Chapter 13

The AWT

The AWT, the *Abstract Windowing Toolkit*, is the portion of the Java run-time library that is involved with creating, displaying, and facilitating user interaction with window objects. The AWT is an example of a software *framework*. A framework is a way of structuring generic solutions to a common problem, using polymorphism as a means of creating specialized solutions for each new application. Thus, examining the AWT will illustrate how polymorphism is used in a powerful and dynamic fashion in the language Java.

13.1 The AWT Class Hierarchy

From the very first, we have said that class Frame represents the Java notion of an application window, a two dimensional graphical surface that is shown on the display device, and through which the user interacts with a computer program. All our applications have been formed by subclassing from Frame, overriding various methods, such as the paint method for repainting the window. In actuality, much of the behavior provided by class Frame is inherited from parent classes (see Figure 13.1). Examining each of these abstractions in turn will help illustrate the functioning of the Java windowing system, as well as illustrating the power of inheritance as a mechanism for code reuse and sharing.

The class Object is the parent class of all classes in Java. It provides the ability to compare two objects for equality, compute a hash value for an object, and determine the class of an object. Methods defined in class Object include the following:

equals (anObject)	returns true if object is equal to argument
getClass ()	returns the name of the class of an object
hashCode ()	returns a hash value for an object
toString ()	returns a string representation of an object

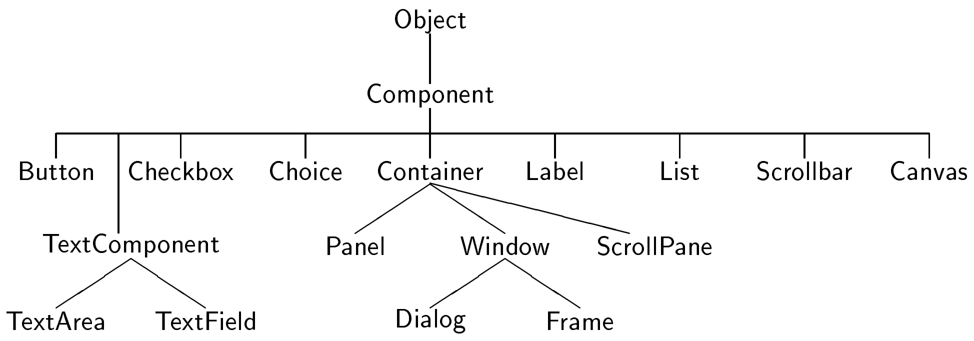


Figure 13.1: The AWT class hierarchy

A Component is something that can be displayed on a two dimensional screen and with which the user can interact. Attributes of a component include a size, a location, foreground and background colors, whether or not it is visible, and a set of listeners for events. Methods defined in class Component include the following:

enable(), disable()	enable/disable a component
setLocation(int,int), getLocation()	set and get component location
setSize(int,int), getSize()	set and get size of component
setVisible(boolean)	show or hide the component
setForeground(Color), getForegound()	set and get foreground colors
setBackground(Color), getBackground()	set and get background colors
setFont(Font), getFont()	set and get font
repaint(Graphics)	schedule component for repainting
paint(Graphics)	repaint component appearance
addMouseListener(MouseListener)	add a mouse listener for component
addKeyListener(KeyListener)	add a keypress listener for component

Besides frames, other types of components include buttons, checkboxes, scroll bars, and text areas.

A Container is a type of component that can nest other components within it. A container is the way that complex graphical interfaces are constructed. A Frame is a type of Container, so it can hold objects such as buttons and scroll bars. When more complicated interfaces are necessary, a Panel (another type of container) can be constructed, which might hold, for example, a collection of buttons. Since this Panel is both a Container and a Component, it can be inserted into the Frame. A container maintains a list of the components it manipulates, as well as a layout manager to determine how the components should be displayed. Methods defined in class Container include the following:

setLayout (LayoutManager)	set layout manager for display
add (Component), remove (Component)	add or remove component from display

A Window is a type of Container. A window is a two-dimensional drawing surface that can be displayed on an output device. A window can be stacked on top of other windows, and moved either to the front or back of the visible windows. Methods defined in class Window include the following:

show()	make the window visible
toFront()	move window to front
toBack()	move window to back

Finally, a Frame is a type of window with a title bar, a menu bar, a border, a cursor, and other properties. Methods defined in class Frame include the following:

setTitle(String), getTitle()	set or get title
setCursor(int)	set cursor
setResizable()	make the window resizable
setMenuBar(MenuBar)	set menu bar for window

If we consider a typical application, such as the CannonWorld application of Chapter 6, we see that it uses methods from a number of different levels of the class hierarchy:

setTitle(String)	inherited from class Frame
setSize(int, int)	inherited from class Component
show()	inherited from class Window
repaint()	inherited from class Component
paint()	inherited from Component, overridden in application class

The power of the AWT, indeed the power of any framework, comes through the use of a polymorphic variable. When the method show in class Window is invoked, it calls the method setVisible in class Component. This method calls repaint, which in turn calls paint. The code for the algorithm used by setVisible and repaint resides in class Component. When it is being executed, the framework "thinks" that it is dealing only with an instance of Component. However, in actuality the method paint that is being executed is the version that has been overridden in the application class. Thus, there are two views of the function being executed, as described in Figure 13.2.

The code in the parent classes (Component, Container, Window and Frame) can all be written without reference to any particular application. Thus, this code can be easily carried from one application to the next. To specialize the design framework to a new application it is only necessary to override the appropriate methods (such as paint or event listeners) to define application specific behavior. Thus, the combination of inheritance, overriding, and polymorphism permits design and software reuse on a grand scale.

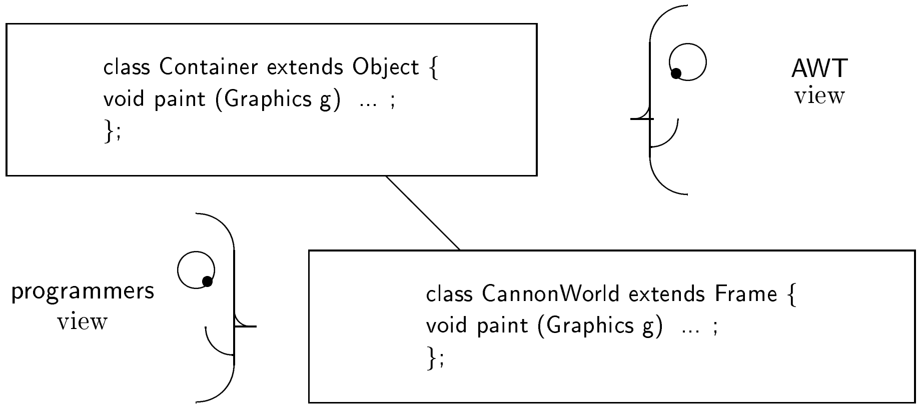


Figure 13.2: Two views of a component

13.2 The Layout Manager

The idea of a layout manager, which is the technique used by the AWT to assign the locations of components within a container, is an excellent illustration of the combination of polymorphic techniques of composition and inheritance. The layout manager is charged with taking the list of components held in a container, and assigning them positions on the surface covered by the container. There are a variety of standard layout managers, each of which will place components in slightly different ways. The programmer developing a graphical user interface creates an instance of a layout manager and hands it to a container. Generally, the task of creation is the only direct interaction the programmer will have with the layout manager, as thereafter all commands will be handled by the container itself.

The connections between the application class, the container, and the layout manager illustrate yet once more the many ways that inheritance, composition, and interfaces can be combined (Figure 13.3). The application class may inherit from the container (as is usually the case when an application is formed using inheritance from class Frame) or it may hold the container as a component. The container itself, however, holds the layout manager as a data field, as part of the internal state of the container. But in actual fact, the variable that holds the layout manager is polymorphic. While the Container thinks that it is maintaining a value of type LayoutManager, in fact it will be holding a value from some other type, such as GridLayout, that is implementing the LayoutManager interface.

There are three different mechanisms at work here; inheritance, composition, and implementation of an interface. Each is serving a slightly different purpose. Inheritance is the *is-a* relation, and links the application class to the parent window class. This allows the code written in the AWT class Window to perform application-specific actions, by invoking

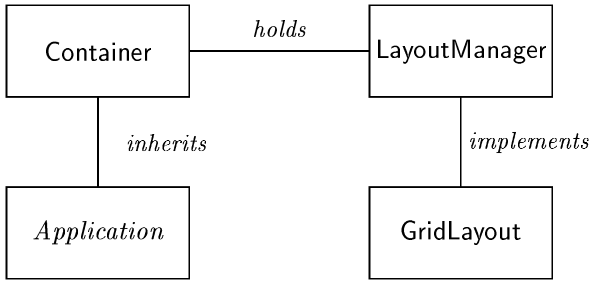


Figure 13.3: Relationships between Layout Manager Components

methods in the application class that override methods in the parent class (paint(), for example). The fact that composition is used to link the container with the layout manager makes the link between these two items very dynamic–the programmer can easily change the type of layout manager being employed by a container. This dynamic behavior is very difficult to achieve using inheritance alone, since the inheritance relationship between a parent and child is established at compile time. Finally, the fact that LayoutManager is simply and interface, and that various different classes of objects implement this interface, means that the programmer is free to develop alternative layout managers using a wide variety of techniques. (This freedom would be much more constrained if, for example, LayoutManager was a class which alternative layout managers needed to extend).

13.2.1 Layout Manager Types

There are five standard types of layout managers. These are BorderLayout, GridLayout, CardLayout, FlowLayout and GridBagLayout. The BorderLayout manager can manage no more than five different components. This is the default layout manager for applications that are constructed by subclassing from Frame. The five locations are shown in Figure 13.4. They correspond to the left and right, top and bottom, and center of the display. Not all five locations need be filled. If a location is not used, the space is allocated to the remaining components.

When a border layout manager is employed the first argument in the add method is used to specify which position a component in filling in a collection:

```
add("North", new Button("quit"));
add("Center", colorField);
```

The next most common type of layout is the GridLayout. The manager for this layout creates rectangular array of components, each occupying the same size portion of the screen.

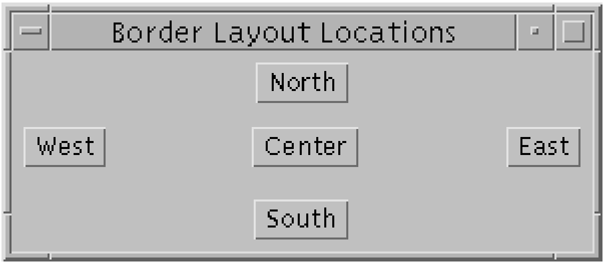


Figure 13.4: Locations Recognized by Border Layout Manager

Using arguments with the constructor, the programmer specifies the number of rows and the number of columns in the grid. Two additional integer arguments can be used to specify a horizontal and vertical space between the components. An example of a panel formatted using a GridLayout is shown in Figure 13.8. The section of code for that application that creates the layout manager is as follows:

```
Panel p = new Panel();
    // make a 4 by 4 grid,
    // with 3 pixels between each element
p.setLayout (new GridLayout(4, 4, 3, 3));

p.add (new ColorButton(Color.black, "black"));
p.add (new ColorButton(Color.blue, "blue"));
```

A FlowLayout manager places components in rows left to right, top to bottom. Unlike the layout created by a GridLayout manager, the components managed by a flow layout manager need not all have the same size. When a component cannot be completely placed on a row without truncation, a new row is created. The flow manager is the default layout manager for the class Panel (as opposed to Frame, where the default manager is a BorderLayout).

A CardLayout manager stacks components vertically. Only one component is visible at any one time. The components managed by a card layout manager can be named (using the string argument to the add method). Subsequently, a named component can be made the visible component. This is one of the few instances where the programmer would have direct interaction with the layout manager.

```
CardLayout lm = new CardLayout();
Panel p = new Panel (lm);
p.add ("One", new Label ("Number One"));
```

```
p.add ("Two", new Label ("Number Two"));
p.add ("Three", new Label ("Number Three"));
...
lm.show ("Two"); // show component "Two"
```

The most general type of layout manager is the GridBagLayout manager. This manager allows the programmer to create a non-uniform grid of squares, and place components in various positions within each square. However, the details of the use of this manager are complex, and will not be described here.

13.3 User Interface Components

The variety of user interface components in the Java AWT library are again a good illustration of the power of polymorphism provided both through inheritance and interfaces. With the exception of menu bars, all the user interface components are subclassed from the parent class Component (Figure 13.1). Containers assume only that the elements they will hold are instances of class Component. In fact, the values they maintain are polymorphic, and represent more specialized values, such as buttons or scroll bars. Thus, the design of the user interface construction system depends upon the mechanisms of inheritance, polymorphism and substitutability.

13.3.1 Labels

The simplest type of user interface component is a Label. A label has only the text it will display. It will display as much of the text as it can in the area it is given.

```
Label lab = new Label("score: 0 to 0");
add ("South", lab); // put label on top of window
```

Unlike other components, a label does not respond to any type of event, such as a mouse click or a key press. However, the text of the label can be changed using the function setText(String), and the current text of a label can be retrieved using getText().

13.3.2 Button

A Button is a labeled component, usually represented by a rounded box, that can respond to user interaction. As we have seen in earlier programs, interaction with a button is achieved by attaching an ActionListener object to the button. The ActionListener object is then notified when the button is pressed.

```
Button b = new Button ("do it!");
```

```
    b.addActionListener (new doIt());
    ..
    private class doIt implements ActionListener {
        public void actionPerformed (ActionEvent e) {
            // what ever do it does
            ...
        }
    }
```

A useful technique is to combine the button object and the button listener in one new class. This new class both subclasses from the original Button class and implements the ActionListener interface. For example, in the case study that is presented in Section 13.5, we create a set of buttons for different colors. Each button holds a color value, and when pressed invokes a method using the color as argument. This class is written as follows:

```
private class ColorButton extends Button implements ActionListener {
    private Color ourColor;

    public ColorButton (Color c, String name) {
        super (name); // create the button
        ourColor = c; // save the color value
        addActionListener (this); // add ourselves as listener
        }

    public void actionPerformed (ActionEvent e) {
            // set color for middle panel
        setFromColor (ourColor);
        }
}
```

Notice how the object registers itself as a listener for button actions. The pseudo-variable this is used when an object needs to denote itself. When pressed, the button will invoke the method actionPerformed, which will then invoke the procedure setFromColor that is found in the surrounding class.

We can even take this technique one step further, and define a generic ButtonAdapter class that is both a button and a listener. The actions of the listener will be encapsulated by an abstract method, which must be implemented by a subclass:

```
abstract class ButtonAdapter extends Button implements ActionListener {
    public ButtonAdapter (String name) {
        super (name);
        addActionListener (this);
```

```
        }

    public void actionPerformed (ActionEvent e) { pressed(); }

    public abstract void pressed ();
}
```

To create a button using this abstraction, the programmer must subclass and override the method pressed. This, however, can be done easily using a class definition expression. The following, for example, creates a button that when pressed will halt the application.

```
    Panel p = new Panel();

    p.add (new ButtonAdapter("Quit"){
        public void pressed () { System.exit(0); }});
```

13.3.3 Canvas

A Canvas is a simple type of component, having only a size and the ability to be a target for drawing operations. Among other uses, the class Canvas is often subclassed to form new types of components. We will illustrate one use of a Canvas when we discuss the class ScrollPane (Section 13.4.1).

13.3.4 Scroll Bars

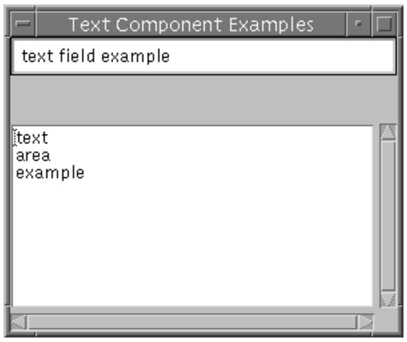
A ScrollBar is a slider, used to specify integer values over a wide range. Scroll bars can be displayed in either a horizontal or a vertical direction. The maximum and minimum values can be specified, as well as the line increment (the amount the scroll bar will move when it is touched in the ends), and the page increment (the amount it will move when it is touched in the background area between the slider and the end). Like a button, interaction is provided for a scroll bar by defining a listener that will be notified when the scroll bar is modified.

The case study at the end of this chapter uses a technique similar to the one described earlier in the section on buttons. Figure 13.8 shows a snapshot of this application, which includes three vertical scroll bars. The class ColorBar represents a scroll bar for maintaining colors. The constructor for the class creates a vertical scroll bar with an initial value of 40 and a range between 0 and 255. The background color for the scroll bar is set using a given argument. Finally, the object itself is made a listener for scroll bar events. When the scroll bar is changed, the method adjustmentValueChanged will be executed. Typically, within this method the current value of the scroll bar would be accessed using getValue(). In this particular application, a bank of three scroll bars will be created, and the value of all three will be recovered in a shared procedure named setFromBar.

```
private class ColorBar extends Scrollbar implements AdjustmentListener {
    public ColorBar (Color c) {
        super (Scrollbar.VERTICAL, 40, 0, 0, 255);
        setBackground (c);
        addAdjustmentListener (this);
    }

    public void adjustmentValueChanged (AdjustmentEvent e) {
        // method setFromBar will get scroll bar
        // value using getValue ();
        setFromBar ();
    }
}
```

13.3.5 Text Components

A text component is used to display editable text. There are two varieties of text components, TextField and TextArea. The first is a fixed size block, while the second uses scroll bars to display a larger block of text, not all of which might be viewable at any one time. The following illustrates these two types of items:



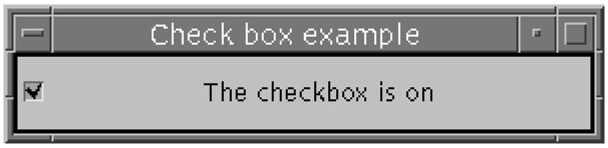
The text in a text component can be set or accessed by the program using the functions setText(String) and getText(). Additional text can be added to the text area using the method append(String). Various other methods can be used to indicate whether or not the text is editable, and to select a subportion of the text. A TextListener can be attached to a text component. The listener must implement the TextListener interface:

```
interface TextListener extends EventListener {
    public void textValueChanged (TextEvent e);
```

```
}
```

13.3.6 Checkbox

A Checkbox is a component that maintains and displays a labeled binary state. The state described by a check box can be either on or off. The current state of the Checkbox can be set or tested by the programmer. A Checkbox is typically used in an application to indicate a binary (on/off, yes/no) choice.



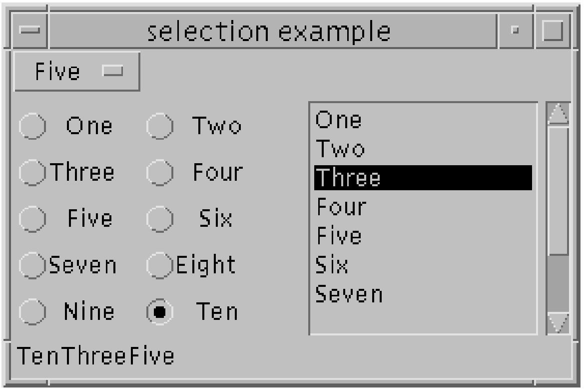
Both the label and the state of the Checkbox can be set by the programmer, using the functions getLabel, setLabel, getState and setState. Changing the state of a check box creates an ItemEvent, that is registered with any ItemListener objects. The following simple application illustrates the use of these methods:

```
class CheckTest extends Frame {
    private Checkbox cb = new Checkbox ("the checkbox is off");

    public static void main (String [ ] args)
        { Frame world = new CheckTest(); world.show(); }

    public CheckTest () {
        setTitle("Check box example"); setSize(300, 70);
        cb.addItemListener (new CheckListener());
        add ("Center", cb);
        }

    private class CheckListener implements ItemListener {

        public void itemStateChanged (ItemEvent e) {
            if (cb.getState())
                cb.setLabel ("The checkbox is on");
            else cb.setLabel ("The checkbox is off");
            }
    }
}
```

13.3.7 Checkbox Groups, Choices and Lists

There are three types of interface components that are typically employed to allow the user to select one item from a large number of possibilities. The first is a group of connected checkboxes, that have the property that only one can be set at any one time. Such a collection is sometimes called a *radio button* group, since their behavior is similar to the way buttons in car radios work. The second form is termed a Choice. A Choice object displays only one selection, but when the user clicks the mouse in the selection area, a pop-up menu appears that allows the choice to be changed to a different selection. A third possibility is termed a List. A List is similar to a Choice, however several possibilities out of the range can be displayed at one time.

The following figure illustrates all three possibilities. The code to produce this example is shown in Figure 13.5.



A Checkbox group should be used when the number of alternatives is small. A choice or a list should be used if the number of alternatives is five or more. A choice takes up less space in the display, but makes it more difficult to view all the alternatives.

To create a Choice or a List object, the programmer specifies each alternative using the method addItem. An ItemListener can be attached to the object. When a selection is made, the listener will be informed using the method itemStateChanged. The text of the selected item can be recovered using the method getSelectedItem.

To structure a group of check boxes as a group, the programmer first creates a CheckboxGroup. This value is then passed as argument to each created check box, along with a third argument that indicates whether or not the check box should be initially active. If more than one button is made active (as here) only the last button will remain active. The current check box can be accessed using the method getSelectedCheckbox.

As a check box group is constructed out of several components, it is almost always laid out on a Panel. The Panel is then placed as a single element in the original layout. This is shown in Figure 13.5. Here a five by two grid is used as layout for the ten check boxes.

```
class ChoiceTest extends Frame {
    public static void main (String [ ] args)
        { Window world = new ChoiceTest(); world.show(); }

    private String [ ] choices = {"One", "Two", "Three", "Four",
        "Five", "Six", "Seven", "Eight", "Nine", "Ten"};
    private Label display = new Label();
    private Choice theChoice = new Choice();
    private List theList = new List();
    private CheckboxGroup theGroup = new CheckboxGroup();
    private ItemListener theListener = new ChoiceListener();

    public ChoiceTest () {
        setTitle ("selection example ");
        setSize (300, 300);
        for (int i = 0; i < 10; i++) {
            theChoice.addItem (choices[i]);
            theList.addItem (choices[i]); }
        theChoice.addItemListener (theListener);
        theList.addItemListener (theListener);
        add ("West", makeCheckBoxes()); add ("North", theChoice);
        add ("East", theList); add ("South", display);
        }

    private class ChoiceListener implements ItemListener {
        public void itemStateChanged (ItemEvent e) {
            display.setText (theGroup.getSelectedCheckboxGroup().getLabel()
                + theList.getSelectedItem() + theChoice.getSelectedItem());
            }
        }

    private Panel makeCheckBoxes() {
        Panel p = new Panel (new GridLayout(5,2));
        for (int i = 0; i < 10; i++) {
            Checkbox cb = new Checkbox(choices[i], theGroup, false);
            cb.addItemListener (theListener); p.add (cb); }
        return p;
        }
}
```

Figure 13.5: Alternative ways to display choices

13.4 Panels

A Panel is a Container that acts like a Component. A panel represents a rectangular region of the display. Each panel holds its own layout manager, which can differ from the layout manager for the application display. Items can be inserted into the panel. The panel, as a single unit, it then inserted into the application display.

The use of a panel is illustrated by the application described in Figure 13.5. Here the method makeCheckBoxes creates a panel to hold the ten check boxes that make up the check box group. This panel is structured, using a GridLayout as a five by two element matrix. This group of ten components can then be treated as a single element, and is placed on the left side of the application layout.

More examples of the use of panels will be provided by the application that will be described in the next section. A snapshot of the window for this application is shown in Figure 13.8. The three scroll bars on the left are placed on a Panel. This panel is laid out using a BorderLayout manager. The procedure to create and return this panel is described as follows:

```
private Panel makeScrollBars () {
    Panel p = new Panel();
    p.setLayout (new BorderLayout());
    p.add("West", redBar);
    p.add("Center", greenBar);
    p.add("East", blueBar);
    return p;
}
```

The panel returned as the result of this method is then placed on the left side of the application window.

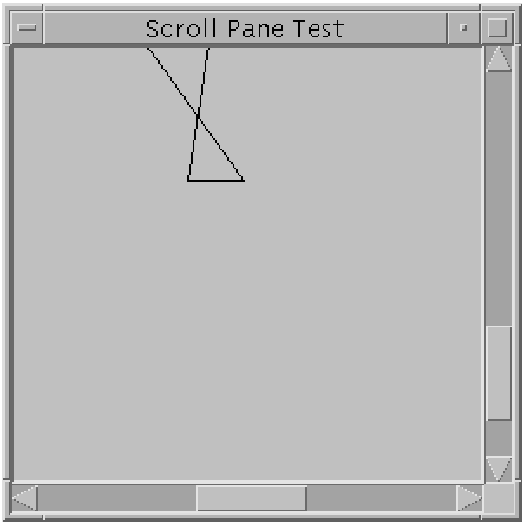
13.4.1 ScrollPane

A ScrollPane is in many ways similar to a Panel. Like a panel, it can hold another component. However, a ScrollPane can only hold one component, and it does not have a layout manager. If the size of the component being held is larger than the size of the ScrollPane itself, scroll bars will be automatically generated to allow the user to move the underlying component.

We illustrate the use of a ScrollPane with a simple test program, shown in Figure 13.6. The application window in this program will be set to 300 by 300 pixels, but a scroll pane is created that holds a canvas that has been sized to 1000 by 1000 pixels. Scroll bars will therefore be added automatically that allow the user to see portions of the underlying canvas. As mouse events are detected by the canvas, points will be added to a Polygon. To paint the application window, the canvas simply draws the polygon values.

```
class BigCanvas extends Frame {

    public static void main ( String [ ] args) {
        Frame world = new BigCanvas();
        world.show();
        }

    private Polygon poly = new Polygon();
    private Canvas cv = new Canvas();

    public BigCanvas () {
        setSize (300, 300);
        setTitle ("Scroll Pane Test");
            // make canvas larger than window
        cv.setSize (1000, 1000);
        cv.addMouseListener (new MouseKeeper());
            // make scroll pane to manage canvas
        ScrollPane sp = new ScrollPane();
        sp.add(cv);
        add("Center", sp);
        }

    public void paint (Graphics g) {
            // redraw canvas
        Graphics gr = cv.getGraphics();
        gr.drawPolygon (poly);
        }

    private class MouseKeeper extends MouseAdapter {
        public void mousePressed (MouseEvent e) {
            poly.addPoint (e.getX(), e.getY());
            repaint();
        }
    }
}
```

Figure 13.6: Test program for Scroll Panes



13.5 Case Study: A Color Display

A simple test program will illustrate how panels and layout managers are used in developing user interfaces. The application will also illustrate the use of scroll bars, and the use of methods provided by the class Color. Finally, we can also use this program to illustrate how nested classes can be employed to combine the actions of creating a new graphical component (such as a button or a slider) and listening for actions relating to the component.

The class ColorTest (Figure 13.7) creates a window for displaying color values. The window, shown in Figure 13.8, is divided into four separate regions. These four regions are managed by the default layout manager for class Frame. This layout manager is a value of type BorderLayout.

At the top (the "north" side) is a text region, a component of type TextField, that describes the current color. To the left (the "west" region) is a trio of sliders that can be used to set the red, green and blue values. To the right (the "east" region) is a four by four bank of 16 buttons. These are constructed on a Panel that is organized by a GridLayout manager. Thirteen of the buttons represent the pre-defined color values. Two more represent the actions of making a color brighter and darker. The final button will halt the application. Finally, in the middle will be a square panel that represents the specified color.

The class ColorTest holds six data fields. The first represents the current color in the middle panel, while the remaining five represent different graphical objects. Three represent the slider, one represents the text field at the top of the page, and one represents the color panel in the middle.

The three sliders make use of the class ColorBar described earlier in Section 13.3.4. The argument used with the constructor for each class is the color to be used in painting the

```
class ColorTest extends Frame {
    static public void main (String [ ] args)
        { Frame window = new ColorTest(); window.show(); }

    private TextField colorDescription = new TextField();
    private Panel colorField = new Panel();
    private Color current = Color.black;
    private Scrollbar redBar = new ColorBar(Color.red);
    private Scrollbar greenBar = new ColorBar(Color.green);
    private Scrollbar blueBar = new ColorBar(Color.blue);

    public ColorTest () {
        setTitle ("color test"); setSize (400, 600);
        add("North", colorDescription);
        add("East", makeColorButtons());
        add("Center", colorField);
        add("West", makeScrollBars());
        setFromColor (current);
        }

    private void setFromColor (Color c) {
        current = c; colorField.setBackground (current);
        redBar.setValue(c.getRed()); greenBar.setValue(c.getGreen());
        blueBar.setValue(c.getBlue());
        colorDescription.setText(c.toString());
        }

    private void setFromBar () {
        int r = redBar.getValue(); int g = greenBar.getValue();
        int b = blueBar.getValue(); setFromColor (new Color(r, g, b));
        }

    private Panel makeColorButtons () { ... }
    private Panel makeScrollBars () { ... }
private class BrightenButton extends Button implements ActionListener ...
private class ColorButton extends Button implements ActionListener ...
private class ColorBar extends Scrollbar implements AdjustmentListener ...
}
```

Figure 13.7: The class ColorTest

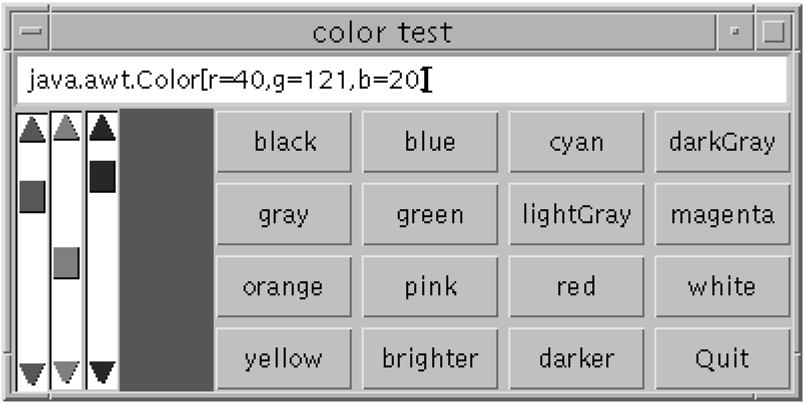


Figure 13.8: Snaphot of ColorTest application

buttons and background for the scroll bar. You will recall that when adjusted, the scroll bar will invoke its listener, which will execute the method adjustmentValueChanged. This method will then execute the procedure setFromBar.

A method makeScrollBars, used to create the panel that holds the three scroll bars, was described earlier in Section 13.4.

The idea of combining inheritance and implementation of an interface is used in creating the buttons that represent the thirteen predefined colors. Each instance of ColorButton, shown earlier in Section 13.3.2, both extends the class Button and implements the ActionListener interface. When the button is pressed, the method setFromColor will be used to set the color of the middle panel using the color stored in the button.

The class BrightenButton is slightly more complex. An index value is stored with the button. This value indicates whether the button represents the "brighten" button or the "darken" button. When pressed, the current color is modified by the appropriate method, and the new value used to set the current color.

```
private class BrightenButton extends Button implements ActionListener {
    private int index;
    public BrightenButton (int i) {
        super ( i == 0 ? "brighter" : "darker");
        index = i;
        addActionListener(this);
        }

    public void actionPerformed (ActionEvent e) {
```

```
        if (index == 0)
            setFromColor (current.brighter());
        else
            setFromColor (current.darker());
        }
}
```

A panel is used to hold the sixteen button values. In this case the layout is described by a 4 by 4 grid pattern. Thirteen represent the predefined buttons. Two represent the brighter and darker buttons. And the final creates a button that when pressed exits the application.

```
private Panel makeColorButtons () {
    Panel p = new Panel();
    p.setLayout (new GridLayout(4,4,3,3));
    p.add (new ColorButton(Color.black, "black"));
    p.add (new ColorButton(Color.blue, "blue"));
    p.add (new ColorButton(Color.cyan, "cyan"));
    p.add (new ColorButton(Color.darkGray, "darkGray"));
    p.add (new ColorButton(Color.gray, "gray"));
    p.add (new ColorButton(Color.green, "green"));
    p.add (new ColorButton(Color.lightGray, "lightGray"));
    p.add (new ColorButton(Color.magenta, "magenta"));
    p.add (new ColorButton(Color.orange, "orange"));
    p.add (new ColorButton(Color.pink, "pink"));
    p.add (new ColorButton(Color.red, "red"));
    p.add (new ColorButton(Color.white, "white"));
    p.add (new ColorButton(Color.yellow, "yellow"));
    p.add (new BrightenButton(0));
    p.add (new BrightenButton(1));
    p.add (new ButtonAdapter("Quit"){
        public void pressed() { System.exit(0); }});
    return p;
}
```

13.6 Dialogs

A Dialog is a special purpose window that is displayed for a short period of time during the course of execution, and thereafter disappears. Dialogs are often used to notify the user of certain events, or to ask simple questions. A dialog must always be attached to an instance of Frame, and disappears automatically when the frame is hidden (such as when the application halts).

Dialog windows can be modal or nonmodal. A modal dialog must be handled, and prevents the user from performing any further action until the dialog is dismissed. A nonmodal dialog, sometimes called a modeless dialog, can be ignored by the user. The processing of actions for a nonmodal dialog is often placed in a separate Thread (See Chapter 20), so that the actions produced by the dialog will not disrupt the continuing processing of the rest of the application. Whether or not a dialog is modal is determined when the dialog is created. The two arguments used in the constructor for the dialog are the application Frame and a boolean value that is true if the dialog is modal.

```
    // create a new nonmodal dialog in current application
Dialog = new Dialog (this, false);
```

Because a Dialog is a type of Window, graphical components can be placed in the dialog area, just as in a Frame or Panel. The default layout manager for a dialog is BorderLayout, the same as with Frame.

The most common functions used with a dialog are not actually defined in the class Dialog, but are inherited from parent classes. These include the following:

setSize(int, int)	set window size
show()	display window
setVisible(false)	remove window from display
setTitle(String), getTitle()	set or get title of window

For modal dialogs, the show method does not return until the dialog is dismissed. Such dialogs must therefore invoke the setVisible(false) method sometime during their processing.

13.6.1 Example Program for Dialogs

An example program will illustrate the creation and manipulation of dialogs. The application shown in Figure 13.10 creates a window with a check box, a button, and a text area. The application window, as well as an example dialog box window, is shown in Figure 13.9. The check box allows the user to specify either a modal or modeless dialog box should be created. The button creates the dialog, while the text area records button presses performed by the dialog.

The procedure makeDialog creates the dialog box. The size of the box is set at 100 by 100 pixels, and four buttons are placed on the box. Three buttons simply type text into the display when pressed, while the last button will hide the dialog. For a modal dialog hiding the dialog is the same as dismissing the dialog box, and returns control to the procedure that created the dialog.

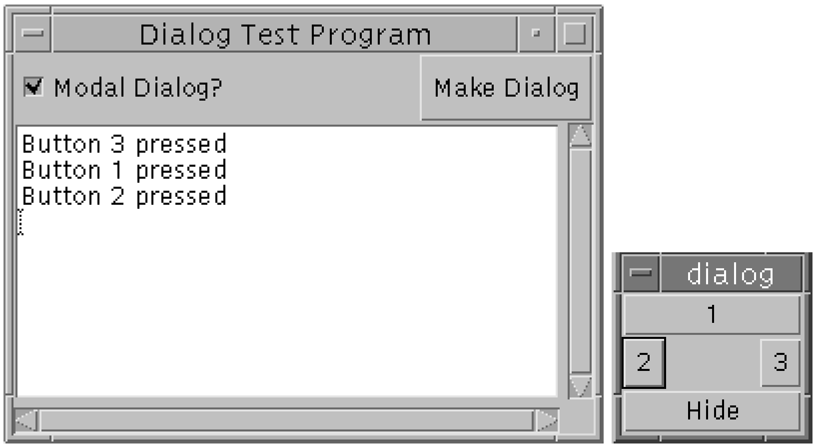


Figure 13.9: Dialog Example Window

13.7 The Menu Bar

Although a menu bar is a graphical component, it is not declared as a subclass of Component. This is because platforms differ in how they handle menu bars, so the implementation must be much more constrained. Both menu bars and menus act much like containers. A menu bar contains a series of menus, and each menu contains a series of menu items.

An instance of *MenuBar* can be attached to a Frame using the method setMenuBar:

```
...
MenuBar bar = new MenuBar();
setMenuBar (bar);
...
```

Individual menus are named, and are placed on the menu bar using the method add:

```
...
Menu helpMenu = new Menu ("Help");
bar.add (helpMenu);
...
```

Menu items are created using the class MenuItem. Each menu item maintains a list of ActionListener objects, the same class used to handle Button events. The listeners will be notified when the menu item is selected.

```
class DialogTest extends Frame {
    static public void main (String [ ] args)
        { Frame world = new DialogTest(); world.show(); }

    private TextArea display = new TextArea();
    private Checkbox cb = new Checkbox("Modal Dialog?");

    public DialogTest () {
        setTitle ("Dialog Test Program");
        setSize (300, 220);

        add ("West", cb);
        add ("East", new Makebutton());
        add ("South", display);
        }

    private class Makebutton extends ButtonAdapter {
        public Makebutton () { super ("Make Dialog"); }
        public void pressed () { makeDialog (cb.getState()); }
        }

    private void makeDialog (boolean modalFlag) {
        final Dialog dlg = new Dialog (this, modalFlag);
        dlg.setSize (100, 100);
        dlg.add ("North", new CountButton(1));
        dlg.add ("West", new CountButton(2));
        dlg.add ("East", new CountButton(3));
        dlg.add ("South", new ButtonAdapter("Hide") {
            public void pressed () { dlg.setVisible(false); }});
        dlg.show();
        }

    private class CountButton extends ButtonAdapter {
        public CountButton (int val) { super ("" + val); }
        public void pressed () {
            display.append("Button " + getLabel() + " pressed\n");}
        }
}
```

Figure 13.10: Example Program for Creating Dialogs

```
    ...
    MenuItem quitItem = new MenuItem ("Quit");
    quitItem.addActionListener (new QuitListener());
    helpMenu.add (quitItem);
    ...
```

There are a number of techniques that can be used to create special-purpose menus, such as tear-off menus, cascading menus, and so on. However, these will not be described here.

13.7.1 A Quit Menu Facility

On many platforms it is sometimes difficult to stop a running Java application. For this reason, it is useful to define a general purpose *Quit* menu bar facility. The class QuitItem (Figure 13.11) creates a listener that will halt the running application when the associated menu item is selected. By overloading the constructor, we make it trivial to add this functionality to any application.

The constructor for QuitItem can be given a MenuItem as argument. In this case it merely attaches itself as a listener to the menu item. Alternatively, it can be given a Menu, in which case it creates a menu item labeled "Quit". Or it can be given a MenuBar, in which case it creates a new menu labeled "Quit" that contains only the quit menu item. Finally, the constructor can be given an application as argument, in which case it creates a new menu bar containing only the one menu which contains only the single quit item. Using the application constructor, a quit menu selection can be added to an application by placing only a single line in the constructor for the application:

```
class ColorTest extends Frame {
    ...
    public ColorTest () {
        ...
            // add quit menu item to application
        new QuitItem (this);
        ...
        }
    }
```

Chapter Summary

The Abstract Windowing Toolkit, or AWT, is the portion of the Java library used for the creation of graphical user interfaces. The design of the AWT is an excellent illustration of

```
class QuitItem implements ActionListener {

    public QuitItem (Frame application) {
        MenuBar mbar = new MenuBar();
        application.setMenuBar (mbar);
        Menu menu = new Menu("Quit");
        mbar.add (menu);
        MenuItem mitem = new MenuItem("Quit");
        mitem.addActionListener (this);
        menu.add (mitem);
        }

    public QuitItem (MenuBar mbar) {
        Menu menu = new Menu("Quit");
        mbar.add (menu);
        MenuItem mitem = new MenuItem("Quit");
        mitem.addActionListener (this);
        menu.add (mitem);
        }

    public QuitItem (Menu menu) {
        MenuItem mitem = new MenuItem("Quit");
        mitem.addActionListener (this);
        menu.add (mitem);
        }

    public QuitItem (MenuItem mitem)
        { mitem.addActionListener (this); }

    public void actionPerformed (ActionEvent e)
         { System.exit(0); }
}
```

Figure 13.11: A General Purpose Quite Item Class

the power of object-oriented techniques. In this chapter we have described the varous AWT components, and the way in which they are used to created user interfaces.

Study Questions

1. What do the letters *AWT* stand for?
2. What are the parent classes of class Frame?
3. In what AWT class is the method setBackground defined?
4. How is a container different from other types of components?
5. Explain why in a framework there are two views of an overridden method, such as paint.
6. What is the task performed by the layout manager?
7. Explain how the three mechanisms of inheritance, composition, and implementation of an interface are all involved in the task of attaching a layout manager to a container.
8. What are the five different layout manager types? Which mangers use the one argument add method, and which use the method in which the first argument is a String value and the second a component?
9. What is the difference between a TextArea and a TextField?
10. What are the three different types of components that allow the user to select one item out of many possibilities?
11. What is a Panel?
12. What are the thirteen predefined values provided by class Color?
13. What do the three numerical values that define a color represent?
14. In what ways is a MenuBar similar to a Component? In what ways is it different?

Exercises

1. Add a menu bar to the Solitare program described in Chapter 9. Then, add two menu items, one to quit the application, and one to reset the application for a new game.
2. Using a text box and a grid of buttons, create a simple calculator application. Buttons correspond to digits and the four arithmetic functions $+$, $-$, $*$ and $/$, as well as the equals sign.