

## CHAPTER

# 32

## Exploring Swing

The previous chapter described several of the core concepts relating to Swing and showed the general form of both a Swing application and a Swing applet. This chapter continues the discussion of Swing by presenting an overview of several Swing components, such as buttons, check boxes, trees, and tables. The Swing components provide rich functionality and allow a high level of customization. Because of space limitations, it is not possible to describe all of their features and attributes. Rather, the purpose of this overview is to give you a feel for the capabilities of the Swing component set.

The Swing component classes described in this chapter are shown here:

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	JTextField	JToggleButton	JTree

These components are all lightweight, which means that they are all derived from **JComponent**.

Also discussed is the **ButtonGroup** class, which encapsulates a mutually exclusive set of Swing buttons, and **ImageIcon**, which encapsulates a graphics image. Both are defined by Swing and packaged in **javax.swing**.

One other point: The Swing components are demonstrated in applets because the code for an applet is more compact than it is for a desktop application. However, the same techniques apply to both applets and applications.

### JLabel and ImageIcon

**JLabel** is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at **JLabel** a bit more closely. **JLabel** can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors. Here are three of them:

```
JLabel(Icon icon)  
JLabel(String str)  
JLabel(String str, Icon icon, int align)
```

Here, *str* and *icon* are the text and icon used for the label. The *align* argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: **LEFT**, **RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

Notice that icons are specified by objects of type **Icon**, which is an interface defined by Swing. The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the **ImageIcon** constructor used by the example in this section:

```
ImageIcon(String filename)
```

It obtains the image in the file named *filename*.

The icon and text associated with the label can be obtained by the following methods:

```
Icon getIcon()
String getText()
```

The icon and text associated with a label can be set by these methods:

```
void setIcon(Icon icon)
void setText(String str)
```

Here, *icon* and *str* are the icon and text, respectively. Therefore, using **setText()** it is possible to change the text inside a label during program execution.

The following applet illustrates how to create and display a label containing both an icon and a string. It begins by creating an **ImageIcon** object for the file **hourglass.png**, which depicts an hourglass. This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
// Demonstrate JLabel and ImageIcon.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JLabelDemo" width=250 height=200>
  </applet>
*/

public class JLabelDemo extends JApplet {

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        }
    }
};
```

```

    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}

private void makeGUI() {

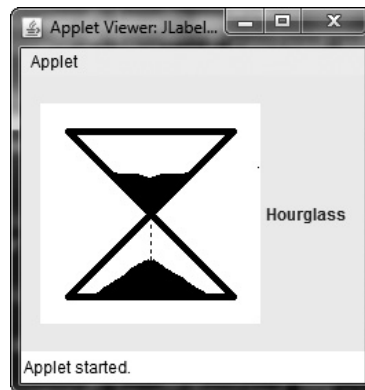
    // Create an icon.
    ImageIcon ii = new ImageIcon("hourglass.png");

    // Create a label.
    JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);

    // Add the label to the content pane.
    add(jl);
}
}

```

Output from the label example is shown here:



## JTextField

**JTextField** is the simplest Swing text component. It is also probably its most widely used text component. **JTextField** allows you to edit one line of text. It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components. **JTextField** uses the **Document** interface for its model. Three of **JTextField**'s constructors are shown here:

```

JTextField(int cols)
JTextField(String str, int cols)
JTextField(String str)

```

Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.

**JTextField** generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses **ENTER**. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in **javax.swing.event**.) Other events are

also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call `getText()`.

The following example illustrates `JTextField`. It creates a `JTextField` and adds it to the content pane. When the user presses `ENTER`, an action event is generated. This is handled by displaying the text in the status window.

```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
   <applet code="JTextFieldDemo" width=300 height=50>
   </applet>
*/

public class JTextFieldDemo extends JApplet {
    JTextField jtf;

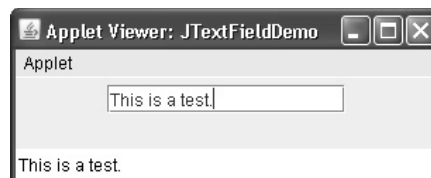
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {

        // Change to flow layout.
        setLayout(new FlowLayout());

        // Add text field to content pane.
        jtf = new JTextField(15);
        add(jtf);
        jtf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                // Show text when user presses ENTER.
                showStatus(jtf.getText());
            }
        });
    }
}
```

Output from the text field example is shown here:



## The Swing Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

**AbstractButton** contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

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

Here, *str* is the text to be associated with the button.

The model used by all buttons is defined by the **ButtonModel** interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

### JButton

The **JButton** class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. **JButton** allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Here, *str* and *icon* are the string and icon used for the button.

When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed( )** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling **setActionCommand( )** on the button. You can obtain the action command by calling **getActionCommand( )** on the event object. It is declared like this:

```
String getActionCommand( )
```

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button

displays an icon that represents a timepiece. When a button is pressed, the name of that timepiece is displayed in the label.

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
   <applet code="JButtonDemo" width=250 height=750>
   </applet>
*/

public class JButtonDemo extends JApplet
implements ActionListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeLaterAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {

        // Change to flow layout.
        setLayout(new FlowLayout());

        // Add buttons to content pane.
        ImageIcon hourglass = new ImageIcon("hourglass.png");
        JButton jb = new JButton(hourglass);
        jb.setActionCommand("Hourglass");
        jb.addActionListener(this);
        add(jb);

        ImageIcon analog = new ImageIcon("analog.png");
        jb = new JButton(analog);
        jb.setActionCommand("Analog Clock");
        jb.addActionListener(this);
        add(jb);

        ImageIcon digital = new ImageIcon("digital.png");
        jb = new JButton(digital);
        jb.setActionCommand("Digital Clock");
        jb.addActionListener(this);
        add(jb);
    }
}
```

```

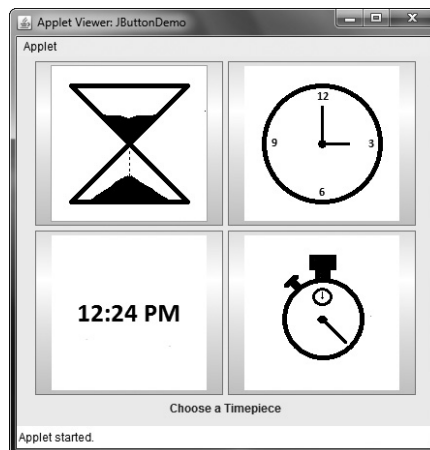
ImageIcon stopwatch = new ImageIcon("stopwatch.png");
jbutton = new JButton(stopwatch);
jbutton.setActionCommand("Stopwatch");
jbutton.addActionListener(this);
add(jbutton);

// Create and add the label to content pane.
jlabel = new JLabel("Choose a Timepiece");
add(jlabel);
}

// Handle button events.
public void actionPerformed(ActionEvent ae) {
    jlabel.setText("You selected " + ae.getActionCommand());
}
}

```

Output from the button example is shown here:



## JToggleButton

A useful variation on the push button is called a *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**, which are described later in this chapter. Thus, **JToggleButton** defines the basic functionality of all two-state components.

**JToggleButton** defines several constructors. The one used by the example in this section is shown here:

```
JToggleButton(String str)
```

This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

**JToggleButton** uses a model defined by a nested class called **JToggleButton.ToggleButtonModel**. Normally, you won't need to interact directly with the model to use a standard toggle button.

Like **JButton**, **JToggleButton** generates an action event each time it is pressed. Unlike **JButton**, however, **JToggleButton** also generates an item event. This event is used by those components that support the concept of selection. When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the **ItemListener** interface. Recall from Chapter 24, that each time an item event is generated, it is passed to the **itemStateChanged()** method defined by **ItemListener**. Inside **itemStateChanged()**, the **getItem()** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event. It is shown here:

```
Object getItem()
```

A reference to the button is returned. You will need to cast this reference to **JToggleButton**.

The easiest way to determine a toggle button's state is by calling the **isSelected()** method (inherited from **AbstractButton**) on the button that generated the event. It is shown here:

```
boolean isSelected()
```

It returns **true** if the button is selected and **false** otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls **isSelected()** to determine the button's state.

```
// Demonstrate JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JToggleButtonDemo" width=200 height=80>
  </applet>
*/

public class JToggleButtonDemo extends JApplet {

    JLabel jlab;
    JToggleButton jtbn;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        }
    }
};
```



```

    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}

private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Create a label.
    jlab = new JLabel("Button is off.");

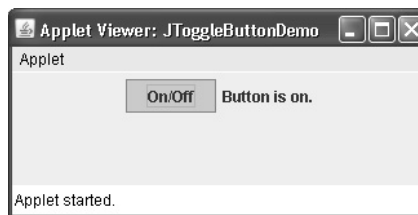
    // Make a toggle button.
    jtbn = new JToggleButton("On/Off");

    // Add an item listener for the toggle button.
    jtbn.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent ie) {
            if(jtbn.isSelected())
                jlab.setText("Button is on.");
            else
                jlab.setText("Button is off.");
        }
    });

    // Add the toggle button and label to the content pane.
    add(jtbn);
    add(jlab);
}
}

```

The output from the toggle button example is shown here:



## Check Boxes

The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons, as just described. **JCheckBox** defines several constructors. The one used here is

```
JCheckBox(String str)
```

It creates a check box that has the text specified by *str* as a label. Other constructors let you specify the initial selection state of the button and specify an icon.