Chapter 1: Fun with Fonts and Colors

In This Chapter

- **Setting the font of a text control**
- **Getting a list of available fonts**
- **Playing with colors**
- **Working with system colors**
- **Setting foreground and background colors**

In this chapter, I look at ways of dressing up the text that appears in Swing controls. In particular, I show you how to change the font that text is displayed in — including bold, italic, and size — as well as how to change the color of your text.

Most of the examples work with labels, but the methods you call to set the font and color are available to all Swing components because they're defined by the `Component` class, which all Swing components inherit.

Also, the information about fonts and colors that I present in this chapter applies to graphics created with the methods of the `Graphics2D` class in Book IX, Chapter 3.

Working with Fonts

In Java, a font is represented by the `Font` class. Each `Font` object has three basic characteristics: the font name, a style identifier (plain, bold, italic, or bold and italic), and a point size.

Although the `Font` class has a ton of methods, you probably won't use them unless you're writing a desktop publishing program in Java. Instead, you can get by with the basic constructor, which has this form:

```
Font(String name, int style, int size)
```

For example, this statement creates a `Font` object for a font named Papyrus:

```
Font("Papyrus", Font.PLAIN, 14)
```

Here, the font style is plain, and the size is 14-point.

Realizing that the `Font` class constructor doesn't really create a font is important. Instead, this constructor creates a `Font` object that represents a font installed already on your computer. Creating a `Font` object with the name Comic Strip doesn't actually create a font named `Comic Strip` unless that font is installed already on the computer.

Using font names

The `name` parameter specifies the name of the installed font you want to use. For example, if you specify `Times New Roman`, the Times New Roman font is used.

Coding string literals for specific fonts is usually not a good idea because you have no way to guarantee that every computer has the exact font you specify. You can get around that problem in two ways:

- **Let the user configure the fonts by picking from a list of available fonts.** Check out the later section, "Getting a list of all available fonts."
- **Use one of several *logical font names* that Java provides in an attempt to let you specify fonts generically.** Table 1-1 lists the logical font names. You don't get much choice when you use logical font names, but at least you can choose between a basic serif font, sans-serif font, and monospaced font. And you can use the `Dialog` and `Dialog Input` fonts to set the font used in dialog boxes.

Table 1-1 Logical Font Names

Logical Font	*Description*
`Serif`	A basic serif font. Times New Roman on Windows, usually Times Roman on other systems.
`SansSerif`	A sans-serif font. Arial on Windows, usually Helvetica on non-Windows systems.
`Monospaced`	A monospaced font. Courier New on Windows, usually Courier on non-Windows systems.
`Dialog`	The font used to display text in system dialog boxes.
`DialogInput`	The font used for text input in system dialog boxes.

Using font styles

Fonts can have one of four styles: plain, bold, italic, and bold-italic. To set the font style, use the following three constants as the second parameter to the `Font` class constructor:

```
Font.BOLD
Font.ITALIC
Font.PLAIN
```

For example, here's how you create a `Font` object for 24-point JSL Ancient Bold:

```
Font("JSL Ancient", Font.BOLD, 24)
```

You may have noticed that bold-italic has no constant. To create a bold-italic font, you combine the `Font.BOLD` and `Font.ITALIC` constants with a `|` operator, like this:

```
Font("Garamond", Font.BOLD | Font.ITALIC, 12)
```

Here, the `Font` object is Garamond, bold and italic, and 12-point.

Setting a component's font

To set the font used to display a component, just call the component's `setFont` method and pass it a `Font` object. For example:

```
JLabel textLabel = new JLabel("Arghh, Matey");
Font f = new Font("JSL Ancient", Font.PLAIN, 16);
textLabel.setFont(f);
```

Here, the font of the label named `textLabel` is set to 16-point JSL Ancient. (JSL Ancient is one of my personal favorites; it's used in the Pirates of the Caribbean ride at Disneyland.)

If the font is used for only one component, you can just create the component right in the `setFont` method call:

```
textLabel.setFont(new Font("JSL Ancient", Font.PLAIN, 16));
```

If you want a component to inherit the font used by the container that holds it (such as a panel), call the component's `setFont` method with the parameter set to `null`. For example, here's code that sets the font for a pair of buttons in a panel named `panel1` to JSL Ancient:

```
JPanel panel1 = new JPanel();
panel1.setFont(new Font("JSL Ancient", Font.PLAIN, 16));

JButton b1 = new JButton("Jolly");
b1.setFont(null);
panel1.add(b1);

JButton b2 = new JButton("Roger");
b2.setFont(null);
panel1.add(b2);
```

In this example, both buttons have their fonts set to `null`, so they both pick up the font of their parent `panel1`.

Getting a list of all available fonts

If you want to let the user pick a font, get a list of all the available fonts on the system so you can put the font names in a combo box. To do that, you first have to get a `GraphicsEnvironment` object that represents the graphics environment the program is running in. The `GraphicsEnvironment` class has a static method — `getLocalGraphicsEnvironment` — that does this for you:

```
GraphicsEnvironment g;
g = GraphicsEnvironment.getLocalGraphicsEnvironment();
```

Note that the `GraphicsEnvironment` class is in the `java.awt` package, so you need to provide an `import` statement to import that package.

After you have a `GraphicsEnvironment` object, you can call its `getAvailableFontFamilyNames` method, which returns an array of strings containing all the font names that are available on the system. For example:

```
String[] fonts;
fonts = g.getAvailableFontFamilyNames();
```

You can then use this array in the constructor of a combo box, like this:

```
JComboBox fontComboBox = new JComboBox(fonts);
```

Then, you can create a font from the name selected by the user with code similar to this:

```
String name = (String) fontComboBox.getSelectedItem();
Font f = new Font(name, Font.PLAIN, 12);
```

A program that plays with fonts

So that you can see how these elements work together, Listing 1-1 presents a simple program that lets the user choose a font, style, and size for the sample text that's displayed. Figure 1-1 shows this program in action. Whenever the user chooses a font or size from one of the combo boxes or selects or deselects one of the check boxes, the font used to display the text at the top of the form is changed accordingly.

Figure 1-1: The Fonts program in action.

Listing 1-1: A Program That Plays with Fonts

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Fonts extends JFrame
{
    public static void main(String [] args)
    {
        new Fonts();
    }

    private JLabel sampleText;                                          →12

    private JComboBox fontComboBox;                                     →14
    private JComboBox sizeComboBox;
    private JCheckBox boldCheck, italCheck;

    private String[] fonts;                                             →18

    public Fonts()
    {
        this.setSize(500,150);
        this.setTitle("Fun with Fonts ");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        FontListener fl = new FontListener();                           →26

        sampleText = new JLabel(
            "All work and no play makes Jack a dull boy");
        this.add(sampleText, BorderLayout.NORTH);                       →30

        GraphicsEnvironment g;                                          →32
        g = GraphicsEnvironment
                .getLocalGraphicsEnvironment();
        fonts = g.getAvailableFontFamilyNames();

        JPanel controlPanel = new JPanel();                             →37

        fontComboBox = new JComboBox(fonts);                            →39
        fontComboBox.addActionListener(fl);
        controlPanel.add(new JLabel("Family: "));
        controlPanel.add(fontComboBox);

        Integer[] sizes = {7, 8, 9, 10, 11, 12,                         →44
                           14, 18, 20, 22, 24, 36};
```

(continued)

Listing 1-1 *(continued)*

```
        sizeComboBox = new JComboBox(sizes);
        sizeComboBox.setSelectedIndex(5);
        sizeComboBox.addActionListener(fl);
        controlPanel.add(new JLabel("Size: "));
        controlPanel.add(sizeComboBox);

        boldCheck = new JCheckBox("Bold");                          →52
        boldCheck.addActionListener(fl);
        controlPanel.add(boldCheck);

        italCheck = new JCheckBox("Ital");                          →56
        italCheck.addActionListener(fl);
        controlPanel.add(italCheck);

        this.add(controlPanel, BorderLayout.SOUTH);                 →60
        fl.updateText();

        this.setVisible(true);
    }

    private class FontListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            updateText();                                           →70
        }

        public void updateText()                                    →73
        {
            String name
                = (String) fontComboBox.getSelectedItem();

            Integer size
                = (Integer)sizeComboBox.getSelectedItem();

            int style;
            if (   boldCheck.isSelected()
                && italCheck.isSelected())
                style = Font.BOLD | Font.ITALIC;
            else if (boldCheck.isSelected())
                style = Font.BOLD;
            else if (italCheck.isSelected())
                style = Font.ITALIC;
            else
                style = Font.PLAIN;

            Font f = new Font(name, style,
                size.intValue());
            sampleText.setFont(f);
        }
    }
}
```

The following paragraphs hit the high points of this program:

→ 12 The label whose font is changed when the user makes a selection.

→ 14 The controls that the user works with to pick the font, size, and style.

→ 18 The `fonts` variable is an array of strings that's used to hold the name of each font available on the system.

→ 26 The `fl` variable holds a reference to the action listener object that handles action events for both combo boxes and both check boxes.

→ 30 The label that contains the sample text is added to the North region of the frame.

→ 32 These lines get the `GraphicsEnvironment` object and then use it to populate the `fonts` array with the font names available on the system.

→ 37 A panel is used to hold the two combo box and two check box controls.

→ 39 These lines create the font combo box and add it to the panel.

→ 44 These lines create the size combo box and add it to the panel. The combo box is filled from an array of integers that lists commonly used point sizes. If you want, you could call `setEditable(true)` to make this combo box editable. Then, the user could type any desired font size into the combo box. To keep the application simple, I did not make the combo box editable.

→ 52 These lines create the bold check box and add it to the panel.

→ 56 These lines create the italic check box and add it to the panel.

→ 60 The panel is added to the South region of the frame. Then, the next line calls the action listener's `updateText` method, which applies the currently selected font, style, and size to the label. (If you don't call this method here, the label is displayed initially with the default font, not with the font indicated by the initial value of the font combo box.)

→ 70 The `actionPerformed` method of the action listener class simply calls `updateText`.

→ 73 The `updateText` method changes the font of the `sampleText` label to the font selected by the user. First, it gets the name selected in the font combo box. Next, it gets the size selected by the user in the size combo box. Because combo boxes return objects, the selected item is cast to an `Integer`. Next, the settings of the two check boxes are evaluated to determine how to set the `style` variable. Finally, a new `Font` object is created and assigned to the `sampleText` label.

Working with Color

In Java, a particular color is represented by an instance of the `Color` class. Every color is a unique combination of three different constituent colors: red, green, and blue. Each constituent is represented by an integer that ranges from 0 to 255, with *zero* meaning the constituent color is completely absent, and *255* meaning the color is completely saturated with the constituent color.

In the following sections, you discover how to use the `Color` class to create color objects. Then, you apply colors to Swing components. And finally, you use a handy Swing dialog box — the Color Chooser.

Creating colors

One way to create a `Color` object is to call the `Color` constructor, passing it the red, green, and blue values you want to use. For example:

```
Color c = new Color(255, 255, 0);
```

Here, a color with full red, full green, and no blue is created. This results in bright yellow.

If all three constituent colors are zero, the resulting color is black. If all three are 255, the result is white. And if all three values are the same, somewhere between 0 and 255, the result is a shade of gray.

Because color numbers can be confusing to work with and hard to remember, the `Color` class provides several static constants that give you pre-defined colors. Table 1-2 lists these constants. For example, here's a statement that creates a `Color` object that represents the color red:

```
Color c = Color.RED;
```

Table 1-2 Constants Defined by the Color Class

BLACK	GRAY	MAGENTA	RED
BLUE	GREEN	ORANGE	WHITE
CYAN	LIGHT_GRAY	PINK	YELLOW
DARK_GRAY			

Colors also have a characteristic called *alpha,* which indicates the color transparency. By default, alpha is set to 255; therefore, the color isn't transparent. If you want to set a different alpha value, you can call a second

`Color` constructor that accepts the alpha value as a fourth parameter. For example:

```
Color c = new Color(255, 0, 0, 128);
```

Here, the color is semitransparent.

The following paragraphs describe a few additional details worth knowing about the `Color` class:

- `Color` objects are immutable; no set methods let you change a color.
- You can get the red, green, blue, and alpha values by using the `getRed`, `getGreen`, `getBlue`, and `getAlpha` methods.
- You can use the `brighter` method to create a color that's brighter than the current color. Likewise, the `darker` method returns a `Color` object that's a little darker than the current color. These methods work by increasing the red, green, and blue values but keeping them in proportion to one another.
- If you call the `Color` constructor with a parameter that's less than zero or greater than 255, `IllegalArgumentException` is thrown. As a result, check the `parameter` values before calling the constructor.
- The `Color` class provides an alternative constructor that lets you set the constituent colors by using `float` values between 0.0 and 1.0.
- The `Color` class is in the `java.awt` package, so you need an `import` statement that specifies either `java.awt.Color` or `java.awt.*`.

Using system colors

You can use the `SystemColor` class to get colors that correspond to the colors configured by the underlying operating system for various GUI elements, such as menu text or the desktop background. Note that the `SystemColor` class extends the `Color` class, so you can use `SystemColor` objects with the `setForeground` and `setBackground` methods or with any other methods that call for `Color` objects.

The `SystemColor` class has a bevy of static methods that return `SystemColor` objects for the colors used by different parts of the system's GUI interface, as listed in Table 1-3. For example, here's a statement that sets the background color of a button to the color used as the background for tooltips:

```
button1.setBackground(SystemColor.info);
```

Table 1-3	SystemColor Methods
Field	*Description*
`static SystemColor activeCaption`	Background color of the active window's title bar.
`static SystemColor activeCaptionBorder`	Border color of the active window's title bar.
`static SystemColor activeCaptionText`	Text color of the active window's title bar.
`static SystemColor control`	Background color used for controls.
`static SystemColor controlText`	Text color used for controls.
`static SystemColor desktop`	Background color used for the desktop.
`static SystemColor inactiveCaption`	Background color used for the title bar of inactive windows.
`static SystemColor inactiveCaptionBorder`	Border color used for the title bar of inactive windows.
`static SystemColor inactiveCaptionText`	Text color used for the title bar of inactive windows.
`static SystemColor info`	Background color used for tooltips.
`static SystemColor infoText`	Text color used for tooltips.
`static SystemColor menu`	Background color used for menus.
`static SystemColor menuText`	Text color used for menus.
`static SystemColor textHighlight`	Background color used for highlighted text.
`static SystemColor textHighlightText`	Text color used for highlighted text.
`static SystemColor textInactiveText`	Text color used for inactive text.
`static SystemColor textText`	Text color used for text boxes and other text controls.
`static SystemColor textHighlight`	Background color used for highlighted text.
`static SystemColor window`	Background color used for windows.
`static SystemColor windowBorder`	Border color used for windows.
`static SystemColor windowText`	Text color used for windows.

Setting the color of Swing components

Every Swing component has two methods that let you set the colors used to draw the component: `setForeground` and `setBackground`. The `setForeground` method sets the color used to draw the component's text, and the `setBackground` method sets the color that fills in behind the text.

For example, here's code that sets the foreground color of a label to red:

```
JLabel errorMessage = new JLabel("Oops!");
errorMessage.setForeground(Color.RED);
```

As with fonts, you can force a component to use the color of its container by setting the color to `null`, like this:

```
textLabel.setForeground(null);
```

Then, if you add `textLabel` to a panel, the label uses the panel's foreground color.

Using a color chooser

The `JColorChooser` class creates a standardized dialog box that lets the user pick a color. This dialog box includes three tabs that let the user choose one of three methods to pick a color:

- The Swatches tab, as shown in Figure 1-2, provides 279 different predefined color choices.
- The HSB tab lets the user select the color by specifying the *hue* (that is, the base color), *saturation* (the amount of the color), and *brightness*.
- The RGB tab lets the user specify the red, green, and blue values for the color.

All you need is one line of code to display a Color Chooser dialog box. Just call the static `showDialog` method, which takes three parameters:

- The parent component to use for the dialog box (`null` to center the dialog box on-screen)
- The text to display in the title bar
- The initial color

The `showDialog` method returns the color selected by the user, or `null` if the user cancels without selecting a color.

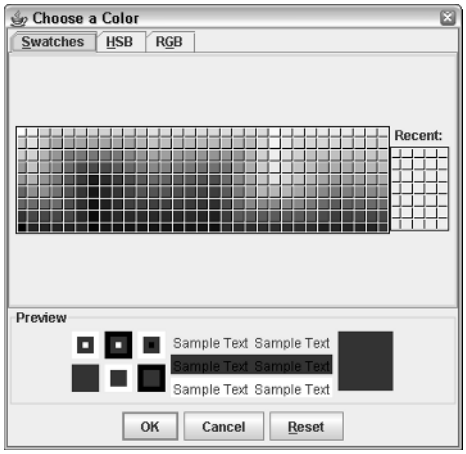

Figure 1-2: A Color Chooser dialog box.

Here's an example:

```
Color c = JColorChooser.showDialog(null, "Choose a Color",
    sampleText.getForeground());
```

Just to prove how easy it is to use a color chooser, Listing 1-2 shows the complete code for a program that uses a color chooser. This program displays the frame shown in Figure 1-3; when the user clicks the Choose Color button, a color chooser just like the one in Figure 1-2 appears. Then, when the user selects a color and clicks OK, the color selected by the user is applied to the label.

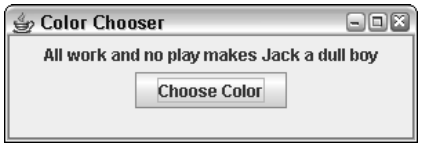

Figure 1-3: The Color Chooser program.

Listing 1-2: A Program That Uses a Color Chooser

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class ColorChooser extends JFrame
{
    public static void main(String [] args)
    {
        new ColorChooser();
```

```
    }

    private JLabel sampleText;                                    →12
    private JButton chooseButton;

    public ColorChooser()
    {
        this.setSize(300,100);
        this.setTitle("Color Chooser");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel1 = new JPanel();

        sampleText = new JLabel(
            "All work and no play makes Jack a dull boy");
        sampleText.setBackground(null);
        panel1.add(sampleText);

        chooseButton = new JButton("Choose Color");
        chooseButton.addActionListener(new ButtonListener());
        panel1.add(chooseButton);

        this.add(panel1);
        this.setVisible(true);
    }

    private class ButtonListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            Color c = JColorChooser.showDialog(null,              →40
                "Choose a Color",
                sampleText.getForeground());
            if (c != null)                                        →43
                sampleText.setForeground(c);
        }
    }
}
```

Here are the key points to note while you peruse this program:

- **→ 12** This label's color is set to the value chosen by the user.
- **→ 40** In the `actionPerformed` method of the action listener attached to the button, this statement calls the static `showDialog` method of the `JColorChooser` class to display a Color Chooser dialog box. The color selected by the user is saved in the variable `c`.
- **→ 43** If `c` is `null`, the user cancelled out of the Color Chooser dialog box, so the label's foreground color is unchanged. Otherwise, the label's `setForeground` method is called to set the label's color to the color chosen by the user.