after text is entered in TextField,you can see output.

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

/*
  <applet code="AwtApp2" width="300" height="300">
  </applet>
*/

public class AwtApp2 extends Applet implements ActionListener
{

    TextField tf1;
```

```
    String s="";
    public void init()
    {
      tf1=new TextField(15);
       add(tf1);
       tf1.addActionListener(this);

    }

    public void actionPerformed(ActionEvent ae)
    {
      s="entered text:"+tf1.getText();
      repaint();
    }

    public void paint(Graphics g)
    {
      g.drawString(s,75,75);
    }
}
```

**Output:**




## Using a 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 is one of the constructor for **TextArea**:

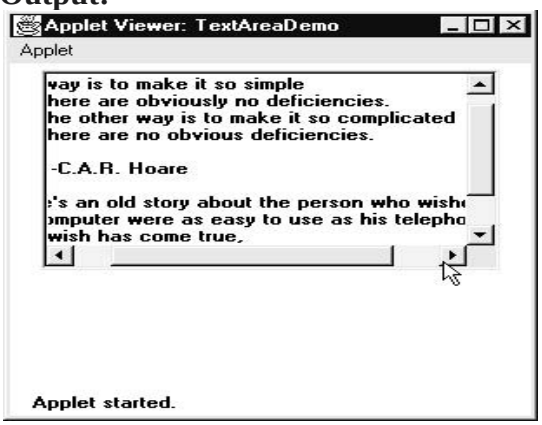
**TextArea(String *str*, int *numLines*, int *numChars*)**

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*.

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet
{
  public void init()
```

```
  {
    String val = "There are two ways of constructing " +
          "a software design.\n" +
          "One way is to make it so simple\n" +
          "that there are obviously no deficiencies.\n" +
          "And the other way is to make it so complicated\n" +
          "that there are no obvious deficiencies.\n\n" +
          " -C.A.R. Hoare\n\n" +
          "There's an old story about the person who wished\n" +
"his computer were as easy to use as his telephone.\n" +
"That wish has come true,\n" +
"since I no longer know how to use my telephone.\n\n" +
" -Bjarne Stroustrup, AT&T, (inventor of C++)";
   TextArea text = new TextArea(val, 10, 30);
   add(text);
  }
}
```

**Output:**




## Understanding Layout Managers:

All of the components that we have shown so far have been positioned by the default layout manager.A layout manager automatically arranges your controls within a window by using some type of algorithm. Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. The **setLayout( )** method has the following general form:

**void setLayout(LayoutManager *layoutObj*)**

Here, *layoutObj* is a reference to the desired layout manager.

you will need to determine the shape and position of each component manually,using the **setBounds( )** method defined by **Component**.

## FlowLayout

**FlowLayout** is the default layout manager. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom.

Here are the constructors for **FlowLayout**:

**FlowLayout( )**

**FlowLayout(int *how*)**

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

The first form creates the default layout, which centers 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**

These values specify left, center, and right alignment, respectively. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

```
// Use left-aligned flow layout.
 import java.awt.*;
 import java.awt.event.*;
 import java.applet.*;
 /*
  <applet code="FlowLayoutDemo" width=250 height=200>
  </applet>
 */
 public class FlowLayoutDemo extends Applet
 implements ItemListener
 {
   String msg = "";
   Checkbox Win98, winNT, solaris, mac;
   public void init()
   {
    // set left-aligned flow layout
    setLayout(new FlowLayout(FlowLayout.LEFT));
    Win98 = new Checkbox("Windows 98/XP", null, true);
    winNT = new Checkbox("Windows NT/2000");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");
    add(Win98);
    add(winNT);
    add(solaris);
    add(mac);
   // register to receive item events
   Win98.addItemListener(this);
   winNT.addItemListener(this);
   solaris.addItemListener(this);
   mac.addItemListener(this);
  }
 // Repaint when status of a check box changes.
 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 = " Windows 98/XP: " + Win98.getState();
  g.drawString(msg, 6, 100);
  msg = " Windows NT/2000: " + winNT.getState();
  g.drawString(msg, 6, 120);
  msg = " Solaris: " + solaris.getState();
  g.drawString(msg, 6, 140);
  msg = " Mac: " + mac.getState();
  g.drawString(msg, 6, 160);
 }
}
```

**Output:**


## BorderLayout:

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by **BorderLayout**:

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**:

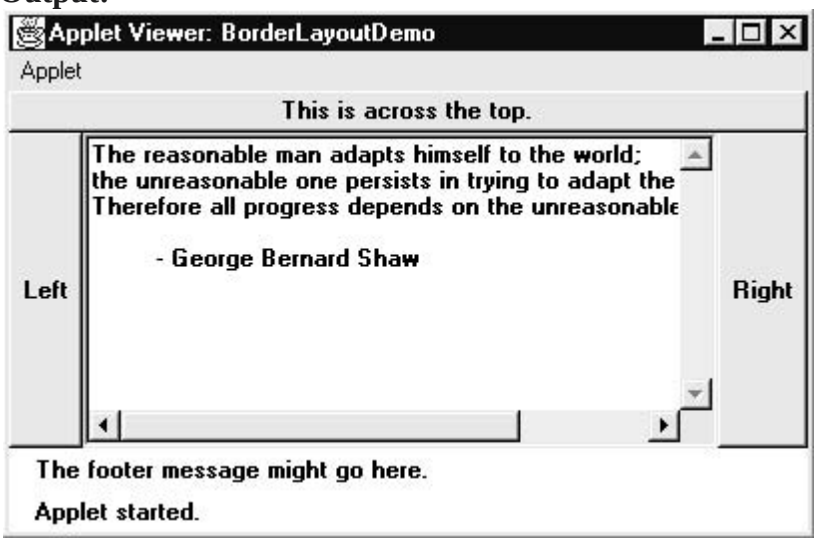
**void add(Component *compObj,* Object *region*);**

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

// Demonstrate BorderLayout.

```
import java.awt.*;
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("This is across the top."),BorderLayout.NORTH);
   add(new Label("The footer message might go here."),BorderLayout.SOUTH);
   add(new Button("Right"), BorderLayout.EAST);
   add(new Button("Left"), BorderLayout.WEST);
   String msg = "The reasonable man adapts " +"himself to the world;\n" +"the unreasonable
one persists in " +
   "trying to adapt the world to himself.\n" +    "Therefore all progress depends " +"on the
unreasonable man.\n\n" +
   " - George Bernard Shaw\n\n";
   add(new TextArea(msg), BorderLayout.CENTER);
  }
}
```

**Output:**




## Using Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the **getInsets( )** method that is defined by **Container**. This function returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed. The constructor for **Insets** is shown here:

**Insets(int *top*, int *left*, int *bottom*, int *right*)**

The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.

The **getInsets( )** method has this general form:

**Insets getInsets( )**

## GridLayout:

**GridLayout** lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

**GridLayout( )**

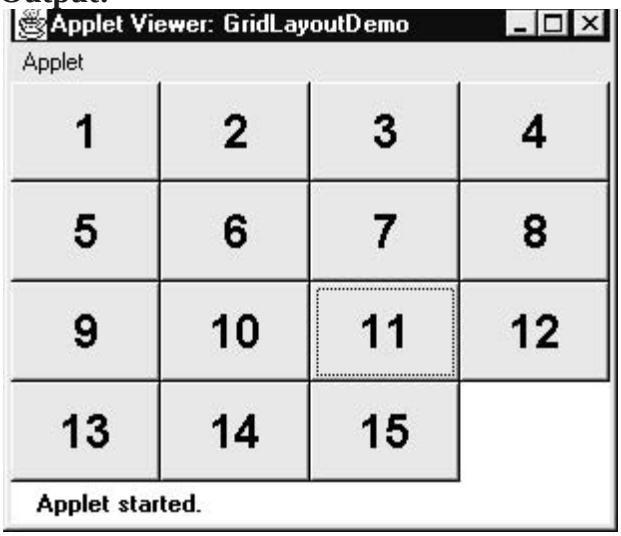
**GridLayout(int *numRows*, int *numColumns* )**

**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.

```
// Demonstrate GridLayout
 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));
     setFont(new Font("SansSerif", Font.BOLD, 24));
     for(int i = 0; i < 15; i++)
     {

           add(new Button("" + i));

      }
   }
}
```

**Output:**



## CardLayout

The **CardLayout** class is unique among the other layout managers in that it stores several different layouts. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.**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.
you will use this form of **add( )** when adding cards to a panel:

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

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**:

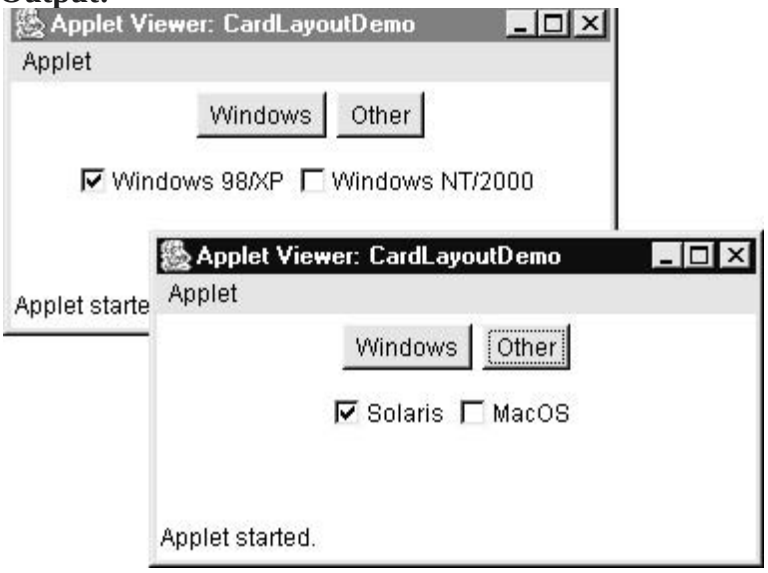
**void first(Container *deck*)**
**void last(Container *deck*)**
**void next(Container *deck*)**
**void previous(Container *deck*)**
**void show(Container *deck*, String *cardName*)**

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card.

```
// Demonstrate CardLayout.
 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, MouseListener
 {
   Checkbox Win98, winNT, solaris, mac;
   Panel osCards;
   CardLayout cardLO;
   Button Win, Other;
   public void init()
   {
     Win = new Button("Windows");
     Other = new Button("Other");
     add(Win);
     add(Other);
     cardLO = new CardLayout();
     osCards = new Panel();
     osCards.setLayout(cardLO); // set panel layout to card layout
     Win98 = new Checkbox("Windows 98/XP", null, true);
     winNT = new Checkbox("Windows NT/2000");
     solaris = new Checkbox("Solaris");
     mac = new Checkbox("MacOS");
```

```
    // add Windows check boxes to a panel
    Panel winPan = new Panel();
     winPan.add(Win98);
     winPan.add(winNT);
    // Add other OS check boxes to a panel
    Panel otherPan = new Panel();
     otherPan.add(solaris);
     otherPan.add(mac);
   // add panels to card deck panel
     osCards.add(winPan, "Windows");
     osCards.add(otherPan, "Other");
   // add cards to main applet panel
   add(osCards);
   // register to receive action events
   Win.addActionListener(this);
   Other.addActionListener(this);
   // register mouse events
   addMouseListener(this);
 }
  // Cycle through panels.
  public void mousePressed(MouseEvent me)
  {
    cardLO.next(osCards);
  }
  // Provide empty implementations for the other MouseListener methods.
  public void mouseClicked(MouseEvent me)
  {
  }
  public void mouseEntered(MouseEvent me)
  {
  }
  public void mouseExited(MouseEvent me)
  {
  }
  public void mouseReleased(MouseEvent me)
  {
  }
  public void actionPerformed(ActionEvent ae)
  {
    if(ae.getSource() == Win)
    {
      cardLO.show(osCards, "Windows");
    }
    else
    {
      cardLO.show(osCards, "Other");
    }
  }
}
```

**Output:**




## Working with Frame Windows:

The type of window you will most often create is derived from **Frame**. As mentioned, it creates a standard-style window.Here are two of **Frame**'s constructors:

**Frame( )**

**Frame(String *title*)**

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title*. The **setSize( )** method is used to set the dimensions of the window. Its signature is
shown here:

**void setSize(int *newWidth*, int *newHeight*)**

The size of the window is specified by *newWidth* and *newHeight*
After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here:

**void setVisible(boolean *visibleFlag*)**

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.
You can change the title in a frame window using **setTitle( )**, which has this general form:

**void setTitle(String *newTitle*)**

Here, *newTitle* is the new title for the window.

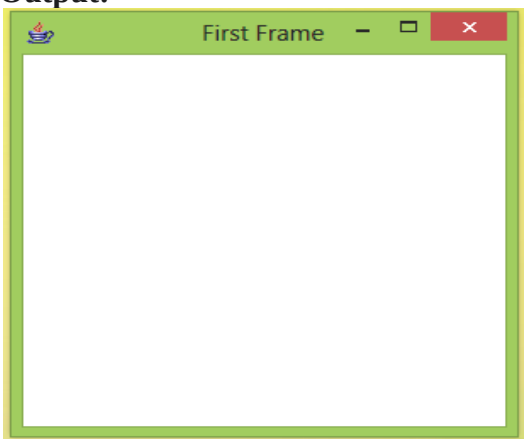
## Closing a Frame Window:

When using a frame window, your program must remove that window from the screen when it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must implement the **windowClosing( )** method of the **WindowListener** interface. Inside **windowClosing( )**, you must remove the window from the screen.

```
//Frame Demo
import java.awt.*;
import java.awt.event.*;
//you will subclass Frame to create a FrameWindow
 class ExFrame extends Frame
 {
    ExFrame()
    {
     setSize(300,300);
```

```
      setTitle("First Frame");
      setVisible(true);
      addWindowListener(new CloseWin(this));

    }
 }
//closing window
 class CloseWin extends WindowAdapter
 {
     ExFrame e;
     CloseWin(ExFrame ef)
     {
       e=ef;
     }
     public void   windowClosing(WindowEvent w )
     {
      e.setVisible(false);
     }
 }

class DemoFrame
{
  public static void main(String args[])
  {
      ExFrame f=new ExFrame();

  }
}
```

**Output:**




## Dialog Boxes:

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are primarily used to obtain user input. They are similar to frame windows, except that dialog boxes are always child windows of a top-level window. Also, dialog boxes don't have menu bars. Dialog boxes are of type **Dialog**. Two commonly used constructors are shown here:

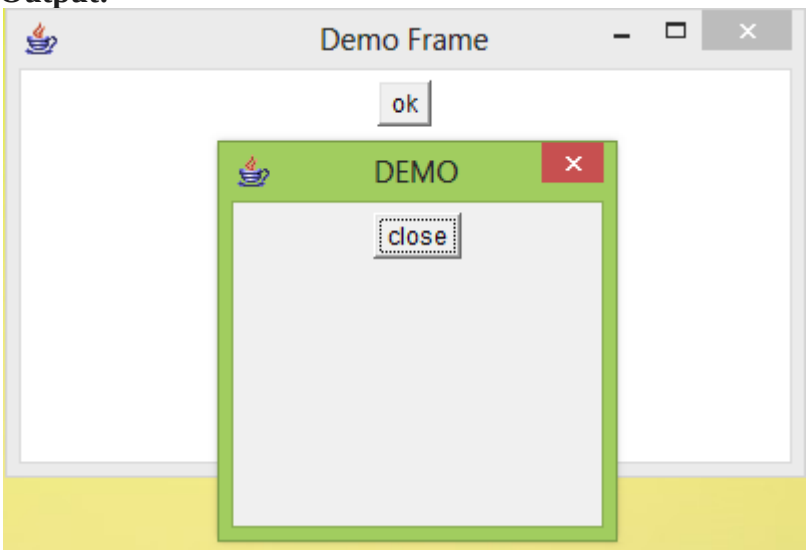
**Dialog(Frame *parentWindow*, boolean *mode*)**
**Dialog(Frame *parentWindow*, String *title*, boolean *mode*)**

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box is modal.Otherwise, it is modeless. The title of the dialog box can be passed in *title*. Generally, you will subclass **Dialog**, adding the functionality required by your application.

```
//DialogBox Demo
import java.awt.*;
import java.awt.event.*;

class Frame1 extends Frame  implements  ActionListener
{
   Button b1;

   Frame1()
   {

      setLayout(new FlowLayout());
     b1=new Button("ok");
     add(b1);
     b1.addActionListener(this);
     addWindowListener(new Close2(this));
   }

   public void actionPerformed(ActionEvent ae)
   {
      DialogExample d=new DialogExample(this);
      d.setSize(200,200);
      d.setVisible(true);
   }
}
//you will subclass Dialog to create dialogbox
class  DialogExample extends Dialog implements ActionListener
{
   Button b2;
   DialogExample(Frame1 f1)
   {
      super(f1,"DEMO",false);

      setLayout(new FlowLayout());
      b2=new Button("close");
      add(b2);
      b2.addActionListener(this);

   }
   public void actionPerformed(ActionEvent ae)
   {
     this.dispose();
   }

}
class Close2 extends WindowAdapter
{
```

```
  Frame1 f3;
     Close2(Frame1 f2)
     {
      f3=f2;
     }
  public void windowClosing(WindowEvent we)
  {
    f3.setVisible(false);
  }
}
class DialogEx
{

  public static void main(String args[])
  {
    Frame1  f=new Frame1();
    f.setSize(400,400);
    f.setVisible(true);
    f.setTitle("Demo Frame");
  }
}
```

**Output:**




## Menu Bars and Menus:

First create menu bar and then create menu ,next create menu item and then add menu item to menu ,menu to menu bar.To create a menu bar, first create an instance of **MenuBar**. This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

**Menu( )**
**Menu(String *optionName*)**

Here, *optionName* specifies the name of the menu selection.Individual menu items are of type **MenuItem**. It defines these constructors:

**MenuItem( )**

**MenuItem(String *itemName*)**

Here, *itemName* is the name shown in the menu.

```
//Menu Demo
import java.awt.*;
import java.awt.event.*;

class MenuFrame extends Frame implements ActionListener
{
   MenuBar mbr;
   Menu file,edit;
   MenuItem m1,m2,m3,m4,m5,m6;

   MenuFrame()
   {
     setSize(300,300);
     setVisible(true);
     setTitle("Menu Frame");
     mbr=new MenuBar();
     setMenuBar(mbr);

     file=new Menu("File");
     edit=new Menu("Edit");
     m1=new MenuItem("New");
     m2=new MenuItem("Save");
     m3=new MenuItem("Close");
     m4=new MenuItem("Copy");
     m5=new MenuItem("Paste");
     m6=new MenuItem("Select All");

     file.add(m1);
     file.add(m2);
     file.add(m3);
     edit.add(m4);
     edit.add(m5);
     edit.add(m6);

     mbr.add(file);
     mbr.add(edit);
      CloseWin2 w=new CloseWin2(this);
     addWindowListener(w);
     m3.addActionListener(w);
     m1.addActionListener(this);

   }
   public void actionPerformed(ActionEvent ae)
   {
     ExDialog ed=new ExDialog(this);
      ed.setSize(200,200);
      ed.setVisible(true);
   }
```

```
}

class ExDialog extends Dialog implements ActionListener
{
   Button b;
   ExDialog(MenuFrame mm)
   {
     super(mm,"Demo Dialog",false);

     setLayout(new FlowLayout());
     add(b=new Button("cancel"));
     b.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae)
   {
     this.dispose();
   }
}
class CloseWin2 extends WindowAdapter implements ActionListener
{
   MenuFrame f;
   CloseWin2(MenuFrame mf)
   {
     f=mf;
   }

   public void windowClosing(WindowEvent we)
   {
     f.setVisible(false);
   }
   public void actionPerformed(ActionEvent ae)
   {
     f.setVisible(false);
   }
}

class DemoMenu
{
  public static void main(String args[])
  {
    MenuFrame m=new MenuFrame();
  }

}
```

**Output:**