

Event Handling

Applets are event-driven programs. Thus, event handling is at the core of successful applet programming. Most events to which our applet will respond are generated by the user. These events are passed to our applet in a variety of ways, with the specific method depending upon the actual event. There are several types of events. 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

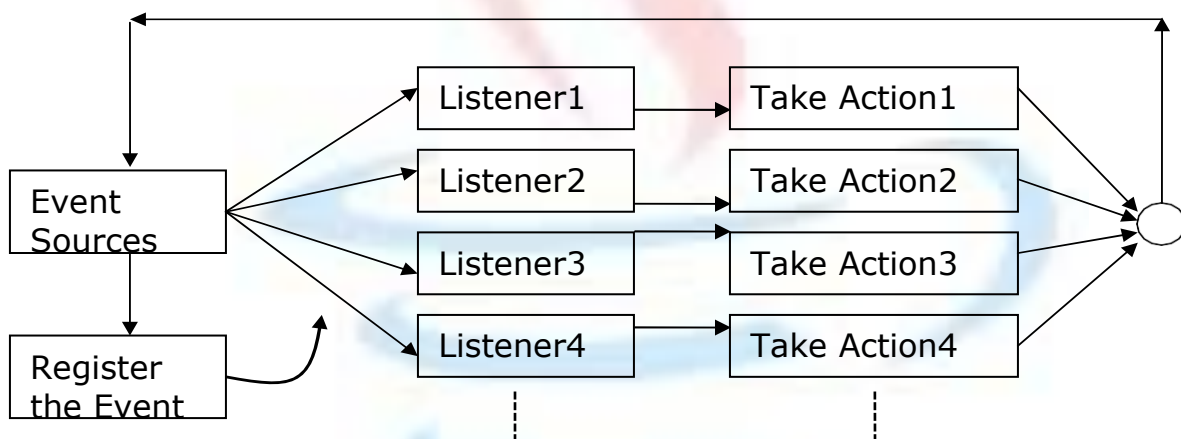


Fig. Delegation Event Model

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. This is a more efficient way to handle events than the design used by the old Java 1.0 approach.

Java also allows us to process events without using the delegation event model. This can be done by extending an AWT component. However, the delegation event model is the preferred design for the reasons just cited.

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, software or hardware failure occurs, or an operation is completed. We 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 addTypeListener(TypeListener el)
```

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called `addKeyListener()`. The method that registers a mouse motion listener is called `addMouseMotionListener()`. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method is this:

```
public void addTypeListener(TypeListener el)  
    throws java.util.TooManyListenersException
```

Here, *Type* is the name of the event and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event. A source must also provide a method that allows a listener to un-register an interest in a specific type of event. The general form of such a method is this:

```
public void removeTypeListener(TypeListener el)
```

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, to remove a keyboard listener, we would call `removeKeyListener()`. The methods that add or remove listeners are provided by the source that generates events. For example, the `Component` class provides methods to add and remove keyboard and mouse event listeners.

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. Any object may receive and process one or both of these events if it provides an implementation of this interface.

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. `EventObject` contains two methods: `getSource()` and `toString()`. The `getSource()` method returns the source of the event. Its general form is shown here:

```
Object getSource( )
```

As expected, `toString()` returns the string equivalent of the event. The class `AWTEvent`, defined within the `java.awt` package, is a subclass of `EventObject`. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.

- `EventObject` is a superclass of all events.
- `AWTEvent` is a superclass of all AWT events that are handled by the delegation event model.

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

Event Class	Description
<code>ActionEvent</code>	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
<code>AdjustmentEvent</code>	Generated when a scroll bar is manipulated.
<code>ComponentEvent</code>	Generated when a component is hidden, moved, resized, or becomes visible.
<code>ContainerEvent</code>	Generated when a component is added to or removed from a container.
<code>FocusEvent</code>	Generated when a component gains or loses keyboard focus.
<code>InputEvent</code>	Abstract super class for all component input event classes.
<code>ItemEvent</code>	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.
<code>KeyEvent</code>	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.
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.

ActionEvent

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. We 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 `getModifiers()` method returns a value that indicates which modifier keys (`ALT`, `CTRL`, `META`, and/or `SHIFT`) were pressed when the event was generated. Its form is shown here:

```
int getModifiers( )
```

The method `getWhen()` that returns the time at which the event took place. This is called the event's timestamp. The `getWhen()` method is shown here.

```
long getWhen( )
```

AdjustmentEvent

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:

<code>BLOCK_DECREMENT</code>	The user clicked inside the scroll bar to decrease its value.
------------------------------	---

BLOCK_INCREMENT The user clicked inside the scroll bar to increase its value.

TRACK The slider was dragged.

UNIT_DECREMENT The button at the end of the scroll bar was clicked to decrease its value.

UNIT_INCREMENT The button at the end of the scroll bar was clicked to increase its value.

The type of the adjustment event may be obtained by the `getAdjustmentType()` method. It returns one of the constants defined by `AdjustmentEvent`. The general form is shown here:

```
int getAdjustmentType( )
```

The amount of the adjustment can be obtained from the `getValue()` method, shown here:

```
int getValue( )
```

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

ComponentEvent

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.

`ComponentEvent` is the super-class either directly or indirectly of `ContainerEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, and `WindowEvent`. The `getComponent()` method returns the component that generated the event. It is shown here:

```
Component getComponent( )
```

ContainerEvent

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.

FocusEvent

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.

If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.) The other component involved in the focus change, called the opposite component, is passed in other. Therefore, if a FOCUS_GAINED event occurred, other will refer to the component that lost focus. Conversely, if a FOCUS_LOST event occurred, other will refer to the component that gains focus.

InputEvent

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

When writing new code, it is recommended that we use the new, extended modifiers rather than the original modifiers. To test if a modifier was pressed at the time an event is generated, use the isAltDown(), isAltGraphDown(), isControlDown(), isMetaDown(), and isShiftDown() methods. The forms of these methods are shown here:

```
boolean isAltDown( )
boolean isAltGraphDown( )
```

```
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

We can obtain a value that contains all of the original modifier flags by calling the `getModifiers()` method. It is shown here:

```
int getModifiers( )
```

We can obtain the extended modifiers by called `getModifiersEx()`, which is shown here.

```
int getModifiersEx( )
```

ItemEvent

An `ItemEvent` is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. There are two types of item events, which are identified by the following integer constants:

<code>DESELECTED</code>	The user deselected an item.
<code>SELECTED</code>	The user selected an item.

In addition, `ItemEvent` defines one integer constant, `ITEM_STATE_CHANGED`, that signifies a change of 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( )
```

KeyEvent

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.

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.

MouseEvent

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.

The most commonly used methods in this class are getX() and getY(). These returns the X and Y coordinate of the mouse when the event occurred. Their forms are shown here:

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


Alternatively, we can use the `getPoint()` method to obtain the coordinates of the mouse. It is shown here:

```
Point getPoint( )
```

It returns a `Point` object that contains the X, Y coordinates in its integer members: `x` and `y`. The `translatePoint()` method changes the location of the event. Its form is shown here:

```
void translatePoint(int x, int y)
```

Here, the arguments `x` and `y` are added to the coordinates of the event. The `getClickCount()` method obtains the number of mouse clicks for this event. Its signature is shown here:

```
int getClickCount( )
```

The `isPopupTrigger()` method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

```
boolean isPopupTrigger( )
```

Java 2, version 1.4 added the `getButton()` method, shown here.

```
int getButton( )
```

It returns a value that represents the button that caused the event. The return value will be one of these constants defined by `MouseEvent`.

`NOBUTTON` `BUTTON1` `BUTTON2` `BUTTON3`

The `NOBUTTON` value indicates that no button was pressed or released.

MouseWheelEvent

The `MouseWheelEvent` class encapsulates a mouse wheel event. It is a subclass of `MouseEvent` and was added by Java 2, version 1.4. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. `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.

MouseEvent defines methods that give us access to the wheel event. For obtaining the number of rotational units, call `getWheelRotation()`, shown here.

```
int getWheelRotation( )
```

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. For obtaining the type of scroll, call `getScrollType()`, shown next.

```
int getScrollType( )
```

It returns either `WHEEL_UNIT_SCROLL` or `WHEEL_BLOCK_SCROLL`. If the scroll type is `WHEEL_UNIT_SCROLL`, we can obtain the number of units to scroll by calling `getScrollAmount()`. It is shown here.

```
int getScrollAmount( )
```

TextEvent

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 `TextEvent` object does not include the characters currently in the text component that generated the event. Instead, our 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.

WindowEvent

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:

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

WINDOW_STATE_CHANGED The state of the window changed.

WindowEvent is a subclass of ComponentEvent. The most commonly used method in this class is getWindow(). It returns the Window object that generated the event. Its general form is shown here:

```
Window getWindow( )
```

Java 2, version 1.4, adds methods that return the opposite window (when a focus event has occurred), the previous window state, and the current window state. These methods are shown here:

```
Window getOppositeWindow()  
int getOldState()  
int getNewState()
```

Sources of Events

Following is list of some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, other components, such as an applet, can generate events. For example, we receive key and mouse events from an applet. (We may also build our own components that generate events.)

<i>Event Source</i>	<i>Description</i>
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.

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.

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)
```

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)
```

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 AWT processes the resize and move events. The `componentResized()` and `componentMoved()` methods are provided for notification purposes only.

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)
```

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)
```

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)
```

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)
```

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)
```


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)
```

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.

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)
```

WindowFocusListener Interface

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

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

WindowFocusListener was added by Java 2, version 1.4.

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)
```

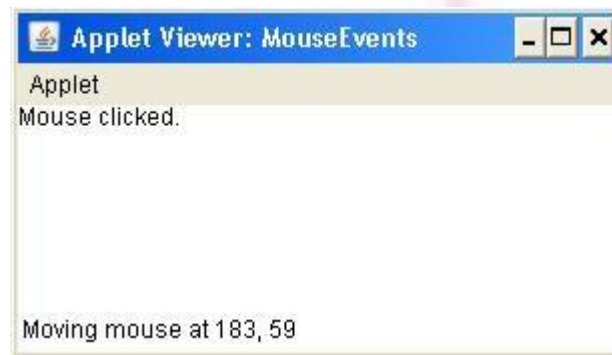
Handling Mouse Events

In order to handle mouse events, we must implement the `MouseListener` and the `MouseMotionListener` interfaces.

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

}



Here, 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

We will be implementing the `KeyListener` interface for handling keyboard events. Before looking at an example, it is useful to review how key events are generated. 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 we care about are actual characters, then we can ignore the information passed by the key press and release events. However, if our program needs to handle special keys, such

as the arrow or function keys, then it must watch for them through the keyPressed() handler.

There is one other requirement that our program must meet before it can process keyboard events: it must request input focus. To do this, call requestFocus(), which is defined by Component. If we don't, then our program will not receive any keyboard events. The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
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);
    }
}
```




If we want to handle the special keys, such as the arrow or function keys, we need to respond to them within the `keyPressed()` handler. They are not available through `keyTyped()`. To identify the keys, we use their virtual key codes. For example, the next method shows the use of special keys:

```
public void keyPressed(KeyEvent ke)
{
    showStatus("Key Down");
    int key = ke.getKeyCode();
    switch(key)
    {
        case KeyEvent.VK_F1:
            msg += "<F1>";
            break;
        case KeyEvent.VK_F2:
            msg += "<F2>";
            break;
        case KeyEvent.VK_F3:
            msg += "<F3>";
            break;
        case KeyEvent.VK_PAGE_DOWN:
            msg += "<PgDn>";
            break;
        case KeyEvent.VK_PAGE_UP:
            msg += "<PgUp>";
            break;
        case KeyEvent.VK_LEFT:
            msg += "<Left Arrow>";
            break;
        case KeyEvent.VK_RIGHT:
            msg += "<Right Arrow>";
            break;
    }
    repaint();
}
```

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 we want to receive and process only some of the events that are handled by a particular event listener interface. We 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 we are interested. 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.

List below shows the commonly used adapter classes in `java.awt.event` and notes the interface that each implements. The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. `AdapterDemo` extends `Applet`. Its `init()` method creates an instance of `MyMouseListener` and registers that object to receive notifications of mouse events. It also creates an instance of `MyMouseMotionAdapter` and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument. `MyMouseListener` implements the `mouseClicked()` method. The other mouse events are silently ignored by code inherited from the `MouseListener` class. `MyMouseMotionAdapter` implements the `mouseDragged()` method. The other mouse motion event is silently ignored by code inherited from the `MouseMotionAdapter` class.

Adapter Class	Listener Interface
<code>ComponentAdapter</code>	<code>ComponentListener</code>
<code>ContainerAdapter</code>	<code>ContainerListener</code>
<code>FocusAdapter</code>	<code>FocusListener</code>
<code>KeyAdapter</code>	<code>KeyListener</code>
<code>MouseListener</code>	<code>MouseListener</code>
<code>MouseMotionAdapter</code>	<code>MouseMotionListener</code>
<code>WindowAdapter</code>	<code>WindowListener</code>

```
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");
    }
}

```

As we can see by looking at the program, not having to implement all of the methods defined by the `MouseMotionListener` and `MouseListener` interfaces saves our considerable amount of effort and prevents our code from becoming cluttered with empty methods.

Inner Classes

For understanding the benefit provided by inner classes, consider the applet shown in the following listing. It does not use an inner class. Its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. `MousePressedDemo` extends `Applet`, and `MyMouseAdapter` extends

MouseListener. The `init()` method of `MousePressedDemo` instantiates `MyMouseListener` and provides this object as an argument to the `addMouseListener()` method. Notice that a reference to the applet is supplied as an argument to the `MyMouseListener` constructor. This reference is stored in an instance variable for later use by the `mousePressed()` method. When the mouse is pressed, it invokes the `showStatus()` method of the applet through the stored applet reference. In other words, `showStatus()` is invoked relative to the applet reference stored by `MyMouseListener`.

```
// This applet does NOT use an inner
class. import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet
{
    public void init()
    {
        addMouseListener(new MyMouseListener(this));
    }
}
class MyMouseListener extends MouseAdapter
{
    MousePressedDemo mousePressedDemo;
    public MyMouseListener(MousePressedDemo mousePressedDemo)
    {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me)
    {
        mousePressedDemo.showStatus("Mouse Pressed.");
    }
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, `InnerClassDemo` is a top-level class that extends `Applet`. `MyMouseListener` is an inner class that extends `MouseListener`. Because `MyMouseListener` is defined within the scope of `InnerClassDemo`, it has access to all of the variables and methods within the scope of that class. Therefore, the `mousePressed()` method can call the `showStatus()` method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass `MyMouseListener()` a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
```

```

import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/
public class InnerClassDemo extends Applet
{
    public void init()
    {
        addMouseListener(new MyMouseAdapter());
    }
    class MyMouseAdapter extends MouseAdapter
    {
        public void mousePressed(MouseEvent me)
        {
            showStatus("Mouse Pressed");
        }
    }
}

```

Anonymous Inner Classes

An anonymous inner class is one that is not assigned a name. Consider the applet shown in the following listing. As before, its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

```

// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet
{
    public void init()
    {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                showStatus("Mouse Pressed");
            }
        });
    }
}

```

There is one top-level class in this program: AnonymousInnerClassDemo. The `init()` method calls the `addMouseListener()` method. Its argument is an

expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully. The syntax `new MouseAdapter() { ... }` indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends `MouseAdapter`. This new class is not named, but it is automatically instantiated when this expression is executed. Because this anonymous inner class is defined within the scope of `AnonymousInnerClassDemo`, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the `showStatus()` method directly. As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow us to create more efficient code.

Handling Buttons

```
// 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);
}
}

```

Handling Checkboxes

```

// 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);
}
}

```

Handling Radio Buttons

```

// 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);
    }
}

```

Handling Choice Controls

```

// 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);
    }
}

```

Handling Lists

```

// 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);
    }
}

```

Handling Scrollbars

```

// 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());
    }
}

```

Handling Text field

```

// 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);
    }
}

```

Handling Menus

```

// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
    <applet code="MenuDemo1" 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 MenuDemo1 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);
    }
}

```

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. We 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)

```

The first form creates a default card layout. The second form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type Panel. This panel must have CardLayout selected as its layout manager. The cards that form the deck are also typically objects of type Panel. Thus, we must create a panel that contains the deck and a panel for each card in the deck. Next, we add to the appropriate panel the components that form each card. We then add these panels to the panel for which CardLayout is the layout manager. Finally, we add this panel to the main applet panel. Once these steps are complete, we must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name. Thus, most of the time, we 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 we have created a deck, our 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. Calling *first()* causes the first card in the deck to be shown. For showing the last card, call *last()* and for the next card, call *next()*. To show the previous card, call *previous()*. Both *next()* and *previous()* automatically cycle back to the top or bottom of the deck, respectively. The *show()* method displays the card whose name is passed in *cardName*. The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Macintosh and Solaris are displayed in the other card.

The process of creating a card layout is visualized as below:

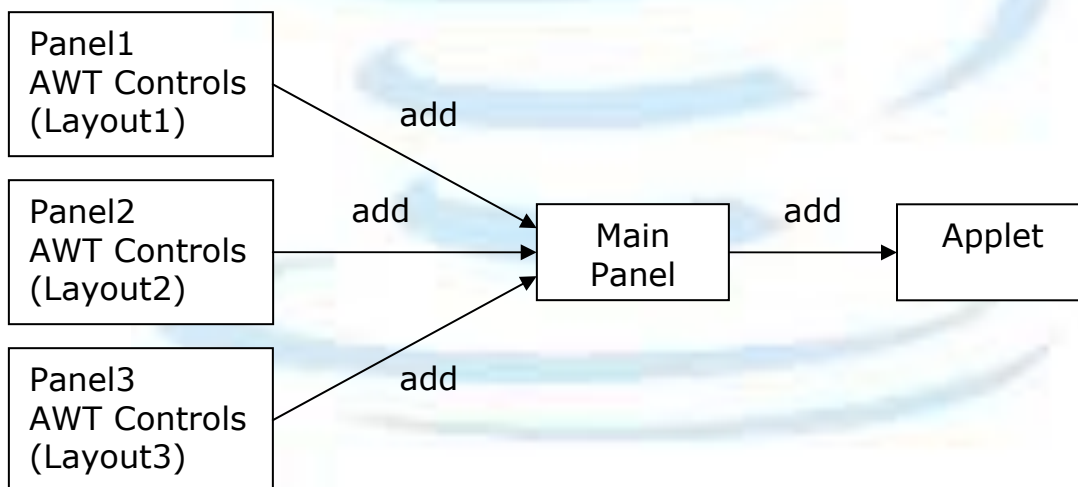


Fig. Creation of card layout

```
// 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.setLayout(new BorderLayout());  
    winPan.add(Win98, BorderLayout.NORTH);  
    winPan.add(winNT, BorderLayout.SOUTH);  
  
    // Add other OS check boxes to a panel  
    Panel otherPan = new Panel();  
    otherPan.add(solaris);  
    otherPan.add(mac);  
    otherPan.setLayout(new GridLayout(2,2));  
  
    // 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);
}

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");
}
}

```

Handling Events by Extending AWT Components

Java also allows us to handle events by subclassing AWT components. Doing so allows us to handle events in much the same way as they were handled under the original 1.0 version of Java. Of course, this technique is discouraged, because it has the same disadvantages of the Java 1.0 event model, the main one being inefficiency. In order to extend an AWT component, we must call the `enableEvents()` method of `Component`. Its general form is shown here:

```
protected final void enableEvents(long eventMask)
```

The `eventMask` argument is a bit mask that defines the events to be delivered to this component. The `AWTEvent` class defines `int` constants for making this mask. Several are shown here:

<code>ACTION_EVENT_MASK</code>	<code>KEY_EVENT_MASK</code>
<code>ADJUSTMENT_EVENT_MASK</code>	<code>MOUSE_EVENT_MASK</code>
<code>COMPONENT_EVENT_MASK</code>	<code>MOUSE_MOTION_EVENT_MASK</code>
<code>CONTAINER_EVENT_MASK</code>	<code>MOUSE_WHEEL_EVENT_MASK</code>
<code>FOCUS_EVENT_MASK</code>	<code>TEXT_EVENT_MASK</code>

INPUT_METHOD_EVENT_MASK WINDOW_EVENT_MASK
ITEM_EVENT_MASK

We must also override the appropriate method from one of our superclasses in order to process the event. Methods listed below most commonly used and the classes that provide them.

Event Processing Methods

Class	Processing Methods
Button	processActionEvent()
Checkbox	processItemEvent()
CheckboxMenuItem	processItemEvent()
Choice	processItemEvent()
Component	processComponentEvent(), processFocusEvent(), processKeyEvent(), processMouseEvent(), processMouseMotionEvent(), processMouseWheelEvent ()
List	processActionEvent(), processItemEvent()
MenuItem	processActionEvent()
Scrollbar	processAdjustmentEvent()
TextComponent	processTextEvent()

Extending Button

The following program creates an applet that displays a button labeled "Test Button". When the button is pressed, the string "action event: " is displayed on the status line of the applet viewer or browser, followed by a count of the number of button presses. The program has one top-level class named ButtonDemo2 that extends Applet. A static integer variable named `i` is defined and initialized to zero. It records the number of button pushes. The `init()` method instantiates `MyButton` and adds it to the applet. `MyButton` is an inner class that extends `Button`. Its constructor uses `super` to pass the label of the button to the superclass constructor. It calls `enableEvents()` so that action events may be received by this object. When an action event is generated, `processActionEvent()` is called. That method displays a string on the status line and calls `processActionEvent()` for the superclass. Because `MyButton` is an inner class, it has direct access to the `showStatus()` method of `ButtonDemo2`.

```
/*  
* <applet code=ButtonDemo2 width=200 height=100>  
* </applet>  
*/  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;
```

```

public class ButtonDemo2 extends Applet
{
    MyButton myButton;
    static int i = 0;
    public void init()
    {
        myButton = new MyButton("Test Button");
        add(myButton);
    }
    class MyButton extends Button
    {
        public MyButton(String label)
        {
            super(label);
            enableEvents(AWTEvent.ACTION_EVENT_MASK);
        }
        protected void processActionEvent(ActionEvent ae)
        {
            showStatus("action event: " + i++);
            super.processActionEvent(ae);
        }
    }
}

```

Extending Checkbox

The following program creates an applet that displays three check boxes labeled "Item 1", "Item 2", and "Item 3". When a check box is selected or deselected, a string containing the name and state of that check box is displayed on the status line of the applet viewer or browser.

The program has one top-level class named `CheckboxDemo2` that extends `Applet`. Its `init()` method creates three instances of `MyCheckbox` and adds these to the applet. `MyCheckbox` is an inner class that extends `Checkbox`. Its constructor uses `super` to pass the label of the check box to the superclass constructor. It calls `enableEvents()` so that item events may be received by this object. When an item event is generated, `processItemEvent()` is called. That method displays a string on the status line and calls `processItemEvent()` for the superclass.

```

/*
 * <applet code=CheckboxDemo2 width=300 height=100>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CheckboxDemo2 extends Applet

```

```

{
    MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
    public void init()
    {
        myCheckbox1 = new MyCheckbox("Item 1");
        add(myCheckbox1);
        myCheckbox2 = new MyCheckbox("Item 2");
        add(myCheckbox2);
        myCheckbox3 = new MyCheckbox("Item 3");
        add(myCheckbox3);
    }
}
class MyCheckbox extends Checkbox
{
    public MyCheckbox(String label)
    {
        super(label);
        enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie)
    {
        showStatus("Checkbox name/state: " + getLabel() +
            "/" + getState());
        super.processItemEvent(ie);
    }
}
}
}

```

Extending a Check Box Group

The following program reworks the preceding check box example so that the check boxes form a check box group. Thus, only one of the check boxes may be selected at any time.

```

/*
 * <applet code=CheckboxGroupDemo2 width=300 height=100>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CheckboxGroupDemo2 extends Applet
{
    CheckboxGroup cbg;
    MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
    public void init()
    {
        cbg = new CheckboxGroup();
        myCheckbox1 = new MyCheckbox("Item 1", cbg, true);

```

```

        add(myCheckbox1);
        myCheckbox2 = new MyCheckbox("Item 2", cbg, false);
        add(myCheckbox2);
        myCheckbox3 = new MyCheckbox("Item 3", cbg, false);
        add(myCheckbox3);
    }
    class MyCheckbox extends Checkbox
    {
        public MyCheckbox(String label, CheckboxGroup cbg,
            boolean flag)
        {
            super(label, cbg, flag);
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
        protected void processItemEvent(ItemEvent ie)
        {
            showStatus("Checkbox name/state: " + getLabel() +
                "/" + getState());
            super.processItemEvent(ie);
        }
    }
}

```

Extending Choice

The following program creates an applet that displays a choice list with items labeled "Red", "Green", and "Blue". When an entry is selected, a string that contains the name of the color is displayed on the status line of the applet viewer or browser. There is one top-level class named ChoiceDemo2 that extends Applet. Its init() method creates a choice element and adds it to the applet. MyChoice is an inner class that extends Choice. It calls enableEvents() so that item events may be received by this object. When an item event is generated, processItemEvent() is called. That method displays a string on the status line and calls processItemEvent() for the superclass.

```

/*
 * <applet code=ChoiceDemo2 width=300 height=100>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ChoiceDemo2 extends Applet
{
    MyChoice choice;
    public void init()
    {
        choice = new MyChoice();
    }
}

```

```

        choice.add("Red");
        choice.add("Green");
        choice.add("Blue");
        add(choice);
    }
    class MyChoice extends Choice
    {
        public MyChoice()
        {
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
        protected void processItemEvent(ItemEvent ie)
        {
            showStatus("Choice selection: " +
                getSelectedItem());
            super.processItemEvent(ie);
        }
    }
}

```

Extending List

The following program modifies the preceding example so that it uses a list instead of a choice menu. There is one top-level class named ListDemo2 that extends Applet. Its init() method creates a list element and adds it to the applet. MyList is an inner class that extends List. It calls enableEvents() so that both action and item events may be received by this object. When an entry is selected or deselected, processItemEvent() is called. When an entry is double-clicked, processActionEvent() is also called. Both methods display a string and then hand control to the superclass.

```

/*
 * <applet code=ListDemo2 width=300 height=100>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ListDemo2 extends Applet
{
    MyList list;
    public void init()
    {
        list = new MyList();
        list.add("Red");
        list.add("Green");
        list.add("Blue");
    }
}

```



```

        add(list);
    }
    class MyList extends List
    {
        public MyList()
        {
            enableEvents(AWTEvent.ITEM_EVENT_MASK |
                AWTEvent.ACTION_EVENT_MASK);
        }
        protected void processActionEvent(ActionEvent ae)
        {
            showStatus("Action event: " +
                ae.getActionCommand());
            super.processActionEvent(ae);
        }
        protected void processItemEvent(ItemEvent ie)
        {
            showStatus("Item event: " + getSelectedItem());
            super.processItemEvent(ie);
        }
    }
}

```

Extending Scrollbar

The following program creates an applet that displays a scroll bar. When this control is manipulated, a string is displayed on the status line of the applet viewer or browser. That string includes the value represented by the scroll bar. There is one top-level class named ScrollbarDemo2 that extends Applet. Its `init()` method creates a scroll bar element and adds it to the applet. `MyScrollbar` is an inner class that extends `Scrollbar`. It calls `enableEvents()` so that adjustment events may be received by this object. When the scroll bar is manipulated, `processAdjustmentEvent()` is called. When an entry is selected, `processAdjustmentEvent()` is called. It displays a string and then hands control to the superclass.

```

/*
 * <applet code=ScrollbarDemo2 width=300 height=100>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ScrollbarDemo2 extends Applet
{
    MyScrollbar myScrollbar;
    public void init()

```

```
{
    myScrollbar = new
    MyScrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
    100);
    add(myScrollbar);
}
class MyScrollbar extends Scrollbar
{
    public MyScrollbar(int style, int initial, int
    thumb, int min, int max)
    {
        super(style, initial, thumb, min, max);
        enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
    }
    protected void processAdjustmentEvent(AdjustmentEvent
    ae)
    {
        showStatus("Adjustment event: " +
        ae.getValue()); setValue(getValue());
        super.processAdjustmentEvent(ae);
    }
}
}
```

The Java logo, featuring the word "Java" in a stylized, red, lowercase font. The letters are thick and rounded, with a slight shadow effect. The logo is positioned at the bottom of the page, centered horizontally.

References

1. Java 2 the Complete Reference,

Fifth Edition by Herbert Schildt, 2001 Osborne

McGraw Hill. Chapter 20: Event Handling

Chapter 21: Introducing the AWT: Working with Windows,
Graphics, and Text

Chapter 22: Using AWT Controls, Layout Managers, and Menus

(Most of the data is referred from this book)

2. Learning Java,

3rd Edition , By Jonathan Knudsen, Patrick Niemeyer, O'Reilly, May
2005

Chapter 19: Layout Managers



Java