Chapter 01
Introduction the Abstract Window Toolkit (AWT)

Contents:

1.1 Working with Windows and AWT
AWT classes
Windows Fundamentals
Working with frame windows
Creating a frame window in applet
Creating windowed program
Display information within a window
1.2 Working with graphics
Working with color
Setting the paint mode
Working with Fonts
Managing text output using Font Metrics
Exploring text & graphics
1.3 Using AWT Controls, Layout Managers and Menus
Control Fundamentals
Labels
Using Buttons
Applying Check Boxes
Checkbox Group
Choice Controls
Using Lists
Managing scroll Bars
Using a Text Field
Using a Text Area
Understanding Layout Managers
Menu Bars and Menu
Dialog Boxes
File Dialog
1.4 Handling events by Extending AWT Components
Exploring the Controls, Menus, and Layout Managers

Abstract Window Toolkit:

The AWT contains numerous classes and methods that allow us to create and manage windows. Although the main purpose of the AWT is to support applet windows, it can also be used to create stand-alone windows that run in a GUI environment, such as Windows.

AWT Classes

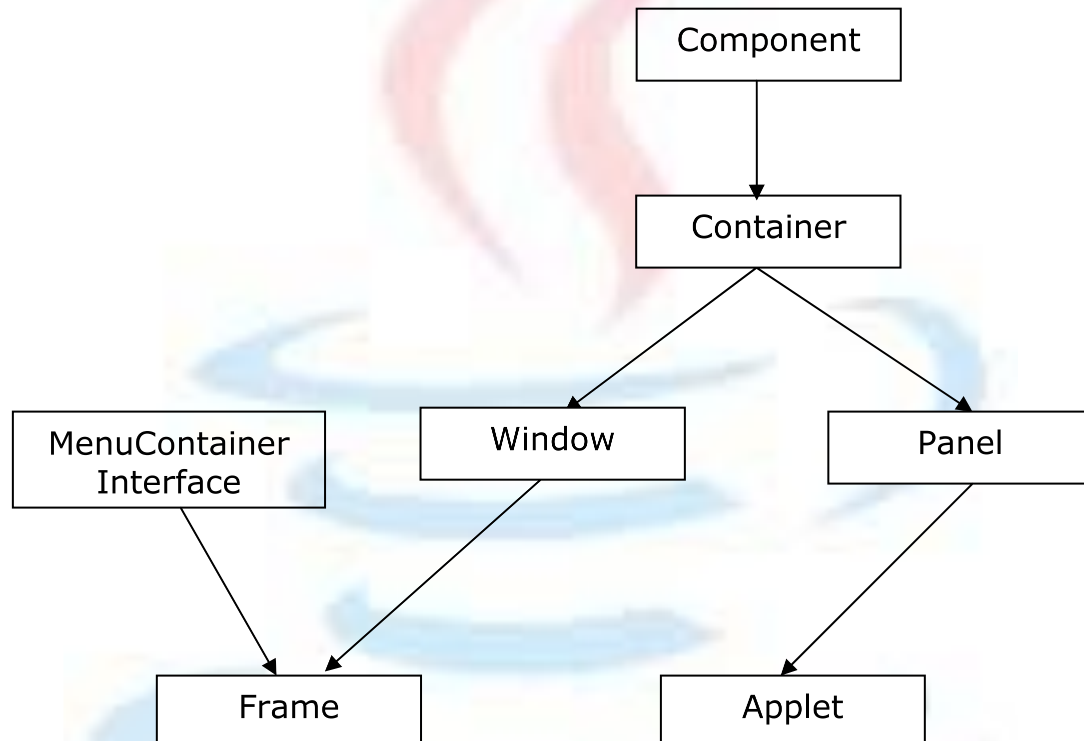
The AWT classes are contained in the java.awt package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe.

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 super-class 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 or printer.
GraphicsEnvironment	Describes the collection of available Font and GraphicsDevice objects.
GridBagConstraints	Defines various constraints relating to the GridBagLayout class.
GridBagLayout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints.
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid.
Image	Encapsulates graphical images.
Insets	Encapsulates the borders of a container.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
MediaTracker	Manages media objects.
Menu	Creates a pull-down menu.
MenuBar	Creates a menu bar.
MenuComponent	An abstract class implemented by various menu classes.
MenuItem	Creates a menu item.
MenuShortcut	Encapsulates a keyboard shortcut for a menu item.
Panel	The simplest concrete subclass of Container.
Point	Encapsulates a Cartesian coordinate pair, stored in ***x*** and ***y***.
Polygon	Encapsulates a polygon.
PopupMenu	Encapsulates a pop-up menu.
PrintJob	An abstract class that represents a print job.
Rectangle	Encapsulates a rectangle.
Robot	Supports automated testing of AWT- based applications.
Scrollbar	Creates a scroll bar control.
ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
TextArea	Creates a multiline edit control.
TextComponent	A superclass for TextArea and TextField.
TextField	Creates a single-line edit control.
Toolkit	Abstract class implemented by the AWT.
Window	Creates a window with no frame, no menu bar,

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure below shows the class hierarchy for Panel and Frame.



Component

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a **visual component**. **All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.** It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A Component object is responsible for remembering the current foreground and background **colors** and the currently selected text **font**.

Container

The Container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a Container (since they are themselves instances of Component). This makes for a multileveled containment system. **A container is responsible for laying out (that is,**

positioning) any components that it contains. It does this through the use of various layout managers.

Panel

The Panel class is a concrete subclass of Container. It doesn't add any new methods; *it simply implements Container*. A Panel may be thought of as a recursively nestable, concrete screen component. Panel is the super-class for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object. In essence, **a Panel is a window that does not contain a title bar, menu bar, or border**. This is why we don't see these items when an applet is run inside a browser. When we run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a Panel object by its add( ) method (inherited from Container). Once these components have been added, we can position and resize them manually using the setLocation( ), setSize( ), or setBounds( ) methods defined by Component.

Window

The Window class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop. Generally, we won't create Window objects directly. Instead, we will use a subclass of Window called Frame.

Frame

Frame encapsulates what is commonly thought of as a "window." **It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.** If we create a Frame object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. When a Frame window is created by a program rather than an applet, a normal window is created.

Canvas

Canvas encapsulates a blank window upon which we can draw.

Dimension

This class encapsulates the 'width' and 'height' of a component (in integer precision) in a single object. The class is associated with certain properties of components. Several methods defined by the Component class and the LayoutManager interface return a Dimension object.

Normally the values of width and height are non-negative integers. The constructors that allow us to create a dimension do not prevent us from setting

a negative value for these properties. If the value of width or height is negative, the behavior of some methods defined by other objects is undefined.

Fields of Dimension:

int height	The height dimension; negative values can be used.
int width	The width dimension; negative values can be used.

Constructors:

Dimension()
It creates an instance of Dimension with a width of zero and a height of zero.

Dimension(Dimension d)
It creates an instance of Dimension whose width and height are the same as for the specified dimension.

Dimension(int width, int height)
It constructs a Dimension and initializes it to the specified width and specified height.

Methods:

boolean equals(Object obj)
It checks whether two dimension objects have equal values.
double getHeight()
It returns the height of this dimension in double precision.
Dimension getSize()
It gets the size of this Dimension object.
double getWidth()
It returns the width of this dimension in double precision.
void setSize(Dimension d)
It sets the size of this Dimension object to the specified size.
void setSize(double width, double height)
It sets the size of this Dimension object to the specified width and height in double precision.
void setSize(int width, int height)
It sets the size of this Dimension object to the specified width and height.

Working with Frame Windows

After the applet, the type of window we will most often create is derived from Frame. We will use it to create child windows within applets, and top-level

or child windows for applications. It creates a standard-style window. Following are two of Frame's constructors:

```
Frame( )
Frame(String title)
```

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title. Note that we cannot specify the dimensions of the window. Instead, we must set the size of the window after it has been created.

Setting the Windows Dimensions

The setSize( ) method is used to set the dimensions of the window. Its signature is shown below:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

The new size of the window is specified by 'newWidth' and 'newHeight', or by the 'width' and 'height' fields of the Dimension object passed in 'newSize'. The dimensions are specified in terms of pixels. The getSize( ) method is used to obtain the current size of a window. Its signature is:

```
Dimension getSize( )
```

This method returns the current size of the window contained within the 'width' and 'height' fields of a Dimension object.

Hiding and showing a Window

After a frame window has been created, it will not be visible until we call setVisible( ). Its signature is:

```
void setVisible(boolean visibleFlag)
```

The component is visible if the argument to this method is true. Otherwise, it is hidden.

Setting a Windows Title

We can change the title in a frame window using setTitle( ), which has this general form:

```
void setTitle(String newTitle)
```

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

Closing a Frame Window

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

Creating a Frame Window in an Applet

The following applet creates a subclass of Frame called SampleFrame. A window of this subclass is instantiated within the init( ) method of AppletFrame. Note that 'SampleFrame' calls Frame's constructor. This causes a standard frame window to be created with the title passed in title. This example overrides the applet window's start( ) and stop( ) method s so that they show and hide the child window, respectively. It also causes the child window to be shown when the browser returns to the applet.

```
/*
   <applet code="AppletFrame" width=300 height=50>
   </applet>
*/
class SampleFrame extends Frame
{
        SampleFrame(String title)
        {
             super(title);
        }
        public void paint(Graphics g)
        {
             g.drawString("This is in frame window", 10, 40);
        }
}
public class AppletFrame extends Applet
{
        Frame f;
        public void init()
        {
             f = new SampleFrame("A Frame Window");
             f.setSize(250, 250);
             f.setVisible(true);
        }
        public void start()
        {
             f.setVisible(true);
        }
```

```
            public void stop()
            {
                 f.setVisible(false);
            }
            public void paint(Graphics g)
            {
                 g.drawString("This is in applet window", 10, 20);
            }
}
```

Output:

Creating a Windowed Program

Although creating applets is the most common use for Java's AWT, it is possible to create stand-alone AWT-based applications, too. To do this, simply create an instance of the window or windows we need inside main( ). For example, the following program creates a simple frame window.

```
public class AppWindow extends Frame
{
     AppWindow(String title)
     {
        super(title);
     }
     public void paint(Graphics g)
     {
        setForeground(Color.red);
        setBackground(Color.cyan);
        g.drawString("This is my frame", 30, 70);
```

```
        }
        public static void main(String args[])
           throws Exception
        {
           AppWindow appwin = new AppWindow("Frame window");
           appwin.setSize(new Dimension(300, 200));
           appwin.setVisible(true);
           Thread.sleep(5000);
           appwin.setTitle("An AWT-Based Application");
           Thread.sleep(5000);
           System.exit(0);
        }
   }
```

Another Program:

```
   public class AppWindow extends Frame
   {
        Font f;
        static int val=5;
        AppWindow(String title)
        {
           super(title);
        }
        public void paint(Graphics g)
        {
           setForeground(Color.red);
           setBackground(Color.cyan);

           Integer i = new Integer(val);
           g.drawString(i.toString(), 30, 70);
           val--;
        }
        public static void main(String args[])
           throws Exception
        {
           AppWindow appwin = new AppWindow("Frame window");
           appwin.setSize(new Dimension(300, 200));
           appwin.setFont(new Font("Arial",Font.BOLD, 40));
           appwin.setVisible(true);
           Thread.sleep(5000);
           appwin.setTitle("An AWT-Based Application");

           for(int i=0;i<5;i++)
           {
              Thread.sleep(1000);
              appwin.repaint();
           }
```

```
            System.exit(0);
        }
    }
```

Displaying information within a Window

In the most general sense, a window is a container for information. Although we have already output small amounts of text to a window in the preceding examples, we have not begun to take advantage of a window's ability to present high-quality text and graphics. Indeed, much of the power of the AWT comes from its support for these items.

Working with Graphics

Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows us to specify any color that we want. It then finds the best match for that color, given the limits of the display hardware currently executing our program or applet. Thus, our code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the Color class.

Constructors:

```
Color(int red, int green, int blue)
Color(int rgbValue)
Color(float red, float green, float blue)
```

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red.
```

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

The final constructor, Color(float, float, float), takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue. Once we have created a color, we can use it to set the foreground and/or

background color by using the setForeground( ) and setBackground( ) methods. You can also select it as the current drawing color.

Color Methods

The Color class defines several methods that help manipulate colors.

Using Hue, Saturation, and Brightness

The hue-saturation-brightness (HSB) color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, hue is a wheel of color. The hue is specified with a number between 0.0 and 1.0 (the colors are approximately: red, orange, yellow, green, blue, indigo, and violet). Saturation is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. Brightness values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. Color supplies two methods that let us convert between RGB and HSB. They are shown here:

```
static int HSBtoRGB(float hue, float sat, float brightness)
static float[ ] RGBtoHSB(int r, int g, int b, float values[ ])
```

HSBtoRGB( ) returns a packed RGB value compatible with the Color(int) constructor. RGBtoHSB( ) returns a float array of HSB values corresponding to RGB integers. If *values* is not null, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed( ), getGreen( ), getBlue( )

We can obtain the red, green, and blue components of a color independently using getRed( ), getGreen( ), and getBlue( ), shown below:

```
int getRed( )
int getGreen( )
int getBlue( )
```

Each of these methods returns the RGB color component found in the invoking Color object in the lower 8 bits of an integer.

getRGB( )

To obtain a packed, RGB representation of a color, use getRGB( ), shown here:

```
int getRGB( )
```

The return value is organized as described earlier.

Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. We can change this color by calling the Graphics method setColor( ):

```
void setColor(Color newColor)
```

Here, 'newColor' specifies the new drawing color. We can obtain the current color by calling getColor( ), shown below:

```
Color getColor( )
```

Example:

```
/*
     <applet code="ColorDemo" width=300 height=200>
     </applet>
*/
public class ColorDemo extends Applet
{
     public void paint(Graphics g)
     {
        Color c1 = new Color(202, 146, 20);
        Color c2 = new Color(110, 169, 107);
        Color c3 = new Color(160, 100, 200);

        g.setColor(c1);
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);

        g.setColor(Color.red);
        g.drawLine(40, 25, 250, 180);

        g.setColor(c3);
        g.drawLine(20, 150, 400, 40);

        g.setColor(c2);
        g.drawOval(10, 10, 50, 50);
        g.fillOval(70, 90, 140, 100);
     }
}
```

Setting the Paint Mode

The paint-mode determines how objects are drawn in a window. By default, new output to a window overwrites any pre-existing contents. However, it is possible to have new objects XORed onto the window by using setXORMode( ), as follows:

```
void setXORMode(Color xorColor)
```

Here, 'xorColor' specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over. To return to overwrite mode, call setPaintMode( ), shown here:

```
void setPaintMode( )
```

In general, we will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.

Working with Fonts

The AWT supports multiple type fonts. It provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts. In Java, fonts have a family name, a logical font name, and a face name. The family name is the general name of the font, such as Courier. The logical name specifies a category of font, such as Monospaced. The face name specifies a specific font, such as Courier Italic. Fonts are encapsulated by the Font class. Several of the methods defined by Font are listed below.

The Font class defines these variables:

Variable	Meaning
String name	Name of the font
float pointSize	Size of the font in points
int size	Size of the font in points
int style	Font style

Method	Description
static Font decode(String str)	Returns a font given its name.
boolean equals(Object FontObj)	Returns true if the invoking object contains the same font as that specified by FontObj. Otherwise, it returns false.
String getFamily( )	Returns the name of the font family to which the invoking font belongs.

staticFontgetFont(Stringproperty)	Returns the font associated with the system property specified by property. Null is returned if property does not exist.
staticFontgetFont(Stringproperty, Font defaultFont)	Returns the font associated with the system property specified by property. The font specified by default Font is returned if property does not exist.
String getFontName()	Returns the face name of the invoking font.
String getName( )	Returns the logical name of the invoking font.
int getSize( )	Returns the size, in points, of the invoking font.
int getStyle( )	Returns the style values of the invoking font.
int hashCode( )	Returns the hash code associated with the invoking object.
boolean isBold( )	Returns true if the font includes the BOLD style value. Otherwise, false is returned.
boolean isItalic( )	Returns true if the font includes the ITALIC style value. Otherwise, false is returned.
boolean isPlain( )	Returns true if the font includes the PLAIN style value. Otherwise, false is returned.
String toString( )	Returns the string equivalent of the invoking font.

Determining the Available Fonts

When working with fonts, often we need to know which fonts are available on our machine. To obtain this information, we can use the getAvailableFontFamilyNames( ) method defined by the GraphicsEnvironment class.

It is shown here:

```
String[ ] getAvailableFontFamilyNames( )
```

This method returns an array of strings that contains the names of the available font families. In addition, the getAllFonts( ) method is defined by the GraphicsEnvironment class. It is shown here:

```
Font[ ] getAllFonts( )
```

This method returns an array of Font objects for all of the available fonts. Since these methods are members of GraphicsEnvironment, we need a GraphicsEnvironment reference to call them. We can obtain this reference by

using the getLocalGraphicsEnvironment( ) static method, which is defined by GraphicsEnvironment. It is shown here:

```
static GraphicsEnvironment getLocalGraphicsEnvironment( )
```

Here is an applet that shows how to obtain the names of the available font families:

```
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet
{
     public void paint(Graphics g)
     {
          String msg = "";
          String FontList[];

          GraphicsEnvironment ge =
              GraphicsEnvironment.getLocalGraphicsEnvironment();
          FontList = ge.getAvailableFontFamilyNames();
          for(int i = 0; i < FontList.length; i++)
              msg += FontList[i] + " ";

          g.drawString(msg, 4, 16);
     }
}
```

Creating and Selecting a Font

In order to select a new font, we must first construct a Font object that describes that font. One Font constructor has this general form:

```
Font(String fontName, int fontStyle, int pointSize)
```

Here, 'fontName' specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support the following fonts:

- Dialog,
- DialogInput,
- Sans Serif,
- Serif,

- Monospaced
- Symbol

Dialog is the font used by our system's dialog boxes. Dialog is also the default if we don't explicitly set a font. We can also use any other fonts supported by our particular environment, but these other fonts may not be universally available. The style of the font is specified by 'fontStyle'. It may consist of one or more of these three constants:

Font.PLAIN, Font.BOLD, and Font.ITALIC.

For combining styles, we can OR them together.

For example, Font.BOLD | Font.ITALIC specifies a bold, italics style.

The size, in points, of the font is specified by 'pointSize'. For using a font that we have created, we must select it using setFont( ), which is defined by Component. It has this general form:

```
void setFont(Font fontObj)
```

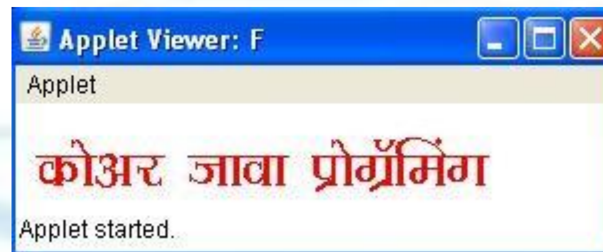
Here, fontObj is the object that contains the desired font.

```
// Displaying different  fonts
  import java.awt.Font;
  import java.awt.Graphics;
  import java.applet.Applet;
  /*
  <applet code="Fonts" width=300 height=150>
  </applet>
  */
  public class Fonts extends Applet
  {
    public void paint(Graphics g)
    {
        Font f1 = new Font("TimesRoman", Font.PLAIN, 18);
        Font f2 = new Font("Courier", Font.BOLD, 16);
        Font f3 = new Font("Arial", Font.ITALIC, 20);
        Font f4 = new Font("Times", Font.BOLD + Font.ITALIC,
                              22);
        g.setFont(f1);
        g.drawString("Times Roman plain font: 18", 10, 30);
        g.setFont(f2);
        g.drawString("Courier bold font: 16", 10, 60);
        g.setFont(f3);
        g.drawString("Arial italic font: 20", 10, 80);
        g.setFont(f4);
        g.drawString("Times bold italic font: 22", 10, 120);
    }
  }
```

We can set any font available in our 'windows\fonts' directory to the text on the applet. For example: the Devnagari font named "shusha" is freely available on internet. The following code displays applet with that font.

```
Font f = new Font("Shusha02", Font.PLAIN, 35);
g.setColor(Color.red);
g.setFont(f);
g.drawString("kaoAr jaavaa p`aoga`^imaMga", 10, 40);
```

Output window:



Obtaining Font Information

Suppose we want to obtain information about the currently selected font. To do this, we must first get the current font by calling getFont( ). This method is defined by the Graphics class, as shown here:

```
Font getFont( )
```

Once we have obtained the currently selected font, we can retrieve information about it using various methods defined by Font. For example, this applet displays the name, family, size, and style of the currently selected font:

```
// Display font info.
```

```
/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/

public class FontInfo extends Applet
{
  public void paint(Graphics g)
  {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();

    String msg = "Family: " + fontName;
    msg += ", Font: " + fontFamily;
    msg += ", Size: " + fontSize + ", Style: ";
    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";

    g.drawString(msg, 4, 16);
  }
}
```

Managing Text Output Using FontMetrics

Java supports a number of fonts. For most fonts, characters are not all the same dimension most fonts are proportional. Also, the height of each character, the length of descenders (the hanging parts of letters, such as y), and the amount of space between horizontal lines vary from font to font. Further, the point size of a font can be changed. That these (and other) attributes are variable would not be of too much consequence except that Java demands that the programmer, manually manage virtually all text output. Given that the size of each font may differ and that fonts may be changed while our program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font. For example, to write one line of text after another implies that we have some way of knowing how tall the font is and how many pixels are needed between lines. To fill this need, the AWT includes the FontMetrics class, which encapsulates various information about a font. Common terminology used when describing fonts:
Height The top-to-bottom size of the tallest character in the font.

Baseline The line that the bottoms of characters are aligned to (not counting descent).
Ascent The distance from the baseline to the top of a character
Descent The distance from the baseline to the bottom of a character
Leading The distance between the bottom of one line of text and the top of the next.

We have used the drawString( ) method in many of the previous examples. It paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods. It is a common error to draw a string at the same coordinate that we would draw a box. For example, if you were to draw a rectangle at coordinate 0,0 of your applet, you would see a full rectangle. If you were to draw the string Typesetting at 0,0, you would only see the tails (or descenders) of the y, p, and g.As you will see, by using font metrics, you can determine the proper placement of each string that you display. FontMetrics defines several methods that help you manage text output. The most commonly used are listed below. These methods help you properly display text in a window. Let’s look at some examples.

int bytesWidth(byte b[ ], int start, int numBytes)
Returns the width of numBytes characters held in array b, beginning at start.

int charWidth(char c[ ], int start, int numChars)
Returns the width of numChars characters held in array c, beginning at start.

Method	Description
int charWidth(char c)	Returns the width of c.
int charWidth(int c)	Returns the width of c.
int getAscent( )	Returns the ascent of the font.
int getDescent( )	Returns the descent of the font.
Font getFont( )	Returns the font.
int getHeight( )	Returns the height of a line of text. This value can be used to output multiple lines of text in a window.
int getLeading( )	Returns the space between lines of text.
int getMaxAdvance( )	Returns the width of the widest character. –1 is returned if this value is not available.
int getMaxAscent( )	Returns the maximum ascent.
int getMaxDescent( )	Returns the maximum descent.
int[ ] getWidths( )	Returns the widths of the first 256 characters.
int stringWidth(String str)	Returns the width of the string specified by str.
String toString( )	Returns the string equivalent of the invoking object.

Exploring Text and Graphics

Although we have covered the most important attributes and common techniques that we will use when displaying text or graphics, it only scratches the surface of Java's capabilities. This is an area in which further refinements and enhancements are expected as Java and the computing environment continue to evolve. For example, Java 2 added a subsystem to the AWT called Java 2D. Java 2D supports enhanced control over graphics, including such things as coordinate translations, rotation, and scaling. It also provides advanced imaging features. If advanced graphics handling is of interest to us, then we will definitely want to explore Java 2D in detail.

Using AWT Controls, Layout Managers and Menus

Controls are components that allow a user to interact with his application in various ways—for example; a commonly used control is the push button. A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them. In addition to the controls, a frame window can also include a standard-style menu bar. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. A menu bar is always positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls. While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task.

Control Fundamentals

The AWT supports the following types of controls:

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

These controls are subclasses of **Component**.

Adding and Removing Controls

In order to include a control in a window, we must add it to the window. So, we must first create an instance of the desired control and then add it to a

window by calling add( ), which is defined by **Container**. The add( ) method has several forms. The following form is the one that is used for the first part of this chapter:

```
Component add(Component compObj)
```

Here, compObj is an instance of the control that we want to add. A reference to compObj is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed. Sometimes we will want to remove a control from a window when the control is no longer needed. For doing this, call remove( ). This method is also defined by Container. It has this general form:

```
void remove(Component obj)
```

Here, obj is a reference to the control that we want to remove. We can remove all controls by calling removeAll( ).

Responding to Controls

Except for labels, which are passive controls, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, our program simply implements the appropriate interface and then registers an event listener for each control that we need to monitor.

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.

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

```
void setText(String str)
String getText( )
```

For setText( ), str specifies the new label. For getText( ), the current label is returned. You can set the alignment of the string within the label by calling setAlignment( ). To obtain the current alignment, call getAlignment( ). The methods are as follows:

```
void setAlignment(int how)
int getAlignment( )
```

Here, how must be one of the alignment constants shown earlier. The following example creates three labels and adds them to an applet:

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

Following is the window created by the LabelDemo applet. Notice that the labels are organized in the window by the default layout manager.

Buttons

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

```
Button( )
Button(String str)
```

The first version creates an empty button. The second creates a button that contains *str* as a label. After a button has been created, we can set its label by calling setLabel( ). We can retrieve its label by calling getLabel( ). These methods are as follows:

```
void setLabel(String str)
String getLabel( )
```

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

```
// Demonstrate Buttons
import java.awt.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet
{
        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);
        }
        public void paint(Graphics g)
        {
                g.drawString(msg, 6, 100);
        }
}
```

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. We 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)
```

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose

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

In order to retrieve the current state of a check box, call getState( ). For setting its state, call setState( ). We can obtain the current label associated with a check box by calling getLabel( ). For setting the label, setLabel( ) is used. These methods are as follows:

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

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

```
// Demonstrate check boxes.
import java.awt.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet
{
    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);
    }
    public void paint(Graphics g)
    {
    }
}
```

Checkbox Group

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, because they act like the station selector on a car radio—only one station can be selected at any one time. For creating a set of mutually exclusive check boxes, we must first define the group to which they will belong and then specify that group when we construct the check boxes. Check box groups are objects of type CheckboxGroup. Only the default constructor is defined, which creates an empty group.

We can determine which check box in a group is currently selected by calling getSelectedCheckbox( ). We can set a check box by calling setSelectedCheckbox( ). These methods are as follows:

```
Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox wh)
```

Here, *wh* is the check box that we want to be selected. The previously selected check box will be turned off. Here is a program that uses check boxes that are part of a group:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/
public class CBGroup extends Applet
{
```

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

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. When inactive, a Choice component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines

the default constructor, which creates an empty list. In order to add a selection to the list, add( ) is used. It has this general form:

```
void add(String name)
```

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur. In order to determine which item is currently selected, we may call either any of the following methods:

```
String getSelectedItem( )
int getSelectedIndex( )
```

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected. For obtaining the number of items in the list, call getItemCount( ). We can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

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

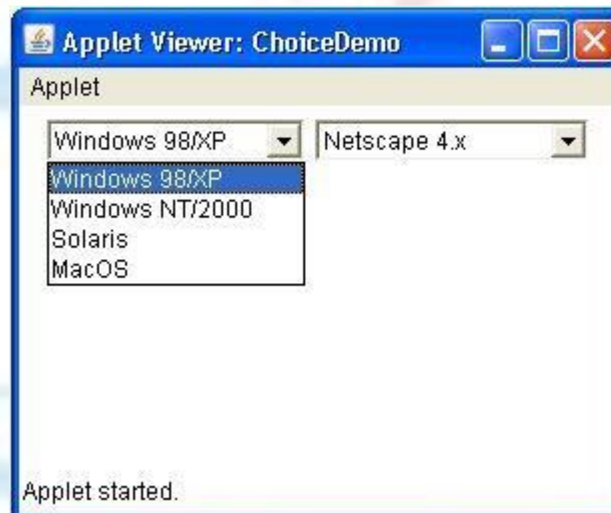
Given an index, we can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

```
String getItem(int index)
```

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

```
// Demonstrate Choice lists.
import java.awt.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet
{
    Choice os, browser;
    String msg = "";
    public void init()
    {
        os = new Choice();
        browser = new Choice();
        os.add("Windows 98/XP");
        os.add("Windows NT/2000");
```

```
        os.add("Solaris");
        os.add("MacOS");
        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(os);
        add(browser);
    }
    public void paint(Graphics g)
    {
    }
}
```



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)
```

The first version creates a List control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies

the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is true, then the user may select two or more items at a time. If it is false, then only one item may be selected. For adding a selection to the list, we can call add( ). It has the following two forms:

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

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. We can specify –1 to add the item to the end of the list. For lists that allow only single selection, we can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

```
String getSelectedItem( )
int getSelectedIndex( )
```

The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, null is returned. getSelectedIndex( ) returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned. For lists that allow multiple selection, we must use either getSelectedItems( ) or getSelectedIndexes( ), shown here, to determine the current selections:

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

The getSelectedItems( ) returns an array containing the names of the currently selected items. getSelectedIndexes( ) returns an array containing the indexes of the currently selected items. In order to obtain the number of items in the list, call getItemCount( ). We can set the currently selected item by using the select( ) method with a zero-based integer index. These methods are shown here:

```
int getItemCount( )
void select(int index)
```

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

String getItem(int index)

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

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet
{
    List os, browser;
    String msg = "";
    public void init()
    {
        os = new List(4, true);
        browser = new List(4, false);
        os.add("Windows 98/XP");
        os.add("Windows NT/2000");
        os.add("Solaris");
        os.add("MacOS");
        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(os);
        add(browser);
    }
    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);
    }
}
```

Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that we can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

```
Scrollbar( )
Scrollbar(int style)
Scrollbar(int style, int iValue, int tSize, int min, int max)
```

The first form creates a vertical scroll bar. The second and third forms allow us to specify the orientation of the scroll bar. If *style* is Scrollbar.VERTICAL, a vertical scroll bar is created. If *style* is Scrollbar.HORIZONTAL, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *iValue*. The number of units represented by the height of the thumb is passed in *tSize*. The minimum and maximum values for the scroll bar are specified by min and max. If we construct a scroll bar by using one of the first two constructors, then we need to set its parameters by using setValues( ), shown here, before it can be used:

```
void setValues(int iValue, int tSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described. In order to obtain the current value of the scroll bar, call getValue( ). It returns the current setting. For setting the current value, we can use setValue( ). These methods are as follows:

```
int getValue( )
void setValue(int newValue)
```

Here, *newValue* specifies the new value for the scroll bar. When we set a value, the slider box inside the scroll bar will be positioned to reflect the new value. We can also retrieve the minimum and maximum values via getMinimum( ) and getMaximum( ), shown here:

```
int getMinimum( )
int getMaximum( )
```

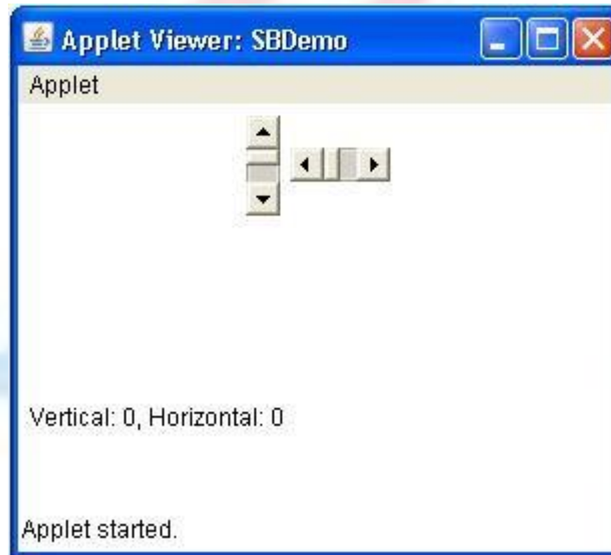
They return the requested quantity. By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. We can change this increment by calling setUnitIncrement( ). By default, page-up and page-down increments are 10. You can change this value by calling setBlockIncrement( ). These methods are shown here:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

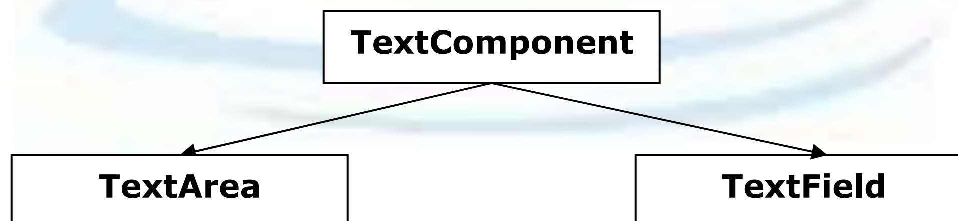
<u>Example</u>:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet
{
    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);
    }
    public void paint(Graphics g)
    {
        msg = "Vertical: " + vertSB.getValue();
        msg += ", Horizontal: " + horzSB.getValue();
        g.drawString(msg, 6, 160);
    }
}
```



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 keys, 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)
```

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width. TextField (and its superclass TextComponent) provides several methods that allow us to utilize a text field. In order to obtain the string currently contained in the text field, we can use getText( ). For setting the text, we call setText( ). These methods are as follows:

```
String getText( )
void setText(String str)
```

Here, *str* is the new string. The user can select a portion of the text in a text field. Also, we can select a portion of text under program control by using select( ). Our program can obtain the currently selected text by calling the getSelectedText( ). These methods are shown here:

```
String getSelectedText( )
void select(int startIndex, int endIndex)
```

The getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at *startIndex* and ending at *endIndex*–1. We can control whether the contents of a text field may be modified by the user by calling setEditable( ). We can determine editability by calling isEditable( ). These methods are shown here:

```
boolean isEditable( )
void setEditable(boolean canEdit)
```

The isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when we will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). We can check a text field to see if it is in this mode with the echoCharIsSet( ) method. We can retrieve the echo character by calling the getEchoChar( ) method. These methods are as follows:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Here, *ch* specifies the character to be echoed.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet
{
    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);
    }
    public void paint(Graphics g)
    {
    }
}
```

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)
```

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

SCROLLBARS_BOTH SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY SCROLLBARS_VERTICAL_ONLY

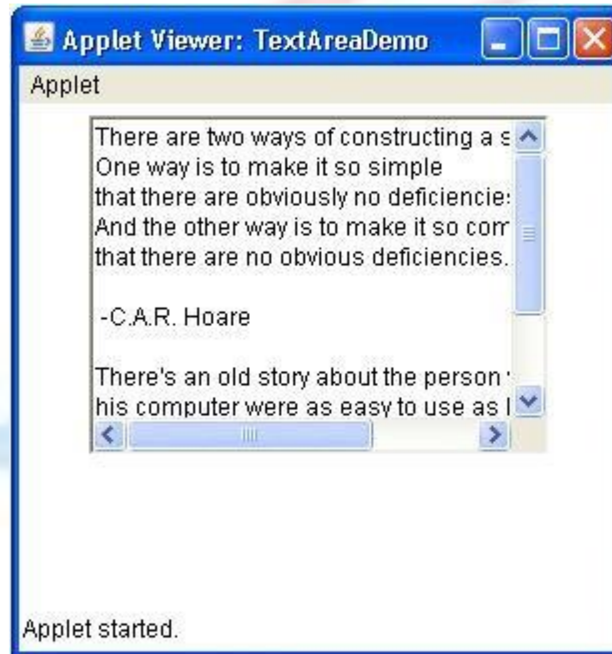
TextArea is a subclass of TextComponent. Therefore, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ), and setEditable( ) methods described in the preceding section. TextArea adds the following methods:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

The append( ) method appends the string specified by *str* to the end of the current text. The insert( ) inserts the string passed in *str* at the specified index. In order to replace text, we call replaceRange( ). It replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*. Text areas are almost self-contained controls.

```
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);
    }
}
```



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 our controls within a window by using some type of algorithm. If we have programmed for other GUI environments, such as Windows, then we are accustomed to laying out our controls by hand. While it is possible to lay out Java controls by hand, too, we generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when we need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used. Whenever a

container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

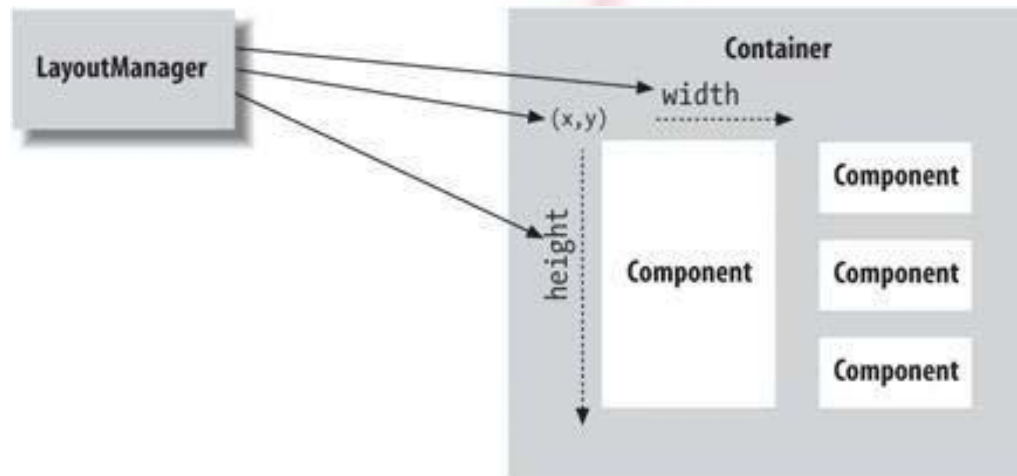


Fig. Layout Managers at work (Ref. No. 2)

The setLayout( ) method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

Here, *layoutObj* is a reference to the desired layout manager. If we wish to disable the layout manager and position components manually, pass null for *layoutObj*. If we do this, we will need to determine the shape and position of each component manually, using the setBounds( ) method defined by Component. Normally, we will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time we add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its minimumLayoutSize( ) and preferredLayoutSize( ) methods. Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ) methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. We may override these methods for controls that we subclass. Default values are provided otherwise. Java has several predefined LayoutManager classes, several of which are described next. We can use the layout manager that best fits our application.

FlowLayout

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more

components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for FlowLayout:

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

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

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

These values specify left, center, and right alignment, respectively. The third form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Here is a version of the CheckboxDemo applet shown earlier, modified so that it uses left-aligned flow layout.

```
public class FlowLayoutDemo extends Applet
{
    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");
        setLayout(new FlowLayout(FlowLayout.CENTER));
        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);
    }
    public void paint(Graphics g)
    {
    }
}
```

BorderLayout

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

```
BorderLayout( )
BorderLayout(int horz, int vert)
```

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

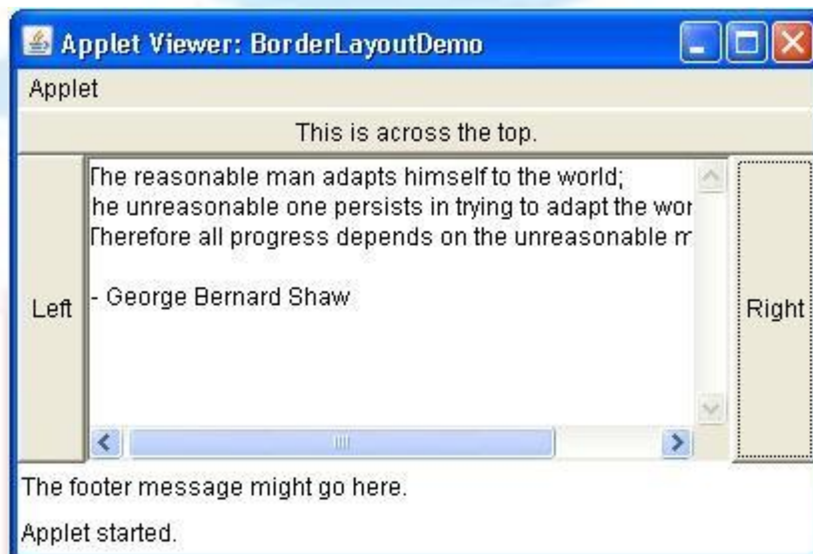
```
BorderLayout.CENTER        BorderLayout.SOUTH
BorderLayout.EAST          BorderLayout.WEST
BorderLayout.NORTH
```

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

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

Here, *compObj* is the component to be added, and region specifies where the component will be added. Here is an example of a BorderLayout with a component in each layout area:

```
import java.awt.*;
import java.applet.*;
/*
<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);
    }
}
```



Insets

Sometimes we will want to leave a small amount of space between the container that holds our components and the window that contains it. For doing this, we have to override the getInsets( ) method that is defined by Container. This function returns an Insets object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for Insets is shown here:

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

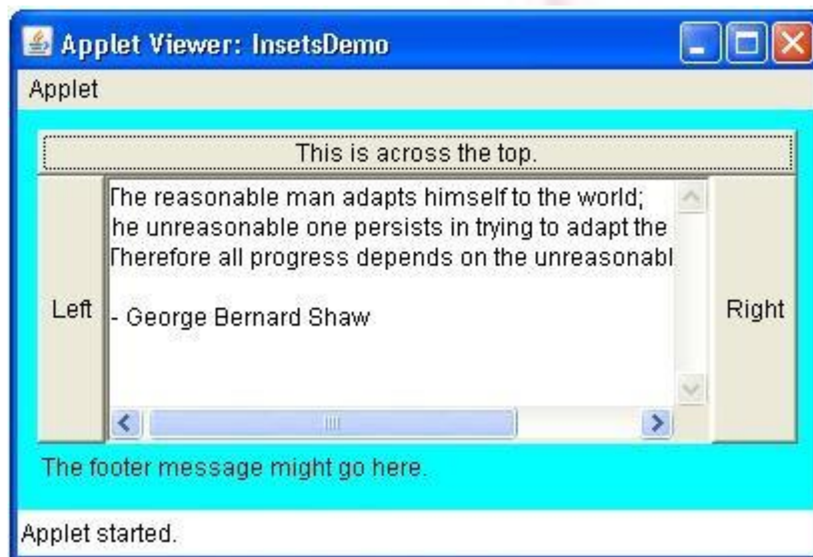
The values passed in top, left, bottom, and right specify the amount of space between the container and its enclosing window. The getInsets( ) method has this general form:

```
Insets getInsets( )
```

When overriding one of these methods, we must return a new Insets object that contains the inset spacing we desire. Here is the preceding BorderLayout example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.

```
public class InsetsDemo extends Applet
{
    public void init()
    {
        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);
    }
    public Insets getInsets()
    {
        return new Insets(10, 10, 10, 10);
    }
```

```
}
```



GridLayout

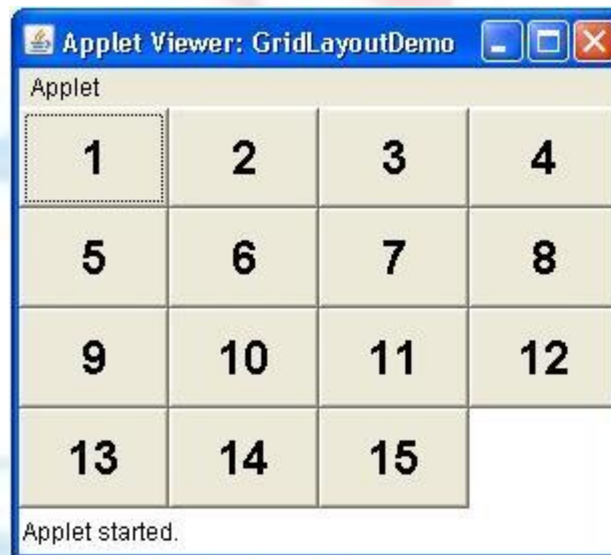
GridLayout lays out components in a two-dimensional grid. When we instantiate a GridLayout, we define the number of rows and columns. The constructors supported by GridLayout are shown here:

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

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either numRows or numColumns can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows. Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

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



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. For creating a menu bar, we first create an instance of MenuBar. This class only defines the default constructor. Next, create instances of Menu that will define the selections displayed on the bar.

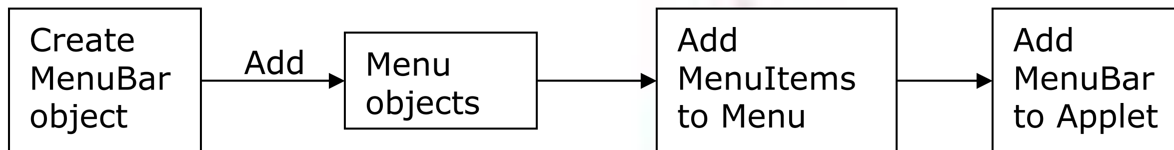


Fig. Creating a menu on Frame

Following are the constructors for Menu:

```
Menu( )
Menu(String optionName)
Menu(String optionName, boolean removable)
```

Here, *optionName* specifies the name of the menu selection. If removable is true, the pop-up menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu. 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 this item. We can disable or enable a menu item by using the setEnabled( ) method. Its form is shown here:

```
void setEnabled(boolean enabledFlag)
```

If the argument *enabledFlag* is true, the menu item is enabled. If false, the menu item is disabled. We can determine an item's status by calling isEnabled( ). This method is shown here:

```
boolean isEnabled( )
```

The isEnabled( ) returns true if the menu item on which it is called is enabled. Otherwise, it returns false. We can change the name of a menu item by calling setLabel( ). We can retrieve the current name by using getLabel( ). These methods are as follows:

```
void setLabel(String newName)
String getLabel( )
```

Here, *newName* becomes the new name of the invoking menu item. getLabel( ) returns the current name. We can create a checkable menu item by

using a subclass of MenuItem called CheckboxMenuItem. It has these constructors:

```
CheckboxMenuItem( )
CheckboxMenuItem(String itemName)
CheckboxMenuItem(String itemName, boolean on)
```

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if on is true, the checkable entry is initially checked. Otherwise, it is cleared. We can obtain the status of a checkable item by calling getState( ). We can set it to a known state by using setState( ). These methods are shown here:

```
boolean getState( )
void setState(boolean checked)
```

If the item is checked, getState( ) returns true. Otherwise, it returns false. For checking an item, pass true to setState( ). To clear an item, pass false. Once we have created a menu item, we must add the item to a Menu object by using add( ), which has the following general form:

```
MenuItem add(MenuItem item)
```

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to add( ) take place. The item is returned. Once we have added all items to a Menu object, we can add that object to the menu bar by using this version of add( ) defined by MenuBar:

```
Menu add(Menu menu)
```

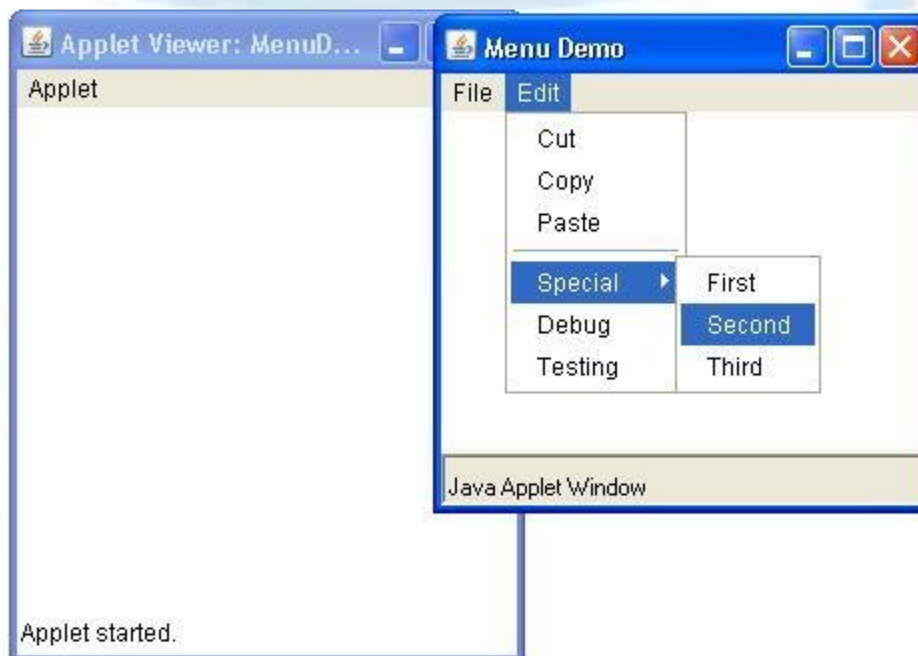
Here, menu is the menu being added. The menu is returned. Menus only generate events when an item of type MenuItem or CheckboxMenuItem is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an ActionEvent object is generated. Each time a check box menu item is checked or unchecked, an ItemEvent object is generated. Thus, we must implement the ActionListener and ItemListener interfaces in order to handle these menu events. The getItem( ) method of ItemEvent returns a reference to the item that generated this event. The general form of this method is shown here:

```
Object getItem( )
```

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/
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);
    }
}
public class MenuDemo extends Applet
{
```

```
    Frame f;
    public void init()
    {
        f = new MenuFrame("Menu Demo");
        f.setVisible(true);
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        setSize(width, height);
        f.setSize(width, height);
    }
    public void start()
    {
        f.setVisible(true);
    }
    public void stop()
    {
        f.setVisible(false);
    }
}
```



Dialog Boxes

Often, we will want to use a dialog box to hold a set of related controls. Dialog boxes are primarily used to obtain user input. They are similar to frame windows, except that dialog boxes are always child windows of a top-level window. Also, dialog boxes don't have menu bars. In other respects, dialog boxes function like frame windows. (We can add controls to them, for example, in the same way that we add controls to a frame window.) 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 we cannot access other parts of our program until we have closed the dialog box. When a modeless dialog box is active, input focus can be directed to another window in our program. Thus, other parts of our 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)
```

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

FileDialog

Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type *FileDialog*. This causes a file dialog box to be displayed. Usually, this is the standard file dialog box provided by the operating system. FileDialog provides these constructors:

```
FileDialog(Frame parent, String boxName)
FileDialog(Frame parent, String boxName, int how)
FileDialog(Frame parent)
```

Here, *parent* is the owner of the dialog box, and *boxName* is the name displayed in the box's title bar. If *boxName* is omitted, the title of the dialog box is empty. If *how* is FileDialog.LOAD, then the box is selecting a file for reading. If *how* is FileDialog.SAVE, the box is selecting a file for writing. The third constructor creates a dialog box for selecting a file for reading.

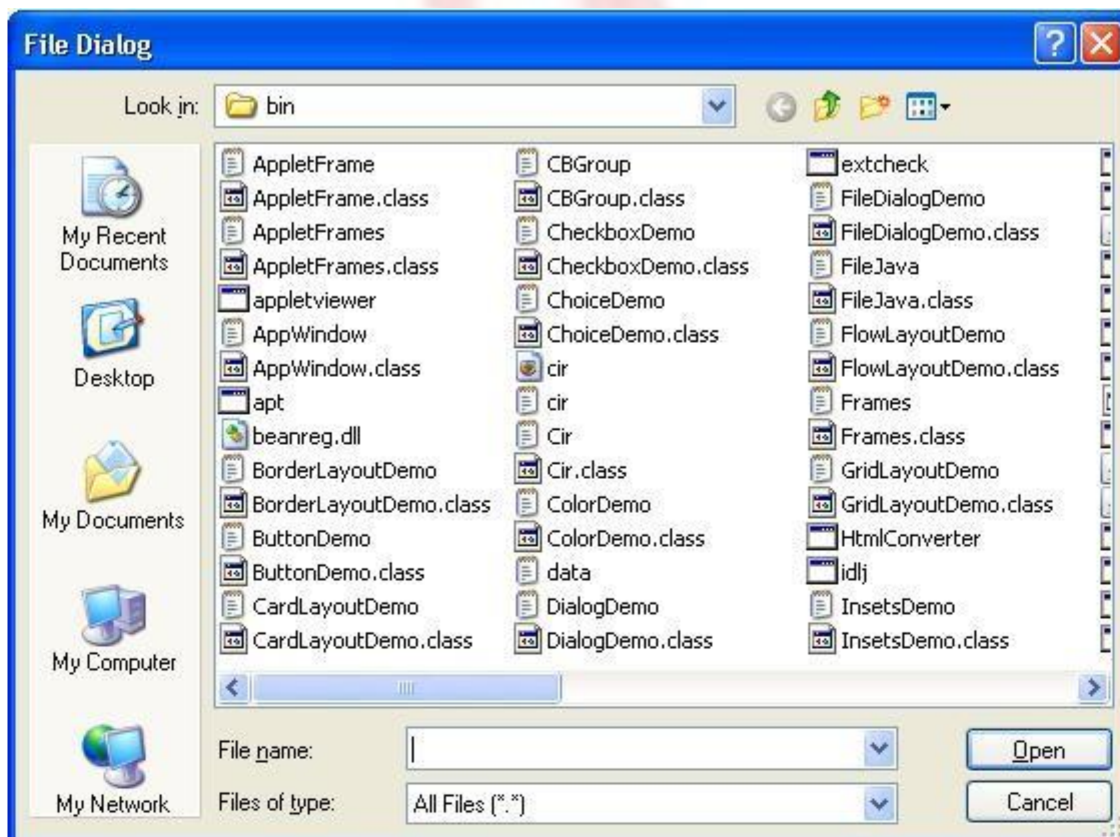
FileDialog( ) provides methods that allows us to determine the name of the file and its path as selected by the user. Here are two examples:

```
String getDirectory( )
String getFile( )
```

These methods return the directory and the filename, respectively. The following program activates the standard file dialog box:

```
import java.awt.*;
class SampleFrame extends Frame
{
    SampleFrame(String title)
    {
        super(title);
```

```
    }
}
class FileDialogDemo
{
    public static void main(String args[])
    {
        Frame f = new SampleFrame("File Dialog Demo");
        f.setVisible(true);
        f.setSize(100, 100);
        FileDialog fd = new FileDialog(f, "File Dialog");
        fd.setVisible(true);
    }
}
```



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

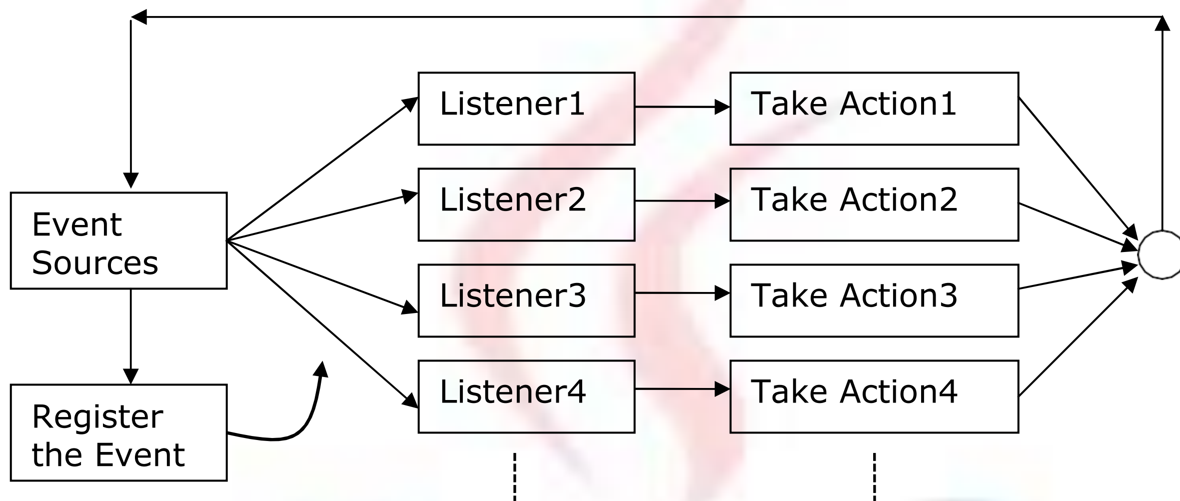


Fig. Delegation Event Model

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: <u>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.</u> 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
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.

MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
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:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.

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

The 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:

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

In addition, ItemEvent defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.
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.

MouseWheelEvent 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:

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

WINDOW_STATE_CHANGED The state of the window changed.

WindowEvent is a subclass of ComponentEvent. 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.)

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

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 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.

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)
```

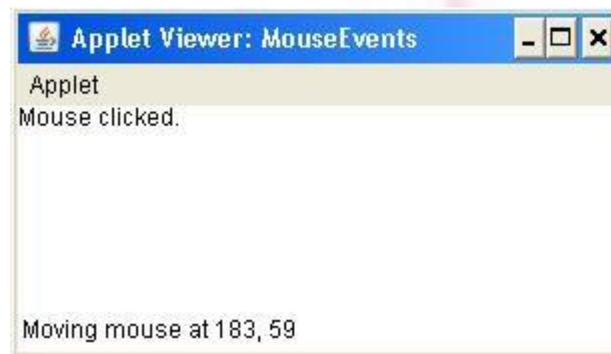
Handling Mouse Events

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.

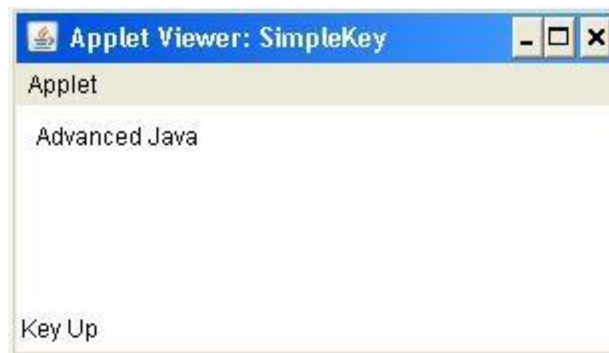
Handling Keyboard Events

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 MyMouseAdapter 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. MyMouseAdapter implements the mouseClicked( ) method. The other mouse events are silently ignored by code inherited from the MouseAdapter 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**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

```
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

MouseAdapter. The init( ) method of MousePressedDemo instantiates MyMouseAdapter 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 MyMouseAdapter 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 MyMouseAdapter.

```
// 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 MyMouseAdapter(this));
	}
}
class MyMouseAdapter extends MouseAdapter
{
	MousePressedDemo mousePressedDemo;
	public MyMouseAdapter(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. MyMouseAdapter is an inner class that extends MouseAdapter. Because MyMouseAdapter 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 MyMouseAdapter( ) 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:

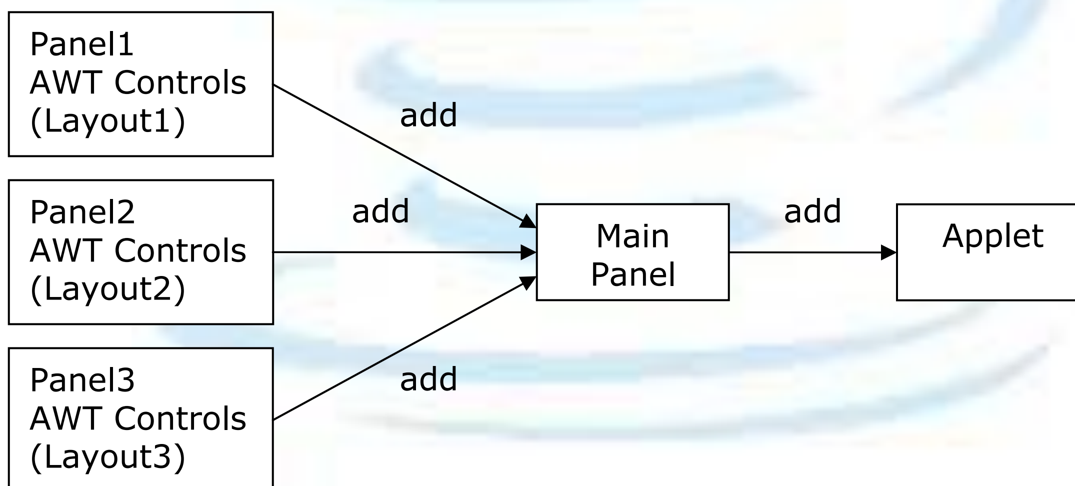


Fig. Creation of card layout

```
// 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:

ACTION_EVENT_MASK	KEY_EVENT_MASK
ADJUSTMENT_EVENT_MASK	MOUSE_EVENT_MASK
COMPONENT_EVENT_MASK	MOUSE_MOTION_EVENT_MASK
CONTAINER_EVENT_MASK	MOUSE_WHEEL_EVENT_MASK
FOCUS_EVENT_MASK	TEXT_EVENT_MASK

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);
        }
    }
}
```

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