

CHAPTER

25

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

The Abstract Window Toolkit (AWT) was Java's first GUI framework, and it has been part of Java since version 1.0. It contains numerous classes and methods that allow you to create windows and simple controls. The AWT was introduced in Chapter 23, where it was used in several short, example applets. This chapter begins a more detailed examination. Here, you will learn how to create and manage windows, manage fonts, output text, and utilize graphics. Chapter 26 describes various AWT controls, such as scroll bars and push buttons. It also explains further aspects of Java's event handling mechanism. Chapter 27 introduces the AWT's imaging subsystem.

It is important to state at the outset that you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. Despite this fact, the AWT remains an important part of Java. To understand why, consider the following.

At the time of this writing, the framework that is most widely used is Swing. Because Swing provides a richer, more flexible GUI framework than does the AWT, it is easy to jump to the conclusion that the AWT is no longer relevant—that it has been fully superseded by Swing. This assumption is, however, false. Instead, an understanding of the AWT is still important because the AWT underpins Swing, with many AWT classes being used either directly or indirectly by Swing. As a result, a solid knowledge of the AWT is still required to use Swing effectively.

Java's newest GUI framework is JavaFX. It is anticipated that, at some point in the future, JavaFX will replace Swing as Java's most popular GUI. Even when this occurs, however, much legacy code that relies on Swing (and thus, the AWT) will still need to be maintained for some time to come. Finally, for some types of small programs (especially small applets) that make only minimal use of a GUI, using the AWT may still be appropriate. Therefore, even though the AWT constitutes Java's oldest GUI framework, a basic working knowledge of its fundamentals is still important today.

Although a common use of the AWT is in applets, it is also used to create stand-alone windows that run in a GUI environment, such as Windows. For the sake of convenience, most of the examples in this chapter are contained in applets. The easiest way to run them

is with the applet viewer. A few examples demonstrate the creation of stand-alone, windowed programs, which can be executed directly.

One last point before beginning: The AWT is quite large and a full description would easily fill an entire book. Therefore, it is not possible to describe in detail every AWT class, method, or instance variable. However, this and the following chapters explain the basic techniques needed to use the AWT. From there, you will be able to explore other parts of the AWT on your own. You will also be ready to move on to Swing.

NOTE If you have not yet read Chapter 24, please do so now. It provides an overview of event handling, which is used by many of the examples in this chapter.

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. Table 25-1 lists some of the many AWT classes.

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

Table 25-1 A Sampling of AWT Classes

Class	Description
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.

Table 25-1 A Sampling of AWT Classes (continued)

Class	Description
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, and no title.

Table 25-1 A Sampling of AWT Classes (*continued*)

Although the basic structure of the AWT has been the same since Java 1.0, some of the original methods were deprecated and replaced by new ones. For backward-compatibility, Java still supports all the original 1.0 methods. However, because these methods are not for use with new code, this book does not describe them.

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 application 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 25-1 shows the class hierarchy for **Panel** and **Frame**. Let’s look at each of these classes now.

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. Except for menus, all user interface elements that are displayed on the screen and that interact with the user are

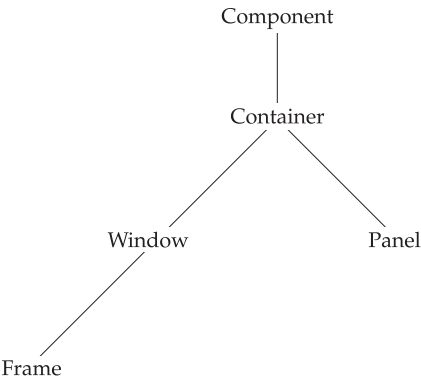


Figure 25-1 The class hierarchy for **Panel** and **Frame**

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. (You already used many of these methods when you created applets in Chapters 23 and 24.) 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, which you will learn about in Chapter 26.

Panel

The **Panel** class is a concrete subclass of **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass 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 you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add()** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation()**, **setSize()**, **setPreferredSize()**, 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, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

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. The precise look of a **Frame** will differ among environments. A number of environments are reflected in the screen captures shown throughout this book.

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. Derived from **Component**, **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

Working with Frame Windows

In addition to the applet, the type of AWT-based window you will most often create is derived from **Frame**. You will use it to create child windows within applets, and top-level or child windows for stand-alone applications. As mentioned, it creates a standard-style window.

Here are two of **Frame**'s constructors:

```
Frame( ) throws HeadlessException
Frame(String title) throws HeadlessException
```

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*. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created. A **HeadlessException** is thrown if an attempt is made to create a **Frame** instance in an environment that does not support user interaction.

There are several key methods you will use when working with **Frame** windows. They are examined here.

Setting the Window's Dimensions

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

```
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. One of its forms is shown here:

```
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 you call **setVisible()**. Its signature is shown here:

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

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

Setting a Window's Title

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

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

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

Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed, by calling `setVisible(false)`. To intercept a window-close event, you must implement the `windowClosing()` method of the `WindowListener` interface. Inside `windowClosing()`, you must remove the window from the screen. The example in the next section illustrates this technique.

Creating a Frame Window in an AWT-Based Applet

While it is possible to simply create a window by creating an instance of `Frame`, you will seldom do so, because you will not be able to do much with it. For example, you will not be able to receive or process events that occur within it or easily output information to it. Most of the time, you will create a subclass of `Frame`. Doing so lets you override `Frame`'s methods and provide event handling.

Creating a new frame window from within an AWT-based applet is actually quite easy. First, create a subclass of `Frame`. Next, override any of the standard applet methods, such as `init()`, `start()`, and `stop()`, to show or hide the frame as needed. Finally, implement the `windowClosing()` method of the `WindowListener` interface, calling `setVisible(false)` when the window is closed.

Once you have defined a `Frame` subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling `setVisible()`. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the `setSize()` method.

The following applet creates a subclass of `Frame` called `SampleFrame`. A window of this subclass is instantiated within the `init()` method of `AppletFrame`. Notice 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's `start()` and `stop()` methods so that they show and hide the child window, respectively. This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AppletFrame" width=300 height=50>
  </applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);

        // 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("This is in frame window", 10, 40);
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;

    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }

    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Create frame window.
public class AppletFrame extends Applet {
    Frame f;
    public void init() {
        f = new SampleFrame("A Frame Window");

        f.setSize(250, 250);
        f.setVisible(true);
    }

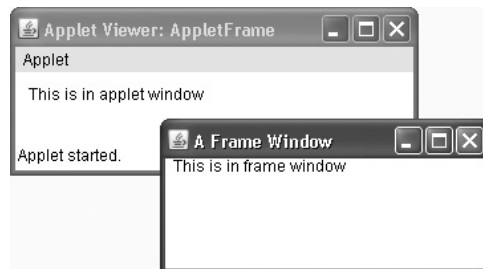
    public void start() {
        f.setVisible(true);
    }

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

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

```

Sample output from this program is shown here:



Handling Events in a Frame Window

Since **Frame** is a subclass of **Component**, it inherits all the capabilities defined by **Component**. This means that you can use and manage a frame window just like you manage an applet's main window, as described earlier in this book. For example, you can override **paint()** to display output, call **repaint()** when you need to restore the window, and add event handlers. Whenever an event occurs in a window, the event handlers defined by that window will be called. Each window handles its own events. For example, the following program creates a window that responds to mouse events. The main applet window also responds to mouse events. When you experiment with this program, you will see that mouse events are sent to the window in which the event occurs.

```
// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="WindowEvents" width=300 height=50>
  </applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame
    implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX=10, mouseY=40;
    int movX=0, movY=0;

    SampleFrame(String title) {
        super(title);
        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // register it to receive those events
        addWindowListener(adapter);
    }

    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
    }

    // Handle mouse entered.
    public void mouseEntered(MouseEvent evtObj) {
        // save coordinates
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just entered child.";
        repaint();
    }
}
```

```

    // Handle mouse exited.
    public void mouseExited(MouseEvent evtObj) {
        // save coordinates
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just left child window.";
        repaint();
    }

    // Handle mouse pressed.
    public void mousePressed(MouseEvent me) {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down";
        repaint();
    }

    // Handle mouse 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();
        movX = me.getX();
        movY = me.getY();
        msg = "**";
        repaint();
    }

    // Handle mouse moved.
    public void mouseMoved(MouseEvent me) {
        // save coordinates
        movX = me.getX();
        movY = me.getY();
        repaint(0, 0, 100, 60);
    }

    public void paint(Graphics g) {
        g.drawString(msg, mouseX, mouseY);
        g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;

```

```
public MyWindowAdapter(SampleFrame sampleFrame) {
    this.sampleFrame = sampleFrame;
}

public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
}
}

// Applet window.
public class WindowEvents extends Applet
    implements MouseListener, MouseMotionListener {

    SampleFrame f;
    String msg = "";
    int mouseX=0, mouseY=10;
    int movX=0, movY=0;

    // Create a frame window.
    public void init() {
        f = new SampleFrame("Handle Mouse Events");
        f.setSize(300, 200);
        f.setVisible(true);

        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // Remove frame window when stopping applet.
    public void stop() {
        f.setVisible(false);
    }

    // Show frame window when starting applet.
    public void start() {
        f.setVisible(true);
    }

    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
    }

    // Handle mouse entered.
    public void mouseEntered(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 24;
        msg = "Mouse just entered applet window.";
        repaint();
    }
}
```

```

// Handle mouse exited.
public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
    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();
}

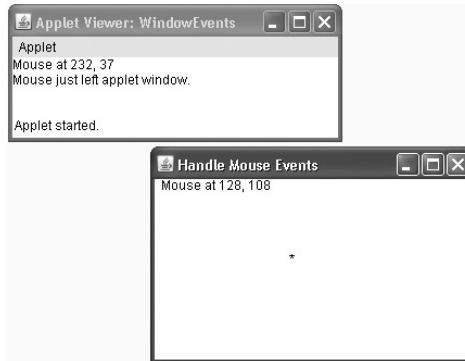
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "**";
    repaint();
}

// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
    // save coordinates
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 20);
}

// Display msg in applet window.
public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
}
}

```

Sample output from this program is shown here:



Creating a Windowed Program

Although creating applets is a common use for Java's AWT, it is also possible to create stand-alone AWT-based applications. To do this, simply create an instance of the window or windows you need inside `main()`. For example, the following program creates a frame window that responds to mouse clicks and keystrokes:

```
// Create an AWT-based application.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Create a frame window.
public class AppWindow extends Frame {
    String keymsg = "This is a test.";
    String mousemsg = "";
    int mouseX=30, mouseY=30;

    public AppWindow() {
        addKeyListener(new MyKeyAdapter(this));
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }

    public void paint(Graphics g) {
        g.drawString(keymsg, 10, 40);
        g.drawString(mousemsg, mouseX, mouseY);
    }

    // Create the window.
    public static void main(String args[]) {
        AppWindow appwin = new AppWindow();

        appwin.setSize(new Dimension(300, 200));
        appwin.setTitle("An AWT-Based Application");
        appwin.setVisible(true);
    }
}
```

```

class MyKeyAdapter extends KeyAdapter {
    AppWindow appWindow;

    public MyKeyAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void keyTyped(KeyEvent ke) {
        appWindow.keymsg += ke.getKeyChar();
        appWindow.repaint();
    };
}

class MyMouseAdapter extends MouseAdapter {
    AppWindow appWindow;

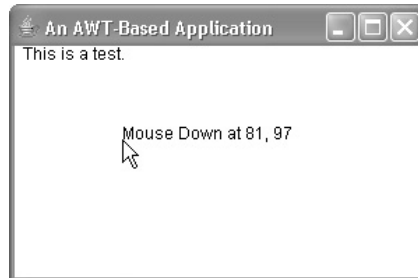
    public MyMouseAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void mousePressed(MouseEvent me) {
        appWindow.mouseX = me.getX();
        appWindow.mouseY = me.getY();
        appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
                               ", " + appWindow.mouseY;
        appWindow.repaint();
    }
}

class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}

```

Sample output from this program is shown here:



Once created, a frame window takes on a life of its own. Notice that **main()** ends with the call to **appwin.setVisible(true)**. However, the program keeps running until you close the window. In essence, when creating a windowed application, you will use **main()** to launch its top-level window. After that, your program will function as a GUI-based application, not like the console-based programs used earlier.

CHAPTER

26

Using AWT Controls, Layout Managers, and Menus

This chapter continues our overview of the Abstract Window Toolkit (AWT). It begins with a look at several of the AWT's controls and layout managers. It then discusses menus and the menu bar. The chapter also includes a discussion of two high-level components: the dialog box and the file dialog box. It concludes with another look at event handling.

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

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. This constitutes the *main menu* of an application. As a general rule, a menu bar is 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. For the first part of this chapter, which introduces various controls, the default layout manager will be used. This displays components in a container using left-to-right, top-to-bottom organization. Once the controls have been covered, several layout managers will be examined. There, you will see ways to better manage the positioning of controls.

Before continuing, it is important to emphasize that today you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. However, the material presented here remains important for the following reasons. First, much of the information and many of the techniques related to controls and event handling are generalizable to the other Java GUI frameworks. (As mentioned in the previous chapter, Swing is built upon the AWT.) Second, the layout managers described here can also be used by Swing. Third, for some small applications, the AWT components might be the appropriate choice. Finally, and perhaps most importantly, you may need to maintain or upgrade legacy code that uses the AWT. Therefore, a basic understanding of the AWT is important for all Java programmers.

AWT Control Fundamentals

The AWT supports the following types of controls:

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

These controls are subclasses of **Component**. Although this is not a particularly rich set of controls, it is sufficient for simple applications. (Note that both Swing and JavaFX provide a substantially larger, more sophisticated set of controls.)

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add()**, which is defined by **Container**. The **add()** method has several forms. The following form is the one that is used for the first part of this chapter:

```
Component add(Component compRef)
```

Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

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

```
void remove(Component compRef)
```

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

Responding to Controls

Except for labels, which are passive, all other 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, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 24, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

The HeadlessException

Most of the AWT controls described in this chapter have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). You can use this exception to write code that can adapt to non-interactive environments. (Of course, this is not always possible.) This exception is not handled by the programs in this chapter because an interactive environment is required to demonstrate the AWT controls.

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

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

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

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

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 window:

```
// 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);
    }
}

```

Here is sample output from the **LabelDemo** applet. Notice that the labels are organized in the window by the default layout manager. Later, you will see how to control more precisely the placement of the labels.



Using Buttons

Perhaps 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( ) throws HeadlessException
Button(String str) throws HeadlessException

```

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

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

```

void setLabel(String str)
String getLabel( )

```

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

Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed()** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Either the button reference or the action command string can be used to identify the button. (You will soon see examples of each approach.)

Here is an example that creates three buttons labeled "Yes", "No", and "Undecided". Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the **getActionCommand()** method on the **ActionEvent** object passed to **actionPerformed()**.

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

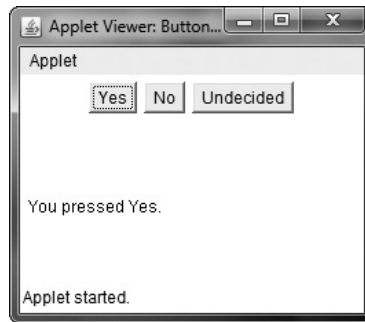


Figure 26-1 Sample output from the **ButtonDemo** applet

Sample output from the **ButtonDemo** program is shown in Figure 26-1.

As mentioned, in addition to comparing button action command strings, you can also determine which button has been pressed by comparing the object obtained from the **getSource()** method to the button objects that you added to the window. To do this, you must keep a list of the objects when they are added. The following applet shows this approach:

```
// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
    <applet code="ButtonList" width=250 height=150>
</applet>
*/

public class ButtonList extends Applet implements ActionListener {
    String msg = "";
    Button bList[] = new Button[3];

    public void init() {
        Button yes = new Button("Yes");
        Button no = new Button("No");
        Button maybe = new Button("Undecided");

        // store references to buttons as added
        bList[0] = (Button) add(yes);
        bList[1] = (Button) add(no);
        bList[2] = (Button) add(maybe);

        // register to receive action events
        for(int i = 0; i < 3; i++) {
            bList[i].addActionListener(this);
        }
    }
}
```

```

public void actionPerformed(ActionEvent ae) {
    for(int i = 0; i < 3; i++) {
        if(ae.getSource() == bList[i]) {
            msg = "You pressed " + bList[i].getLabel();
        }
    }
    repaint();
}

public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
}
}

```

In this version, the program stores each button reference in an array when the buttons are added to the applet window. (Recall that the **add()** method returns a reference to the button when it is added.) Inside **actionPerformed()**, this array is then used to determine which button has been pressed.

For simple programs, it is usually easier to recognize buttons by their labels. However, in situations in which you will be changing the label inside a button during the execution of your program, or using buttons that have the same label, it may be easier to determine which button has been pushed by using its object reference. It is also possible to set the action command string associated with a button to something other than its label by calling **setActionCommand()**. This method changes the action command string, but does not affect the string used to label the button. Thus, setting the action command enables the action command and the label of a button to differ.

In some cases, you can handle the action events generated by a button (or some other type of control) by use of an anonymous inner class (as described in Chapter 24) or a lambda expression (discussed in Chapter 15). For example, assuming the previous programs, here is a set of action event handlers that use lambda expressions:

```

// Use lambda expressions to handle action events.
yes.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});

no.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});

maybe.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});

```

This code works because **ActionListener** defines a functional interface, which is an interface with exactly one abstract method. Thus, it can be used by a lambda expression. In general, you can use a lambda expression to handle an AWT event when its listener defines a functional interface. For example, **ItemListener** is also a functional interface. Of course,

whether you use the traditional approach, an anonymous inner class, or a lambda expression will be determined by the precise nature of your application. The remaining examples in this chapter use the traditional approach to event handling so that they can be compiled by nearly any version of Java. However, you might find it interesting to try converting the event handlers to lambda expressions or anonymous inner classes, where appropriate.

Applying Check Boxes

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

Checkbox supports these constructors:

`Checkbox()` throws `HeadlessException`

`Checkbox(String str)` throws `HeadlessException`

`Checkbox(String str, boolean on)` throws `HeadlessException`

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

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

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

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

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

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

Handling Check Boxes

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

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
   <applet code="CheckboxDemo" width=240 height=200>
   </applet>
*/

public class CheckboxDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;

    public void init() {
        windows = new Checkbox("Windows", null, true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        add(windows);
        add(android);
        add(solaris);
        add(mac);

        windows.addItemListener(this);
        android.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: " + windows.getState();
        g.drawString(msg, 6, 100);
        msg = "  Android: " + android.getState();
        g.drawString(msg, 6, 120);
        msg = "  Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = "  Mac OS: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
```

Sample output is shown in Figure 26-2.



Figure 26-2 Sample output from the **CheckboxDemo** applet

CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons*, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.

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

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

Here, *which* is the check box that you want to be selected. The previously selected check box will be turned off.

Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
    <applet code="CBGroup" width=240 height=200>
    </applet>
*/

public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;
    CheckboxGroup cbg;
```



```
public void init() {
    cbg = new CheckboxGroup();
    windows = new Checkbox("Windows", cbg, true);
    android = new Checkbox("Android", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    windows.addItemListener(this);
    android.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);
}
}
```

Sample output generated by the **CBGroup** applet is shown in Figure 26-3. Notice that the check boxes are now circular in shape.



Figure 26-3 Sample output from the **CBGroup** applet

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** defines only the default constructor, which creates an empty list.

To add a selection to the list, call **add()**. It has this general form:

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

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

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

```
String getItem( )
int getSelectedIndex( )
```

The **getSelectedItem()** method returns a string containing the name of the item. **getSelectedIndex()** returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

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

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

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

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

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

Handling Choice Lists

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

Here is an example that creates two **Choice** menus. One selects the operating system. The other selects the browser.

```
// 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");
        os.add("Android");
        os.add("Solaris");
        os.add("Mac OS");

        // add items to browser list
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Chrome");

        // 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);
    }
}
```

Sample output is shown in Figure 26-4.

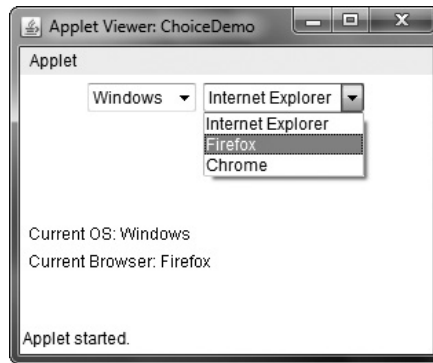


Figure 26-4 Sample output from the **ChoiceDemo** applet

Using 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()` throws `HeadlessException`
`List(int numRows)` throws `HeadlessException`
`List(int numRows, boolean multipleSelect)` throws `HeadlessException`

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

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

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

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

For lists that allow only single selection, you can determine which item is currently selected by calling either `getSelectedItem()` or `getSelectedIndex()`. These methods are shown here:

`String getItemSelected()`
`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, you must use either **getSelectedItems()** or **getSelectedIndexes()**, shown here, to determine the current selections:

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

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

To obtain the number of items in the list, call **getItemCount()**. You 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, you can obtain the name associated with the item at that index by calling **getItem()**, which has this general form:

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

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

Handling Lists

To process list events, you will need to implement the **ActionListener** interface. Each time a **List** item is double-clicked, an **ActionEvent** object is generated. Its **getActionCommand()** method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated. Its **getStateChange()** method can be used to determine whether a selection or deselection triggered this event. **getItemSelectable()** returns a reference to the object that triggered this event.

Here is an example that converts the **Choice** controls in the preceding section into **List** components, one multiple choice and the other single choice:

```
// 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");
```

```

os.add("Android");
os.add("Solaris");
os.add("Mac OS");

// add items to browser list
browser.add("Internet Explorer");
browser.add("Firefox");
browser.add("Chrome");

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);
}
}

```

Sample output generated by the **ListDemo** applet is shown in Figure 26-5.

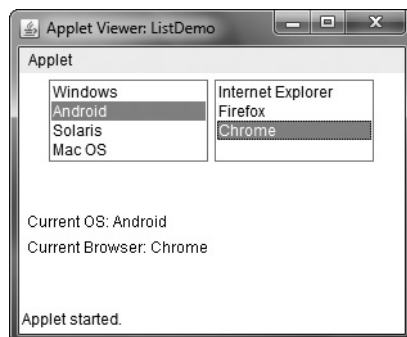


Figure 26-5 Sample output from the **ListDemo** applet

Managing Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you 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( ) throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
    throws HeadlessException
```

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

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

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

The parameters have the same meaning as they have in the third constructor just described.

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

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

Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

You 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. You 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)
```

Handling Scroll Bars

To process scroll bar events, you need to implement the **AdjustmentListener** interface. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType()** method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position. Notice the use of **setPreferredSize()** to set the size of the 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);
        vertSB.setPreferredSize(new Dimension(20, 100));

        horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                               0, 1, 0, width);
        horzSB.setPreferredSize(new Dimension(100, 20));

        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());
}
}

```

Sample output from the **SBDemo** applet is shown in Figure 26-6.

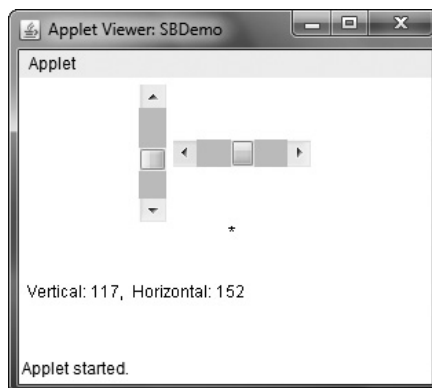


Figure 26-6 Sample output from the **SBDemo** applet

Using a 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( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

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

TextField (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText()**. To set the text, call **setText()**. These methods are as follows:

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

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select()**. Your program can obtain the currently selected text by calling **getSelectedText()**. These methods are shown here:

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

getSelectedText() returns the selected text. The **select()** method selects the characters beginning at *startIndex* and ending at *endIndex* - 1.

You can control whether the contents of a text field may be modified by the user by calling **setEditable()**. You can determine editability by calling **isEditable()**. These methods are shown here:

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

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 you 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). You can check a text field to see if it is in this mode with the **echoCharIsSet()** method. You 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. If *ch* is zero, then normal echoing is restored.

Handling a TextField

Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Here is an example that creates the classic user name and password screen:

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

Sample output from the **TextFieldDemo** applet is shown in Figure 26-7.



Figure 26-7 Sample output from the **TextFieldDemo** applet

Using a TextArea

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

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

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

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

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

TextArea adds the following editing 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. `insert()` inserts the string passed in *str* at the specified index. To replace text, 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. Your program incurs virtually no management overhead. Normally, your program simply obtains the current text when it is needed. You can, however, listen for **TextEvents**, if you choose.

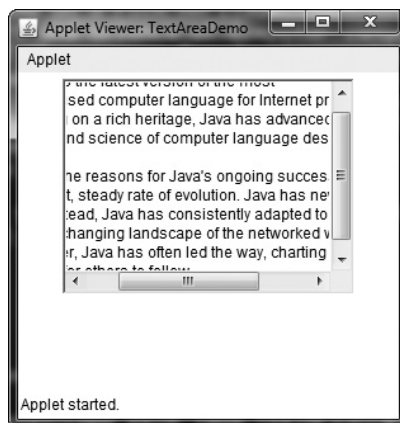
The following program creates a **TextArea** control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/

public class TextAreaDemo extends Applet {
    public void init() {
        String val =
            "Java 8 is the latest version of the most\n" +
            "widely-used computer language for Internet programming.\n" +
            "Building on a rich heritage, Java has advanced both\n" +
            "the art and science of computer language design.\n\n" +
            "One of the reasons for Java's ongoing success is its\n" +
            "constant, steady rate of evolution. Java has never stood\n" +
            "still. Instead, Java has consistently adapted to the\n" +
            "rapidly changing landscape of the networked world.\n" +
            "Moreover, Java has often led the way, charting the\n" +
            "course for others to follow.";

        TextArea text = new TextArea(val, 10, 30);
        add(text);
    }
}
```

Here is sample output from the **TextAreaDemo** applet:



Understanding Layout Managers

All of the components that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. If you have

programmed for other GUI environments, such as Windows, then you may have laid out your controls by hand. While it is possible to lay out Java controls by hand, too, you 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 you 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.

The **setLayout()** method has the following general form:

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

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*. If you do this, you will need to determine the shape and position of each component manually, using the **setBounds()** method defined by **Component**. Normally, you 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 you 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. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined **LayoutManager** classes, several of which are described next. You can use the layout manager that best fits your 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. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to 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 you specify how each line is aligned. Valid values for *how* are as follows:

```
FlowLayout.LEFT  
FlowLayout.CENTER  
FlowLayout.RIGHT  
FlowLayout.LEADING  
FlowLayout.TRAILING
```

These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you 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 in this chapter, modified so that it uses left-aligned flow layout:

```
// Use left-aligned flow layout.  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
  <applet code="FlowLayoutDemo" width=240 height=200>  
  </applet>  
*/  
  
public class FlowLayoutDemo extends Applet  
    implements ItemListener {  
  
    String msg = "";  
    Checkbox windows, android, solaris, mac;  
  
    public void init() {  
        // set left-aligned flow layout  
        setLayout(new FlowLayout(FlowLayout.LEFT));  
  
        windows = new Checkbox("Windows", null, true);  
        android = new Checkbox("Android");  
        solaris = new Checkbox("Solaris");  
        mac = new Checkbox("Mac OS");  
  
        add(windows);  
        add(android);  
        add(solaris);  
        add(mac);  
  
        // register to receive item events  
        windows.addItemListener(this);  
        android.addItemListener(this);  
        solaris.addItemListener(this);  
        mac.addItemListener(this);  
    }  
}
```

```

// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// Display current state of the check boxes.
public void paint(Graphics g) {

    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}

```

Here is sample output generated by the **FlowLayoutDemo** applet. Compare this with the output from the **CheckboxDemo** applet, shown earlier in Figure 26-2.

BorderLayout

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

```

BorderLayout()
BorderLayout(int horz, int vert)

```

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

BorderLayout defines the following constants that specify the regions:

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

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

```
void add(Component compRef, Object region)
```

Here, *compRef* is a reference to 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:

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

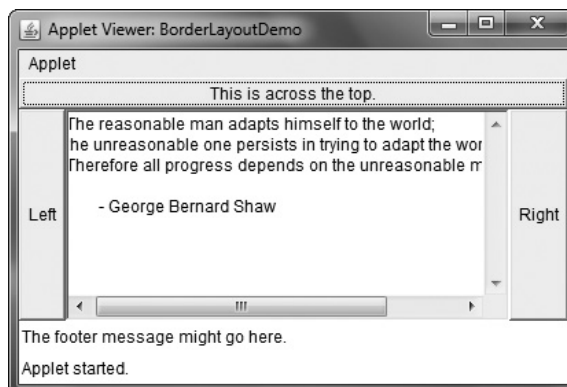
public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());

        add(new Button("This is across the top."),
            BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            "        - George Bernard Shaw\n\n";

        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```

Sample output from the **BorderLayoutDemo** applet is shown here:



Using Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the `getInsets()` method that is defined by **Container**. This method 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 this method, you must return a new **Insets** object that contains the inset spacing you 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.

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

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

        setLayout(new BorderLayout());

        add(new Button("This is across the top."),
            BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            "        - George Bernard Shaw\n\n";
```

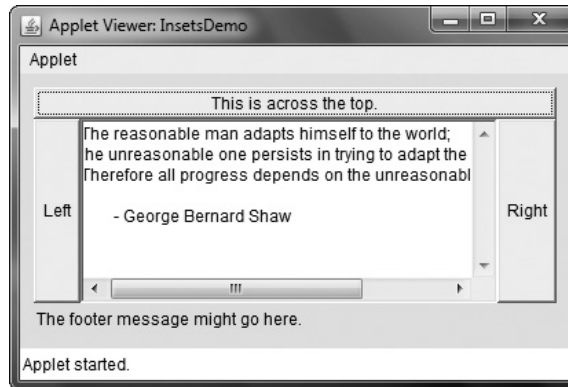
```

    add(new TextArea(msg), BorderLayout.CENTER);
}

// add insets
public Insets getInsets() {
    return new Insets(10, 10, 10, 10);
}
}

```

Sample output from the **InsetsDemo** applet is shown here:



GridLayout

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

```

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

```

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. 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:

```

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

```

```

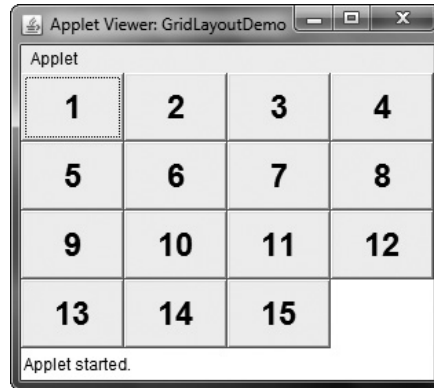
public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));

        setFont(new Font("SansSerif", Font.BOLD, 24));

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

```

Following is sample output generated by the **GridLayoutDemo** applet:



TIP You might try using this example as the starting point for a 15-square puzzle.

CardLayout

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

CardLayout provides these two constructors:

```

CardLayout()
CardLayout(int horz, int vert)

```

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

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add()** when adding cards to a panel:

```
void add(Component panelRef, Object name)
```

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelRef*.

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

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

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first()** causes the first card in the deck to be shown. To show the last card, call **last()**. To show 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. Mac OS and Solaris are displayed in the other card.

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

public class CardLayoutDemo extends Applet
    implements ActionListener, MouseListener {
```

```

Checkbox windowsXP, windows7, windows8, android, 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

    windowsXP = new Checkbox("Windows XP", null, true);
    windows7 = new Checkbox("Windows 7", null, false);
    windows8 = new Checkbox("Windows 8", null, false);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // add Windows check boxes to a panel
    Panel winPan = new Panel();
    winPan.add(windowsXP);
    winPan.add(windows7);
    winPan.add(windows8);

    // Add other OS check boxes to a panel
    Panel otherPan = new Panel();
    otherPan.add(android);
    otherPan.add(solaris);
    otherPan.add(mac);

    // add panels to card deck panel
    osCards.add(winPan, "Windows");
    osCards.add(otherPan, "Other");

    // add cards to main applet panel
    add(osCards);

    // register to receive action events
    Win.addActionListener(this);
    Other.addActionListener(this);

    // register mouse events
    addMouseListener(this);
}

// Cycle through panels.
public void mousePressed(MouseEvent me) {
    cardLO.next(osCards);
}

```

```
// Provide empty implementations for the other MouseListener methods.
public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) {
}
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}

public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {

        cardLO.show(osCards, "Windows");
    }
    else {
        cardLO.show(osCards, "Other");
    }
}
}
```

Here is sample output generated by the **CardLayoutDemo** applet. Each card is activated by pushing its button. You can also cycle through the cards by clicking the mouse.



GridBagLayout

Although the preceding layouts are perfectly acceptable for many uses, some situations will require that you take a bit more control over how the components are arranged. A good way to do this is to use a grid bag layout, which is specified by the **GridBagLayout** class. What makes the grid bag useful is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number

of columns. This is why the layout is called a *grid bag*. It's a collection of small grids joined together.

The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager. Although **GridBagLayout** is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works.

GridBagLayout defines only one constructor, which is shown here:

```
GridBagLayout( )
```

GridBagLayout defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: **setConstraints()**. It is shown here:

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

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.

The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a **GridBagConstraints** object. **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component. These are shown in Table 26-1. Several are described in greater detail in the following discussion.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER .
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are GridBagConstraints.NONE (the default), GridBagConstraints.HORIZONTAL , GridBagConstraints.VERTICAL , GridBagConstraints.BOTH .
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE .
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE .
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.

Table 26-1 Constraint Fields Defined by **GridBagConstraints**

Field	Purpose
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

Table 26-1 Constraint Fields Defined by **GridBagConstraints** (continued)

GridBagConstraints also defines several static fields that contain standard constraint values, such as **GridBagConstraints.CENTER** and **GridBagConstraints.VERTICAL**.

When a component is smaller than its cell, you can use the **anchor** field to specify where within the cell the component's top-left corner will be located. There are three types of values that you can give to **anchor**. The first are absolute:

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

As their names imply, these values cause the component to be placed at the specific locations.

The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages. The relative values are shown here:

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

Their names describe the placement.

The third type of values that can be given to **anchor** allows you to position components relative to the baseline of the row. These values are shown here:

GridBagConstraints.BASELINE	GridBagConstraints.BASELINE_LEADING
GridBagConstraints.BASELINE_TRAILING	GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE_LEADING	GridBagConstraints.ABOVE_BASELINE_TRAILING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_BASELINE_LEADING
GridBagConstraints.BELOW_BASELINE_TRAILING	

The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

The **weightx** and **weighty** fields are both quite important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.

The **gridwidth** variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use **GridBagConstraints.REMAINDER**. To specify that a component use the next-to-last cell in a row, use **GridBagConstraints.RELATIVE**. The **gridheight** constraint works the same way, but in the vertical direction.

You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to **ipadx**. To pad vertically, assign a value to **ipady**.

Here is an example that uses **GridBagLayout** to demonstrate several of the points just discussed:

```
// Use GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
    <applet code="GridBagDemo" width=250 height=200>
    </applet>
*/

public class GridBagDemo extends Applet
    implements ItemListener {

    String msg = "";
    Checkbox windows, android, solaris, mac;

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

        // Define check boxes.
        windows = new Checkbox("Windows ", null, true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        // Define the grid bag.

        // Use default row weight of 0 for first row.
        gbc.weightx = 1.0; // use a column weight of 1
```

```

gbc.ipadx = 200; // pad by 200 units
gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left

gbc.anchor = GridBagConstraints.NORTHEAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(windows, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(android, gbc);

// Give second row a weight of 1.
gbc.weighty = 1.0;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(solaris, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(mac, gbc);

// Add the components.
add(windows);
add(android);
add(solaris);
add(mac);

// Register to receive item events.
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}

// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// Display current state of the check boxes.
public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}

```

Sample output produced by the program is shown here.



In this layout, the operating system check boxes are positioned in a 2×2 grid. Each cell has a horizontal padding of 200. Each component is inset slightly (by 4 units) from the top left. The column weight is set to 1, which causes any extra horizontal space to be distributed evenly between the columns. The first row uses a default weight of 0; the second has a weight of 1. This means that any extra vertical space is added to the second row.

GridBagLayout is a powerful layout manager. It is worth taking some time to experiment with and explore. Once you understand what the various settings do, you can use **GridBagLayout** to position components with a high degree of precision.

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 the AWT 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.

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

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

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the 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( ) throws HeadlessException
MenuItem(String itemName) throws HeadlessException
MenuItem(String itemName, MenuShortcut keyAccel) throws HeadlessException
```

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

You 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.

You can determine an item's status by calling **isEnabled()**. This method is shown here:

```
boolean isEnabled()
```

isEnabled() returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

You can change the name of a menu item by calling **setLabel()**. You 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.

You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**. It has these constructors:

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

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.

You can obtain the status of a checkable item by calling **getState()**. You 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**. To check an item, pass **true** to **setState()**. To clear an item, pass **false**.

Once you have created a menu item, you 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 you have added all items to a **Menu** object, you 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 generate events only 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. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand()** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **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.

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

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

    MenuFrame(String title) {
        super(title);

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

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

        Menu edit = new Menu("Edit");
        MenuItem item6, item7, item8, item9;
        edit.add(item6 = new MenuItem("Cut"));
        edit.add(item7 = new MenuItem("Copy"));
```

```

edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));

Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);

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

mbar.add(edit);

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

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

public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

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

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

```

```

        g.drawString("Testing is off.", 10, 240);
    }
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;

    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;

    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    // Handle action events.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = ae.getActionCommand();
        if (arg.equals("New..."))
            msg += "New.";
        else if (arg.equals("Open..."))
            msg += "Open.";
        else if (arg.equals("Close"))
            msg += "Close.";
        else if (arg.equals("Quit..."))
            msg += "Quit.";
        else if (arg.equals("Edit"))
            msg += "Edit.";
        else if (arg.equals("Cut"))
            msg += "Cut.";
        else if (arg.equals("Copy"))
            msg += "Copy.";
        else if (arg.equals("Paste"))
            msg += "Paste.";
        else if (arg.equals("First"))
            msg += "First.";
        else if (arg.equals("Second"))
            msg += "Second.";
        else if (arg.equals("Third"))
            msg += "Third.";
        else if (arg.equals("Debug"))
            msg += "Debug.";
        else if (arg.equals("Testing"))
            msg += "Testing.";
    }
}

```



```

        menuFrame.msg = msg;
        menuFrame.repaint();
    }

    // Handle item events.
    public void itemStateChanged(ItemEvent ie) {
        menuFrame.repaint();
    }
}

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

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

        setSize(new Dimension(width, height));

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

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

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

```

Sample output from the **MenuDemo** applet is shown in Figure 26-8.

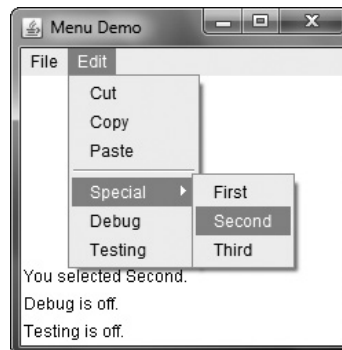


Figure 26-8 Sample output from the **MenuDemo** applet

There is one other menu-related class that you might find interesting: **PopupMenu**. It works just like **Menu**, but produces a menu that can be displayed at a specific location. **PopupMenu** provides a flexible, useful alternative for some types of menuing situations.

Dialog Boxes

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don't have menu bars, but in other respects, they function like frame windows. (You can add controls to them, for example, in the same way that you 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 you cannot access other parts of your program until you have closed the dialog box. When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible. In the AWT, dialog boxes are of type **Dialog**. Two commonly used constructors are shown here:

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

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

Following is a modified version of the preceding menu program that displays a modeless dialog box when the New option is chosen. Notice that when the dialog box is closed, **dispose()** is called. This method is defined by **Window**, and it frees all system resources associated with the dialog box window.

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

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

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

```

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

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

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

    MenuFrame(String title) {
        super(title);

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

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

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

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

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

        mbar.add(edit);
    }
}

```

```

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

        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // register it to receive those events
        addWindowListener(adapter);
    }

    public void paint(Graphics g) {
        g.drawString(msg, 10, 200);

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

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

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;

    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.dispose();
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;

```

```

public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
}

// Handle action events.
public void actionPerformed(ActionEvent ae) {
    String msg = "You selected ";
    String arg = ae.getActionCommand();
    // Activate a dialog box when New is selected.
    if(arg.equals("New...")) {
        msg += "New.";
        SampleDialog d = new
            SampleDialog(menuFrame, "New Dialog Box");
        d.setVisible(true);
    }
    // Try defining other dialog boxes for these options.
    else if(arg.equals("Open..."))
        msg += "Open.";
    else if(arg.equals("Close"))
        msg += "Close.";
    else if(arg.equals("Quit..."))
        msg += "Quit.";
    else if(arg.equals("Edit"))
        msg += "Edit.";
    else if(arg.equals("Cut"))
        msg += "Cut.";
    else if(arg.equals("Copy"))
        msg += "Copy.";
    else if(arg.equals("Paste"))
        msg += "Paste.";
    else if(arg.equals("First"))
        msg += "First.";
    else if(arg.equals("Second"))
        msg += "Second.";
    else if(arg.equals("Third"))
        msg += "Third.";
    else if(arg.equals("Debug"))
        msg += "Debug.";
    else if(arg.equals("Testing"))
        msg += "Testing.";
    menuFrame.msg = msg;
    menuFrame.repaint();
}

public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
}
}

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

```

```

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

    setSize(width, height);

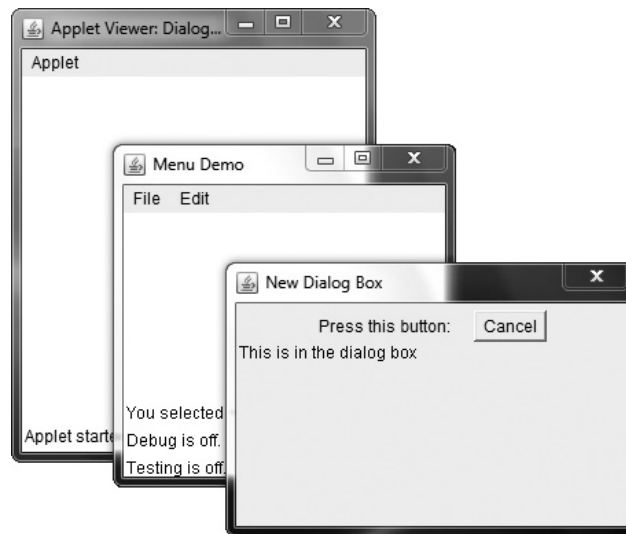
    f.setSize(width, height);
    f.setVisible(true);
}

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

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

```

Here is sample output from the **DialogDemo** applet:



TIP On your own, try defining dialog boxes for the other options presented by the menus.

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. Here are three **FileDialog** constructors:

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

Here, *parent* is the owner of the dialog box. The *boxName* parameter specifies 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. If *how* is omitted, the box is selecting a file for reading.

FileDialog provides methods that allow you 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:

```
/* Demonstrate File Dialog box.

   This is an application, not an applet.
*/
import java.awt.*;
import java.awt.event.*;

// Create a subclass of Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);

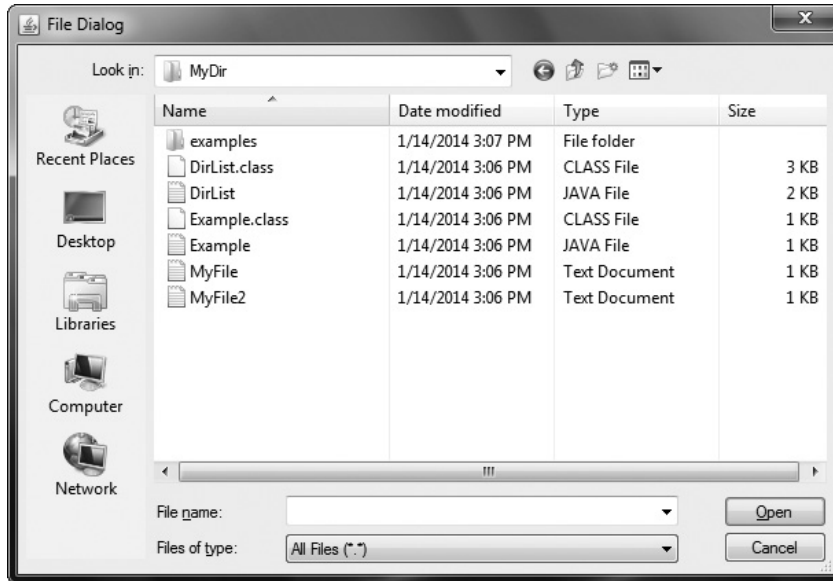
        // remove the window when closed
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }
}

// Demonstrate FileDialog.
class FileDialogDemo {
    public static void main(String args[]) {
        // create a frame that owns the dialog
        Frame f = new SampleFrame("File Dialog Demo");
        f.setVisible(true);
        f.setSize(100, 100);

        FileDialog fd = new FileDialog(f, "File Dialog");

        fd.setVisible(true);
    }
}
```

The output generated by this program is shown here. (The precise configuration of the dialog box may vary.)



One last point: Beginning with JDK 7, you can use **FileDialog** to select a list of files. This functionality is supported by the **setMultipleMode()**, **isMultipleMode()**, and **getFiles()** methods.

A Word About Overriding **paint()**

Before concluding our examination of AWT controls, a short word about overriding **paint()** is in order. Although not relevant to the simple AWT examples shown in this book, when overriding **paint()**, there are times when it is necessary to call the superclass implementation of **paint()**. Therefore, for some programs, you will need to use this **paint()** skeleton:

```
public void paint(Graphics g) {
    // code to repaint this window

    // Call superclass paint()
    super.paint(g);
}
```


In Java, there are two general types of components: heavyweight and lightweight. A heavyweight component has its own native window, which is called its *peer*. A lightweight component is implemented completely in Java code and uses the window provided by an ancestor. The AWT controls described and used in this chapter are all heavyweight. However, if a container holds any lightweight components (that is, has lightweight child components), your override of **paint()** for that container must call **super.paint()**. By calling **super.paint()**, you ensure that any lightweight child components, such as lightweight controls, get properly painted. If you are unsure of a child component's type, you can call **isLightweight()**, defined by **Component**, to find out. It returns **true** if the component is lightweight, and **false** otherwise.