UNIT – VII

Event Handling : Events, Event sources, Event classes, Event Listeners, Delegation event model, handling mouse and keyboard events, Adapter classes.
The AWT class hierarchy, user interface components- labels, button, canvas, scrollbars, text components, check box, check box groups, choices, lists panels – scrollpane, dialogs, menubar, graphics, layout manager – layout manager types – boarder, grid, flow, card and grib bag.

EVENT HANDLING

Event handling is at the core of successful applet programming. Most events to which the applet will respond are generated by the user. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the java.awt.event package.

The Delegation Event Model

- ✓ The modern approach to handling events is based on 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. 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

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.
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.

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)

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 .
For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved.

EVENT CLASSES

The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is EventObject , which is in java.util .It is the superclass for all events.

Its one constructor is shown here:

EventObject(Object src)

EventObject contains two methods: getSource( ) and toString( ) .

The getSource( ) method returns the source of the event.
Object getSource( )

toString( ) returns the string equivalent of the event.

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

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; so 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.

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 )
```

The AdjustmentEvent Class

An AdjustmentEvent is generated by a scroll bar. There are five types of adjustment events. The AdjustmentEvent class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

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.
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. The constants and their meanings are shown here:

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

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 int constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED . They indicate that a component has been added to or removed from the container. ContainerEvent is a subclass of ComponentEvent and has this constructor:

ContainerEvent(Component src , int type , Component comp)

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 . FocusEvent is a subclass of ComponentEvent and has these constructors:

FocusEvent(Component src , int type)
FocusEvent(Component src , int type , boolean temporaryFlag)

The InputEvent Class

The abstract class InputEvent is a subclass of ComponentEvent 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 BUTTON2_MASK META_MASK
ALT_GRAPH_MASK BUTTON3_MASK SHIFT_MASK
BUTTON1_MASK CTRL_MASK

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, Java 2, version 1.4 added the following extended modifier values.

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 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 book.) 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.

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 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).

MouseWheelEvent defines these two integer constants.

WHEEL_BLOCK_SCROLL A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL A line-up or line-down scroll event occurred.

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)

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.
(Added by Java 2, version 1.4.)

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.

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 ComponentListener Interface

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)

The ContainerListener Interface

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)

The FocusListener Interface

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)

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 key, three events are generated in A 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 MouseWheelListener Interface

This interface defines the mouseWheelMoved( ) method that is invoked when the mouse wheel is moved. Its general form is shown here.

void mouseWheelMoved(MouseWheelEvent mwe)
MouseWheelListener was added by Java 2, version 1.4.

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)

The WindowFocusListener Interface

This interface defines two methods: windowGainedFocus( ) and windowLostFocus( ) . These are called when a window gains or losses input focus. Their general forms are shown here.

void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
WindowFocusListener was added by Java 2, version 1.4.

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)

Handling Mouse Events

To handle mouse events, we must implement the MouseListener and the MouseMotionListener interfaces.

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

Handling Keyboard Events

When a key is pressed, a KEY_PRESSED event is generated. This results in a call to the keyPressed( ) event handler. When the key is released, a KEY_RELEASED event is generated and the keyReleased( ) handler is executed. If a character is generated by the keystroke, then a KEY_TYPED event is sent and the keyTyped( ) handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events

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

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, 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( ) . The empty implementation of mouseMoved( ) would handle the mouse motion events for you.

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

```
// Demonstrate an adapter.
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");
}
}
```

ABSTRACT WINDOW TOOLKIT (AWT)

The Abstract Window Toolkit (AWT) support for applets. The AWT contains numerous classes and methods that allow you to create and manage windows.

AWT Classes

The AWT classes are contained in the java.awt package. It is one of Java's largest packages.

Class	**Description**
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
	Button Creates a push button control.
Canvas	A blank, semantics-free window.
CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of Component that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in width , and the height is stored in height .
Event	Encapsulates events.
EventQueue	Queues events.
FileDialog	Creates a window from which a file can be selected.
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.

FontMetrics	Encapsulates various information related to a font. This information helps you display text in a window.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
GraphicsDevice	Describes a graphics device such as a screen

Control Fundamentals

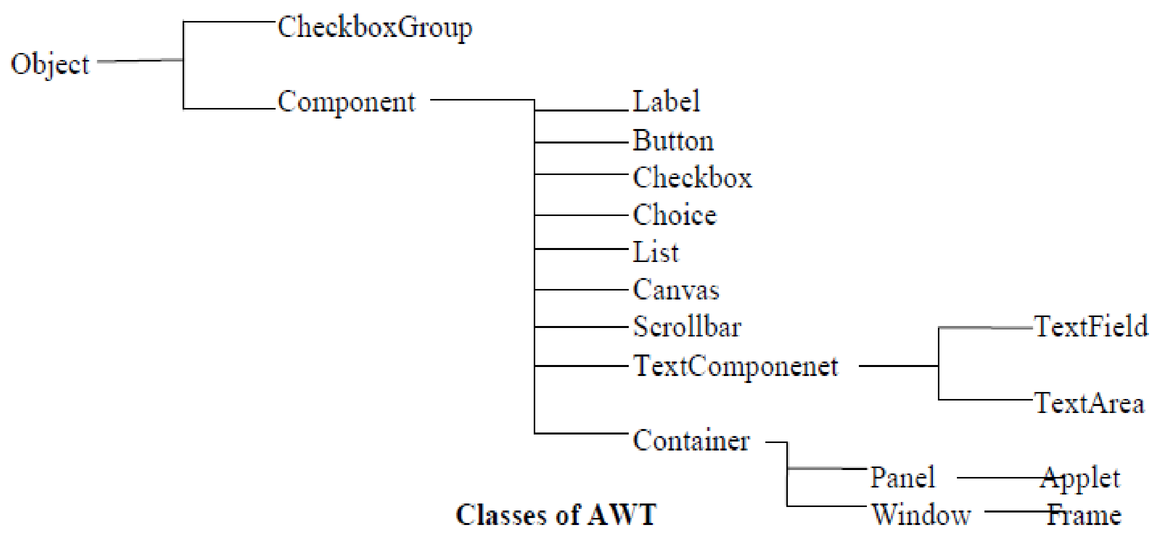
The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text editing

User interaction with the program is of two types:

- CUI (Character User Interface): In CUI user interacts with the application by typing characters or commands. In CUI user should remember the commands. It is not user friendly.
- GUI (Graphical User Interface): - In GUI user interacts with the application through graphics. GUI is user friendly. GUI makes application attractive. It is possible to simulate real object in GUI programs.

In java to write GUI programs we can use awt (Abstract Window Toolkit) package. java.awt is a package that provides a set of classes and interfaces to create GUI programs.



These controls are subclasses of Component .

Listeners and Listener Methods: Listeners are available for components. A Listener is an interface that listens to an event from a component. Listeners are available in java.awt.event package. The methods in the listener interface are to be implemented, when using that listener.

Component	Listener	Listener methods
Button	ActionListener	public void actionPerformed (ActionEvent e)
Checkbox	ItemListener	public void itemStateChanged (ItemEvent e)
CheckboxGroup	ItemListener	public void itemStateChanged (ItemEvent e)
TextField	ActionListener FocusListener	public void actionPerformed (ActionEvent e) public void focusGained (FocusEvent e) public void focusLost (FocusEvent e)
TextArea	ActionListener FocusListener	public void actionPerformed (ActionEvent e) public void focusGained (FocusEvent e) public void focusLost (FocusEvent e)
Choice	ActionListener ItemListener	public void actionPerformed (ActionEvent e) public void itemStateChanged (ItemEvent e)
List	ActionListener ItemListener	public void actionPerformed (ActionEvent e) public void itemStateChanged (ItemEvent e)
Scrollbar	AdjustmentListener MouseMotionListener	public void adjustmentValueChanged (AdjustmentEvent e) public void mouseDragged (MouseEvent e) public void mouseMoved (MouseEvent e)
Label	No listener is needed	

Labels

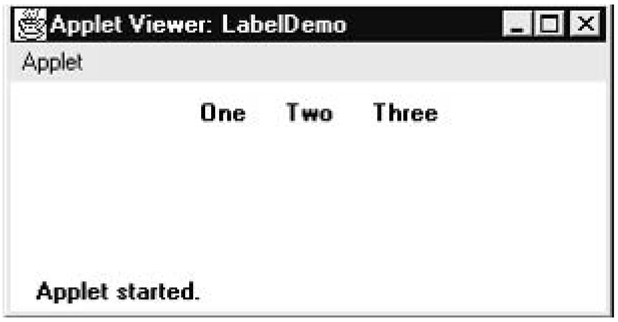
The easiest control to use is a label. A label is an object of type Label , and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user.

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 .

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



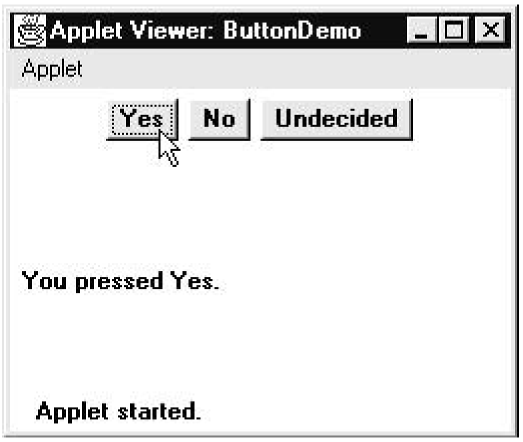
Buttons

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.

Button class is useful to create push buttons. A push button triggers a series of events.
· To create push button: Button b1 =new Button("label");
· To get the label of the button: String l = b1.getLabel();
· To set the label of the button: b1.setLabel("label");
· To get the label of the button clicked: String str = ae.getActionCommand();
where ae is object of ActionEvent

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



```
You pressed Yes.
Applet started.
```

Check Boxes

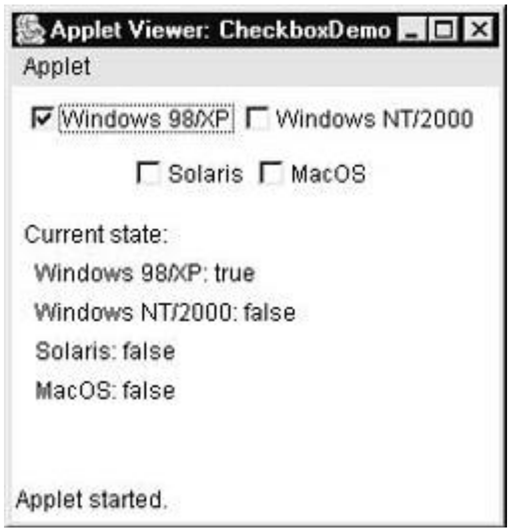
A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class.

Checkbox supports these constructors:
Checkbox( )

```
Checkbox(String  str )
Checkbox(String  str , boolean  on )
Checkbox(String  str , boolean  on , CheckboxGroup  cbGroup )
Checkbox(String  str , CheckboxGroup  cbGroup , boolean  on )
```

A Checkbox is a square shapped box which provides a set of options to the user.
· To create a Checkbox: Checkbox cb = new Checkbox ("label");
· To create a checked Checkbox: Checkbox cb = new Checkbox ("label", null, true);
· To get the state of a Checkbox: boolean b = cb.getState ();
· To set the state of a Checkbox: cb.setState (true);
· To get the label of a Checkbox: String s = cb.getLabel ();

```
// 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();
}
// 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 = "  MacOS: " + mac.getState();
g.drawString(msg, 6, 160);
}
}
```



```
Current state:
 Windows 98/XP: true
 Windows NT/2000: false
 Solaris: false
 MacOS: false
```

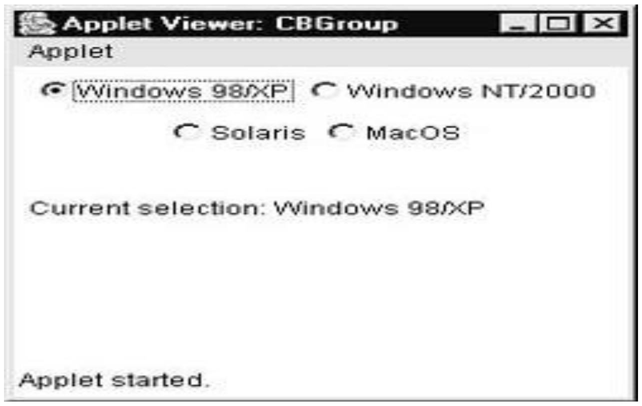
Applet started.

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.
A Radio button represents a round shaped button such that only one can be selected from a panel. Radio button can be created using CheckboxGroup class and Checkbox classes.
· To create a radio button: CheckboxGroup cbg = new CheckboxGroup ();
Checkbox cb = new Checkbox ("label", cbg, true);

· To know the selected checkbox: Checkbox cb = cbg.getSelectedCheckbox ();
·To know the selected checkbox label: String label = cbg.getSelectedCheckbox ().getLabel ();

```
// 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() {
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();
}

// 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: Windows 98/XP

Applet started.

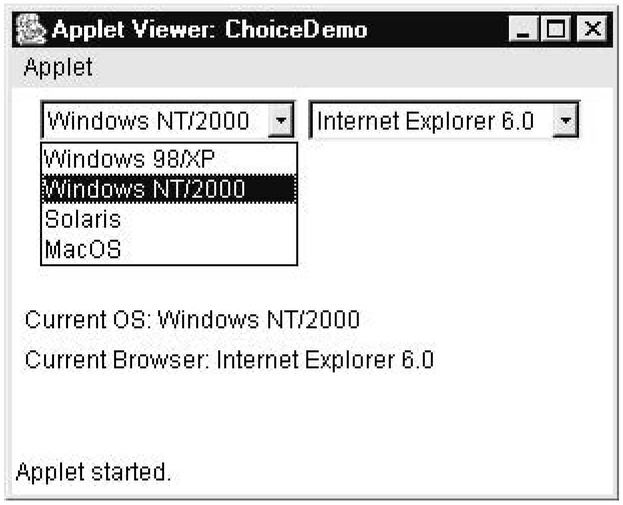
Choice Controls

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 menu is a popdown list of items. Only one item can be selected.

· To create a choice menu: Choice ch = new Choice();
· To add items to the choice menu: ch.add ("text");
· To know the name of the item selected from the choice menu:
String s = ch.getSelectedItem ();
· To know the index of the currently selected item: int i = ch.getSelectedIndex(); This method returns -1, if nothing is selected.

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



```
Current OS: Windows NT/2000
Current Browser: Internet Explorer 6.0
Applet started.
```

Lists

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

A List box is similar to a choice box, it allows the user to select multiple items.

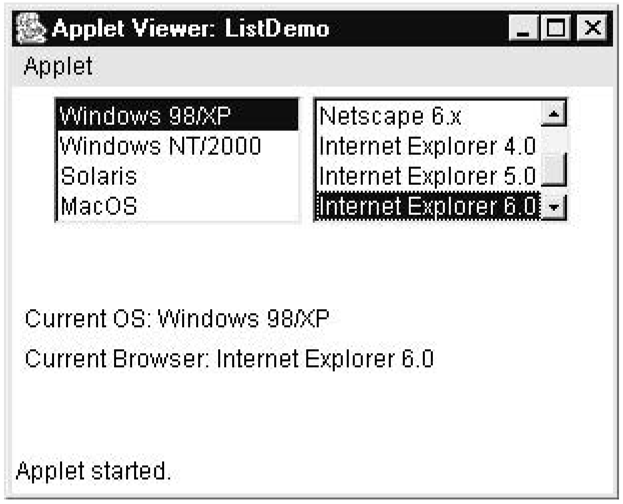
· To create a list box: List lst = new List();
(or)
List lst = new List (3, true);

This list box initially displays 3 items. The next parameter true represents that the user can select more than one item from the available items. If it is false, then the user can select only one item.

· To add items to the list box: lst.add("text");
· To get the selected items: String x[] = lst.getSelectedItems();
· To get the selected indexes: int x[] = lst.getSelectedIndexes ();

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



```
Current OS: Windows 98/XP
Current Browser: Internet Explorer 6.0
```

Applet started.

Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically.

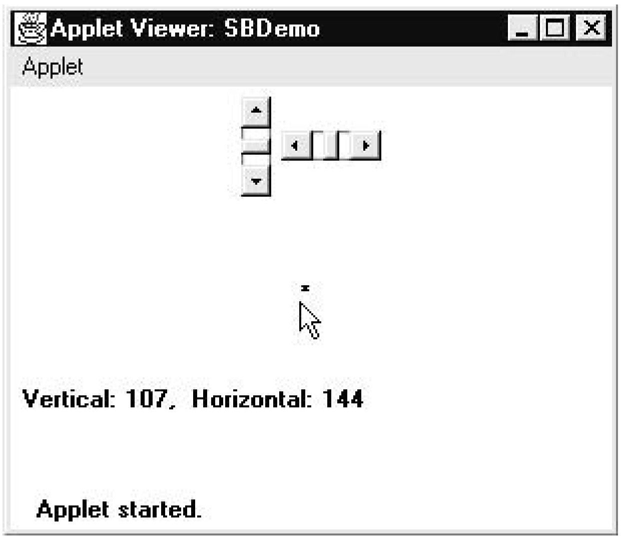
Scrollbar class is useful to create scrollbars that can be attached to a frame or text area. Scrollbars can be arranged vertically or horizontally.
· To create a scrollbar : Scrollbar sb = new Scrollbar (alignment, start, step, min, max);

alignment: Scrollbar.VERTICAL, Scrollbar.HORIZONTAL
start: starting value (e.g. 0)
step: step value (e.g. 30) // represents scrollbar length
min: minimum value (e.g. 0)
max: maximum value (e.g. 300)

· To know the location of a scrollbar: int n = sb.getValue ();
· To update scrollbar position to a new position: sb.setValue (int position);
· To get the maximum value of the scrollbar: int x = sb.getMaximum ();
· To get the minimum value of the scrollbar: int x = sb.getMinimum ();
· To get the alignment of the scrollbar: int x = getOrientation ();
This method return 0 if the scrollbar is aligned HORIZONTAL, 1 if aligned VERTICAL.

```
// 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);
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());
}
}
```



```
Vertical: 107, Horizontal: 144
```

Applet started.

TextField

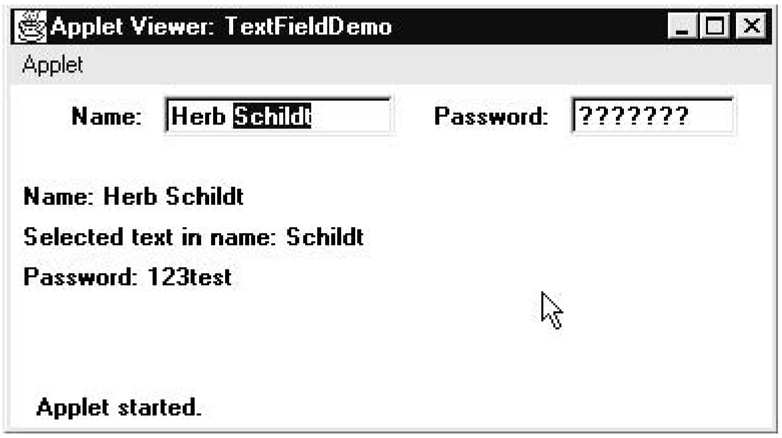
The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow eys, cut and paste keys, and mouse selections. TextField is a subclass of TextComponent . TextField defines the following constructors:

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

TextField allows a user to enter a single line of text.
· To create a TextField: TextField tf = new TextField(25);
(or)
TextField tf = new TextField ("defaulttext", 25);
· To get the text from a TextField: String s = tf.getText();
· To set the text into a TextField: tf.setText("text");
· To hide the text being typed into the TextField by a character: tf.setEchoChar('char');

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/

public class TextFieldDemo extends Applet
implements ActionListener {

TextField name, pass;

public void init() {
Label namep = new Label("Name: ", Label.RIGHT);
Label passp = new Label("Password: ", Label.RIGHT);
name = new TextField(12);
pass = new TextField(8);
pass.setEchoChar('?');

add(namep);
add(name);
add(passp);
add(pass);

// register to receive action events
name.addActionListener(this);
pass.addActionListener(this);
}

// User pressed Enter.
public void actionPerformed(ActionEvent ae) {
repaint();
}

public void paint(Graphics g) {
g.drawString("Name: " + name.getText(), 6, 60);
g.drawString("Selected text in name: "
+ name.getSelectedText(), 6, 80);
g.drawString("Password: " + pass.getText(), 6, 100);
}
}
```



```
Name: Herb Schildt        Password: ???????

Name: Herb Schildt
Selected text in name: Schildt
Password: 123test

Applet started.
```

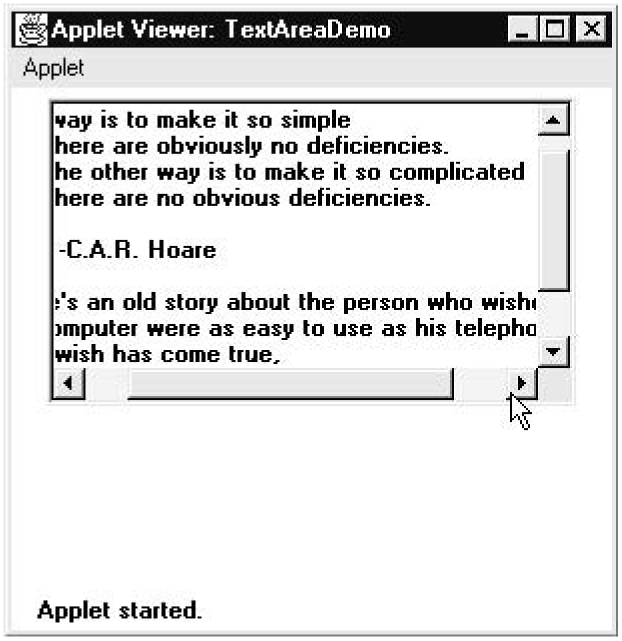
TextArea

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( )
TextArea(int numLines, int numChars )
TextArea(String str )
TextArea(String str , int numLines , int numChars )
TextArea(String str , int numLines , int numChars , int sBars )
```

TextArea supports getText () and setText ()

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



```
vay is to make it so simple
here are obviously no deficiencies.
he other way is to make it so complicated
here are no obvious deficiencies.

-C.A.R. Hoare

:'s an old story about the person who wishe
omputer were as easy to use as his telepho
wish has come true.
```

Applet started.

Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in Java by the following classes: MenuBar , Menu , and MenuItem . In general, a menu bar contains one or more Menu objects. Each Menu object contains a list of MenuItem objects. Each MenuItem object represents something that can be selected by the user. Since Menu is a subclass of MenuItem , a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type CheckboxMenuItem and will have a check mark next to them when they are selected.

Following are the constructors for Menu :

Menu( )
Menu(String optionName)
Menu(String optionName , boolean removable)
.
Individual menu items are of type MenuItem . It defines these constructors:

MenuItem( )
MenuItem(String itemName)
MenuItem(String itemName , MenuShortcut keyAccel)

Here, itemName is the name shown in the menu, and keyAccel is the menu shortcut for

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/

// Create a subclass of Frame
class MenuFrame extends Frame {
String msg = "";
CheckboxMenuItem debug, test;

MenuFrame(String title) {
super(title);

// create menu bar and add it to frame
MenuBar mbar = new MenuBar();
setMenuBar(mbar);

// create the menu items
Menu file = new Menu("File");
MenuItem item1, item2, item3, item4, item5;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(item4 = new MenuItem("-"));
file.add(item5 = new MenuItem("Quit..."));
mbar.add(file);

Menu edit = new Menu("Edit");
MenuItem item6, item7, item8, item9;
edit.add(item6 = new MenuItem("Cut"));
edit.add(item7 = new MenuItem("Copy"));
edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));
Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);
// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);

mbar.add(edit);

// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);

// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}

public void paint(Graphics g) {
g.drawString(msg, 10, 200);
if(debug.getState())
g.drawString("Debug is on.", 10, 220);
else
g.drawString("Debug is off.", 10, 220);

if(test.getState())
g.drawString("Testing is on.", 10, 240);
else
g.drawString("Testing is off.", 10, 240);
}
}
```

```
class MyWindowAdapter extends WindowAdapter {
MenuFrame menuFrame;
public MyWindowAdapter(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
public void windowClosing(WindowEvent we) {
menuFrame.setVisible(false);
}
}

class MyMenuHandler implements ActionListener, ItemListener {
MenuFrame menuFrame;
public MyMenuHandler(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
// Handle action events
public void actionPerformed(ActionEvent ae) {
String msg = "You selected ";
String arg = (String)ae.getActionCommand();
if(arg.equals("New..."))
msg += "New.";
else if(arg.equals("Open..."))
msg += "Open.";
else if(arg.equals("Close"))
msg += "Close.";
else if(arg.equals("Quit..."))
msg += "Quit.";
else if(arg.equals("Edit"))
msg += "Edit.";
else if(arg.equals("Cut"))
msg += "Cut.";
else if(arg.equals("Copy"))
msg += "Copy.";
else if(arg.equals("Paste"))
msg += "Paste.";
else if(arg.equals("First"))
msg += "First.";
else if(arg.equals("Second"))
msg += "Second.";
else if(arg.equals("Third"))
msg += "Third.";
else if(arg.equals("Debug"))
msg += "Debug.";
else if(arg.equals("Testing"))
msg += "Testing.";
menuFrame.msg = msg;
menuFrame.repaint();
}
// Handle item events
public void itemStateChanged(ItemEvent ie) {
menuFrame.repaint();
}
}

// Create frame window.
public class MenuDemo extends Applet {
Frame f;
public void init() {
f = new MenuFrame("Menu Demo");
int width = Integer.parseInt(getParameter("width"));
int height = Integer.parseInt(getParameter("height"));

setSize(new Dimension(width, height));

f.setSize(width, height);
f.setVisible(true);
}

public void start() {
f.setVisible(true);
}

public void stop() {
f.setVisible(false);
}
}
```

Dialog Boxes

We 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. In other respects, dialog boxes function like frame windows.
Dialog boxes may be modal or modeless. When a modal dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box. When a modeless dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible. 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)

```
// Demonstrate Dialog box.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="DialogDemo" width=250 height=250>
</applet>
*/

// Create a subclass of Dialog.
class SampleDialog extends Dialog implements ActionListener {
SampleDialog(Frame parent, String title) {
super(parent, title, false);
setLayout(new FlowLayout());
setSize(300, 200);

add(new Label("Press this button:"));
Button b;
add(b = new Button("Cancel"));
b.addActionListener(this);
}

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

public void paint(Graphics g) {
g.drawString("This is in the dialog box", 10, 70);
}
}

// Create a subclass of Frame.
class MenuFrame extends Frame {
String msg = "";
CheckboxMenuItem debug, test;

MenuFrame(String title) {
super(title);

// create menu bar and add it to frame
MenuBar mbar = new MenuBar();
setMenuBar(mbar);

// create the menu items
Menu file = new Menu("File");
```

```
MenuItem item1, item2, item3, item4;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(new MenuItem("-"));
file.add(item4 = new MenuItem("Quit..."));
mbar.add(file);

Menu edit = new Menu("Edit");
MenuItem item5, item6, item7;
edit.add(item5 = new MenuItem("Cut"));
edit.add(item6 = new MenuItem("Copy"));
edit.add(item7 = new MenuItem("Paste"));
edit.add(new MenuItem("-"));

Menu sub = new Menu("Special", true);
MenuItem item8, item9, item10;
sub.add(item8 = new MenuItem("First"));
sub.add(item9 = new MenuItem("Second"));
sub.add(item10 = new MenuItem("Third"));
edit.add(sub);

// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);

mbar.add(edit);

// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);
```

```
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString(msg, 10, 200);

if(debug.getState())
g.drawString("Debug is on.", 10, 220);
else
g.drawString("Debug is off.", 10, 220);

if(test.getState())
g.drawString("Testing is on.", 10, 240);
else
  g.drawString("Testing is off.", 10, 240);
}
}

class MyWindowAdapter extends WindowAdapter {
MenuFrame menuFrame;
public MyWindowAdapter(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
public void windowClosing(WindowEvent we) {
menuFrame.dispose();
}
}

class MyMenuHandler implements ActionListener, ItemListener {
MenuFrame menuFrame;
public MyMenuHandler(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
// Handle action events
public void actionPerformed(ActionEvent ae) {
String msg = "You selected ";
String arg = (String)ae.getActionCommand();
// Activate a dialog box when New is selected.
if(arg.equals("New...")) {
msg += "New.";
SampleDialog d = new
SampleDialog(menuFrame, "New Dialog Box");
d.setVisible(true);
}
```

```
// Try defining other dialog boxes for these options.
else if(arg.equals("Open..."))
msg += "Open.";
else if(arg.equals("Close"))
msg += "Close.";
else if(arg.equals("Quit..."))
msg += "Quit.";
else if(arg.equals("Edit"))
msg += "Edit.";
else if(arg.equals("Cut"))
msg += "Cut.";
else if(arg.equals("Copy"))
msg += "Copy.";
else if(arg.equals("Paste"))
msg += "Paste.";
else if(arg.equals("First"))
msg += "First.";
else if(arg.equals("Second"))
msg += "Second.";
else if(arg.equals("Third"))
msg += "Third.";
else if(arg.equals("Debug"))
msg += "Debug.";
else if(arg.equals("Testing"))
msg += "Testing.";
menuFrame.msg = msg;
menuFrame.repaint();
}
public void itemStateChanged(ItemEvent ie) {
menuFrame.repaint();
}
}

// Create frame window.
public class DialogDemo extends Applet {
Frame f;

public void init() {
f = new MenuFrame("Menu Demo");
int width = Integer.parseInt(getParameter("width"));
int height = Integer.parseInt(getParameter("height"));

setSize(width, height);

f.setSize(width, height);
f.setVisible(true);
```

```
}

public void start() {
f.setVisible(true);
}

public void stop() {
  f.setVisible(false);
}
}
```

Graphics

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window.

Graphics class and is obtained in two ways:
■ It is passed to an applet when one of its various methods, such as **paint( )** or **update( )**, is called.
■ It is returned by the **getGraphics( )** method of **Component**.

Drawing Lines

Lines are drawn by means of the **drawLine( )** method, shown here:
void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)
drawLine( ) displays a line in the current drawing color that begins at *startX*,*startY* and ends at *endX*,*endY*.

The following applet draws several lines:

```
// Draw lines
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
public void paint(Graphics g) {
g.drawLine(0, 0, 100, 100);
g.drawLine(0, 100, 100, 0);
g.drawLine(40, 25, 250, 180);
g.drawLine(75, 90, 400, 400);
g.drawLine(20, 150, 400, 40);
g.drawLine(5, 290, 80, 19);
}
}
```

Drawing Rectangles

The **drawRect( )** and **fillRect( )** methods display an outlined and filled rectangle, respectively. They are shown here:
void drawRect(int *top*, int *left*, int *width*, int *height*)
void fillRect(int *top*, int *left*, int *width*, int *height*)

The upper-left corner of the rectangle is at *top*,*left*. The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use **drawRoundRect( )** or **fillRoundRect( )**, both shown here:

void drawRoundRect(int *top*, int *left*, int *width*, int *height*,int *xDiam*, int *yDiam*)
void fillRoundRect(int *top*, int *left*, int *width*, int *height*, int *xDiam*, int *yDiam*)

```
// Draw rectangles
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics g) {
g.drawRect(10, 10, 60, 50);
g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15);
g.fillRoundRect(70, 90, 140, 100, 30, 40);
}
}
```

Drawing Ellipses and Circles

To draw an ellipse, use **drawOval( )**. To fill an ellipse, use **fillOval( )**. These methods are shown here:

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

```
// Draw Ellipses
import java.awt.*;
import java.applet.*;
/*
```

```
<applet code="Ellipses" width=300 height=200>
</applet>
*/
public class Ellipses extends Applet {
public void paint(Graphics g) {
g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
}
}
```

Drawing Arcs

Arcs can be drawn with **drawArc( )** and **fillArc( )**, shown here:
void drawArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*,int *sweepAngle*)
void fillArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*,int *sweepAngle*)

The arc is bounded by the rectangle whose upper-left corner is specified by *top*,*left* and whose width and height are specified by *width* and *height*. The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle*. Angles are specified in degrees. Zero degrees is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180.

The following applet draws several arcs:

```
// Draw Arcs
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcs" width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
public void paint(Graphics g) {
g.drawArc(10, 40, 70, 70, 0, 75);
g.fillArc(100, 40, 70, 70, 0, 75);
g.drawArc(10, 100, 70, 80, 0, 175);
g.fillArc(100, 100, 70, 90, 0, 270);
g.drawArc(200, 80, 80, 80, 0, 180);
}
}
```

Drawing Polygons

It is possible to draw arbitrarily shaped figures using **drawPolygon( )** and **fillPolygon( )**, shown here:
void drawPolygon(int *x*[ ], int *y*[ ], int *numPoints*)
void fillPolygon(int *x*[ ], int *y*[ ], int *numPoints*)

The polygon's endpoints are specified by the coordinate pairs contained within the *x* and *y* arrays. The number of points defined by *x* and *y* is specified by *numPoints*. There are alternative forms of these methods in which the polygon is specified by a **Polygon** object.

The following applet draws an hourglass shape:

```
// Draw Polygon
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/
public class HourGlass extends Applet {
public void paint(Graphics g) {
int xpoints[] = {30, 200, 30, 200, 30};
int ypoints[] = {30, 30, 200, 200, 30};
int num = 5;
g.drawPolygon(xpoints, ypoints, num); } }
```

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 the controls within a window by using some type of algorithm.

Layout Manager is an interface that arranges the components on the screen.
Layout Manager is implemented in the following classes:

FlowLayout
BorderLayout
CardLayout
GridLayout
GridBagLayout

FlowLayout

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

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

BorderLayout

BorderLayout is useful to arrange the components in the 4 borders of the frame
as well as in the center. The borders are specified as South, North, East, West and Center.

Here are the constructors defined by BorderLayout :
BorderLayout( )
BorderLayout(int horz , int vert)

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

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 .

The getInsets( ) method has this general form:
Insets getInsets( )
When overriding one of these methods, you must return a new Insets object that contains the inset spacing you desire.

```
// Demonstrate BorderLayout with insets.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/

public class InsetsDemo extends Applet {
public void init() {
// set background color so insets can be easily seen
setBackground(Color.cyan);

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);
}

 // add insets
public Insets getInsets() {
```

```
return new Insets(10, 10, 10, 10);
}
}
```

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 )
```

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

CardLayout

The CardLayout class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:
CardLayout( )
CardLayout(int horz , int vert)

```
// 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");
}
}
}
```