3

In this chapter:
- *Fonts*
- *FontMetrics*
- *Color*
- *SystemColor*
- *Displaying Colors*
- *Using Desktop Colors*

Fonts and Colors

This chapter introduces the `java.awt` classes that are used to work with different fonts and colors. First, we discuss the `Font` class, which determines the font used to display text strings, whether they are drawn directly on the screen (with `drawString()`) or displayed within a component like a text field. The `FontMetrics` class gives you detailed information about a font, which you can use to position text strings intelligently. Next, the `Color` class is used to represent colors and can be used to specify the background color of any object, as well as the foreground color used to display a text string or a shape. Finally, the `SystemColor` class (which is new to Java 1.1) provides access to the desktop color scheme.

3.1 Fonts

An instance of the `Font` class represents a specific font to the system. Within AWT, a font is specified by its name, style, and point size. Each platform that supports Java provides a basic set of fonts; to find the fonts supported on any platform, call `Toolkit.getDefaultToolkit().getFontList()`. This method returns a `String` array of the fonts available. Under Java 1.0, on any platform, the available fonts were: TimesRoman, Helvetica, Courier, Dialog, DialogInput, and ZapfDingbats. For copyright reasons, the list is substantially different in Java 1.1: the available font names are TimesRoman ☆, Serif, Helvetica ☆, SansSerif, Courier ☆, Monospaced, Dialog, and DialogInput. The actual fonts available aren't changing; the deprecated font names are being replaced by non-copyrighted equivalents. Thus, TimesRoman is now Serif, Helvetica is now SansSerif, and Courier is Monospaced. The ZapfDingbats font name has been dropped completely because the characters in this font have official Unicode mappings in the range \u2700 to \u27ff.

NOTE If you desire non-Latin font support with Java 1.1, use the Unicode mappings for the characters. The actual font used is specified in a set of *font.properties* files in the *lib* subdirectory under *java.home*. These localized font files allow you to remap the "Serif", "SansSerif", and "Monospaced" names to different fonts.

The font's `style` is passed with the help of the class variables `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. The combination `Font.BOLD | Font.ITALIC` specifies bold italics.

A font's `size` is represented as an integer. This integer is commonly thought of as a point size; although that's not strictly correct, this book follows common usage and talks about font sizes in points.

It is possible to add additional font names to the system by setting properties. For example, putting the line below in the properties file or a resource file (resource files are new to Java 1.1) defines the name "AvantGarde" as an alias for the font SansSerif:

```
awt.font.avantgarde=SansSerif
```

With this line in the properties file, a Java program can use "AvantGarde" as a font name; when this font is selected, AWT uses the font SansSerif for display. The property name must be all lowercase. Note that we haven't actually added a new font to the system; we've only created a new name for an old font. See the discussion of `getFont()` and `decode()` for more on font properties.

3.1.1 The Font Class

Constants

There are four styles for displaying fonts in Java: plain, bold, italic, and bold italic. Three class constants are used to represent font styles:

public static final int BOLD
: The `BOLD` constant represents a boldface font.

public static final int ITALIC
: The `ITALIC` constant represents an italic font.

public static final int PLAIN
: The `PLAIN` constant represents a plain or normal font.

The combination `BOLD | ITALIC` represents a bold italic font. `PLAIN` combined with either `BOLD` or `ITALIC` represents bold or italic, respectively.

There is no style for underlined text. If you want underlining, you have to do it manually, with the help of `FontMetrics`.

NOTE If you are using Microsoft's SDK, the `com.ms.awt.FontX` class includes direct support for underlined, strike through (line through middle), and outline fonts.

Variables

Three protected variables access the font setting. They are initially set through the `Font` constructor. To read these variables, use the `Font` class's "get" methods.

protected String name
: The name of the font.

protected int size
: The size of the font.

protected int style
: The style of the font. The style is some logical combination of the constants listed previously.

Constructors

public Font (String name, int style, int size)
: There is a single constructor for `Font`. It requires a `name`, `style`, and `size`. `name` represents the name of the font to create, case insensitive.

```
setFont (new Font ("TimesRoman", Font.BOLD | Font.ITALIC, 20));
```

Characteristics

public String getName ()
: The `getName()` method returns the font's logical name. This is the name passed to the constructor for the specific instance of the `Font`. Remember that system properties can be used to alias font names, so the name used in the constructor isn't necessarily the actual name of a font on the system.

public String getFamily ()
: The `getFamily()` method returns the actual name of the font that is being used to display characters. If the font has been aliased to another font, the `getFamily()` method returns the name of the platform-specific font, not the alias. For example, if the constructor was `new Font ("AvantGarde", Font.PLAIN, 10)` and the `awt.font.avantgarde=Helvetica` property is set,

then `getName()` returns AvantGarde, and `getFamily()` returns Helvetica. If nobody set the property, both methods return AvantGarde, and the system uses the default font (since AvantGarde is a nonstandard font).

public int getStyle ()

The `getStyle()` method returns the current style of the font as an integer. Compare this value with the constants `Font.BOLD`, `Font.PLAIN`, and `Font.ITALIC` to see which style is meant. It is easier to use the `isPlain()`, `isBold()`, and `isItalic()` methods to find out the current style. `getStyle()` is more useful if you want to copy the style of some font when creating another.

public int getSize ()

The `getSize()` method retrieves the point size of the font, as set by the size parameter in the constructor. The actual displayed size may be different.

public FontPeer getPeer () ★

The `getPeer()` method retrieves the platform-specific peer object. The object `FontPeer` is a platform-specific subclass of `sun.awt.PlatformFont`. For example, on a Windows 95 platform, this would be an instance of `sun.awt.windows.WFontPeer`.

Styles

public boolean isPlain ()

The `isPlain()` method returns `true` if the current font is neither bold nor italic. Otherwise, it returns `false`.

public boolean isBold ()

The `isBold()` method returns `true` if the current font is either bold or bold and italic. Otherwise, it returns `false`.

public boolean isItalic ()

The `isItalic()` method returns `true` if the current font is either italic or bold and italic. Otherwise, it returns `false`.

Font properties

Earlier, you saw how to use system properties to add aliases for fonts. In addition to adding aliases, you can use system properties to specify which fonts your program will use when it runs. This allows your users to customize their environments to their liking; your program reads the font settings at run-time, rather than using hard-coded settings. The format of the settings in a properties file is:

```
propname=fontname-style-size
```

where `propname` is the name of the property being set, `fontname` is any valid font

name (including aliases), `style` is `plain`, `bold`, `italic`, or `bolditalic`, and `size` represents the desired size for the font. `style` and `size` default to `plain` and 12 points. Order is important; the font's style must always precede its size.

For example, let's say you have three areas on your screen: one for menus, one for labels, and one for input. In the system properties, you allow users to set three properties: `myPackage.myClass.menuFont`, `myPackage.myClass.labelFont`, and `myPackage.myClass.inputFont`. One user sets two:

```
myPackage.myClass.menuFont=TimesRoman-italic-24
myPackage.myClass.inputFont=Helvetica
```

The user has specified a Times font for menus and Helvetica for other input. The property names are up to the developer. The program uses `getFont()` to read the properties and set the fonts accordingly.

NOTE The location of the system properties file depends on the run-time environment and version you are using. Normally, the file goes into a subdirectory of the installation directory, or for environments where users have home directories, in a subdirectory for the user. Sun's HotJava, JDK, and *appletviewer* tools use the *properties* file in the *.hotjava* directory.

Most browsers do not permit modifying properties, so there is no file.

Java 1.1 adds the idea of "resource files," which are syntactically similar to properties files. Resource files are then placed on the server or within a directory found in the `CLASSPATH`. Updating the properties file is no longer recommended.

---

public static Font getFont (String name)

The `getFont()` method gets the font specified by the system property `name`. If `name` is not a valid system property, `null` is returned. This method is implemented by a call to the next version of `getFont()`, with the `defaultFont` parameter set to `null`.

Assuming the properties defined in the previous example, if you call the `getFont()` method with `name` set to `myPackage.myClass.menuFont`, the return value is a 24-point, italic, TimesRoman `Font` object. If called with `name` set to `myPackage.myClass.inputFont`, `getFont()` returns a 12-point, plain Helvetica `Font` object. If called with `myPackage.myClass.labelFont` as `name`, `getFont()` returns `null` because this user did not set the property `myPackage.myClass.labelFont`.

public static Font getFont (String name, Font defaultFont)

The `getFont()` method gets the font specified by the system property `name`. If `name` is not a valid system property, this version of `getFont()` returns the `Font` specified by `defaultFont`. This version allows you to provide defaults in the event the user does not wish to provide his own font settings.

public static Font decode (String name) ★

The `decode()` method provides an explicit means to decipher font property settings, regardless of where the setting comes from. (The `getFont()` method can decipher settings, but only if they're in the system properties file.) In particular, you can use `decode()` to look up font settings in a resource file. The format of `name` is the same as that used by `getFont()`. If the contents of `name` are invalid, a 12-point plain font is returned. To perform the equivalent of `getFont("myPackage.myClass.menuFont")` without using system properties, see the following example. For a more extensive example using resource files, see Appendix A.

```
// Java 1.1 only
InputStream is = instance.getClass().getResourceAsStream("propfile");
Properties p = new Properties();
try {
    p.load (is);
    Font f = Font.decode(p.getProperty("myPackage.myClass.menuFont"));
} catch (IOException e) {
    System.out.println ("error loading props...");
}
```

Miscellaneous methods

public int hashCode ()

The `hashCode()` method returns a hash code for the font. This hash code is used whenever a `Font` object is used as the key in a `Hashtable`.

public boolean equals (Object o)

The `equals()` method overrides the `equals()` method of `Object` to define equality for `Font` objects. Two `Font` objects are equal if their size, style, and name are equal. The following example demonstrates why this is necessary.

```
Font a = new Font ("TimesRoman", Font.PLAIN, 10);
Font b = new Font ("TimesRoman", Font.PLAIN, 10);
// displays false since the objects are different objects
System.out.println (a == b);
// displays true since the objects have equivalent settings
System.out.println (a.equals (b));
```

public String toString ()

The `toString()` method of `Font` returns a string showing the current family, name, style, and size settings. For example:

```
java.awt.Font[family=TimesRoman,name=TimesRoman,style=bolditalic,size=20]
```

3.2 FontMetrics

The abstract `FontMetrics` class provides the tools for calculating the actual width and height of text when displayed on the screen. You can use the results to position objects around text or to provide special effects like shadows and underlining.

Like the `Graphics` class, `FontMetrics` is abstract. The run-time Java platform provides a concrete implementation of `FontMetrics`. You don't have to worry about the actual class; it is guaranteed to implement all the methods of `FontMetrics`. In case you're curious, on a Windows 95 platform, either the class `sun.awt.win32.Win32FontMetrics` (JDK1.0) or the class `sun.awt.windows.WFontMetrics` (JDK1.1) extends `FontMetrics`. On a UNIX/Motif platform, the class is `sun.awt.motif.X11FontMetrics`. With the Macintosh, the class is `sun.awt.macos.MacFontMetrics`. If you're not using the JDK, the class names may be different, but the principle still applies: you don't have to worry about the concrete class.

3.2.1 The FontMetrics Class

Variables

protected Font font

The font whose metrics are contained in this `FontMetrics` object; use the `getFont()` method to get the value.

Constructors

protected FontMetrics (Font font)

There is no visible constructor for `FontMetrics`. Since the class is abstract, you cannot create a `FontMetrics` object. The way to get the `FontMetrics` for a font is to ask for it. Through the current graphics context, call the method `getGraphics().getFontMetrics()` to retrieve the `FontMetrics` for the current font. If a graphics context isn't available, you can get a `FontMetrics` object from the default `Toolkit` by calling the method `Toolkit.getDefaultToolkit().getFontMetrics (aFontObject)`.

Font height

Four variables describe the height of a font: leading (pronounced like the metal), ascent, descent, and height. Leading is the amount of space required between lines of the same font. Ascent is the space above the baseline required by the tallest character in the font. Descent is the space required below the baseline by the lowest descender (the "tail" of a character like "y"). Height is the total of the three: ascent, baseline, and descent. Figure 3-1 shows these values graphically.

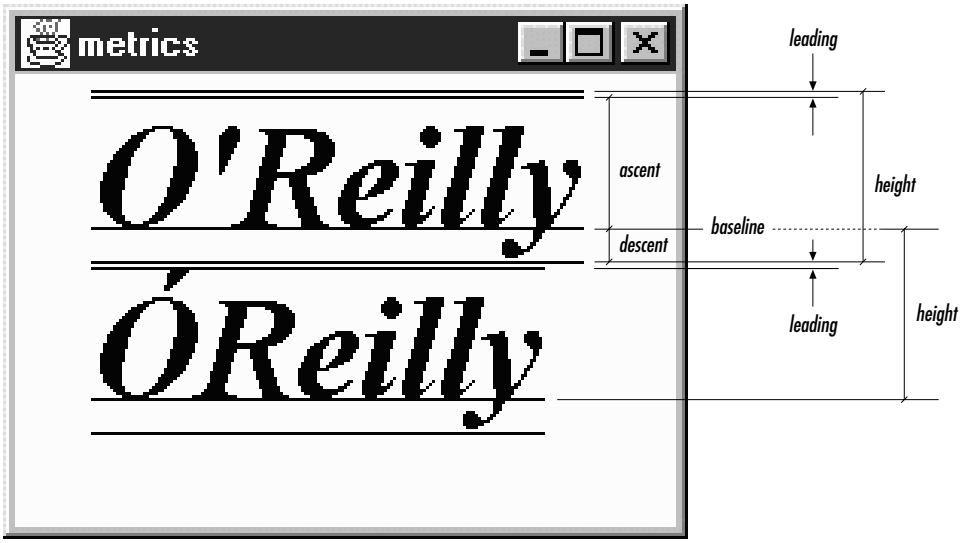

Figure 3–1: Font height metrics

If that were the entire story, it would be simple. Unfortunately, it isn't. Some special characters (for example, capitals with umlauts or accents) are taller than the "tallest" character in the font; so Java defines a value called `maxAscent` to account for these. Similarly, some characters descend below the "greatest" descent, so Java defines a `maxDescent` to handle these cases.

NOTE It seems that on Windows and Macintosh platforms there is no difference between the return values of `getMaxAscent()` and `getAscent()`, or between `getMaxDescent()` and `getDescent()`. On UNIX platforms, they sometimes differ. For developing truly portable applications, the `max` methods should be used where necessary.

---

public int getLeading ()

The `getLeading()` method retrieves the leading required for the `FontMetrics` of the font. The units for this measurement are pixels.

public int getAscent ()

The `getAscent()` method retrieves the space above the baseline required for the tallest character in the font. The units for this measurement are pixels. You cannot get the ascent value for a specific character.

public int getMaxAscent ()

`getMaxAscent()` retrieves the height above the baseline for the character that's really the tallest character in the font, taking into account accents, umlauts, tildes, and other special marks. The units for this measurement are pixels. If you are using only ordinary ASCII characters below 128 (i.e., the English language character set), `getMaxAscent()` is not necessary.

If you're using `getMaxAscent()`, avoid `getHeight()`; `getHeight()` is based on `getAscent()` and doesn't account for extra space.

For some fonts and platforms, `getAscent()` may include the space for the diacritical marks.

public int getDescent ()

The `getDescent()` method retrieves the space below the baseline required for the deepest character for the font. The units for this measurement are pixels. You cannot get the descent value for a specific character.

public int getMaxDescent ()
public int getMaxDecent ()

Some fonts may have special characters that extend farther below the baseline than the value returned by `getDescent()`. `getMaxDescent()` returns the real maximum descent for the font, in pixels. In most cases, you can still use the `getDescent()` method; visually, it is okay for an occasional character to extend into the space between lines. However, if it is absolutely, positively necessary that the descent space does not overlap with the next line's ascent requirements, use `getMaxDescent()` and avoid `getDescent()` and `getHeight()`.

An early beta release of the AWT API included the method `getMaxDecent()`. It is left for compatibility with early beta code. Avoid using it; it is identical to `getMaxDescent()` in every way except spelling. Unfortunately, it is not flagged as deprecated.

public int getHeight ()

The `getHeight()` method returns the sum of `getDescent()`, `getAscent()`, and `getLeading()`. In most cases, this will be the distance between successive baselines when you are displaying multiple lines of text. The height of a font in pixels is not necessarily the size of a font in points.

Don't use `getHeight()` if you are displaying characters with accents, umlauts, and other marks that increase the character's height. In this case, compute the height yourself using the `getMaxAscent()` method. Likewise, you shouldn't use the method `getHeight()` if you are using `getMaxDescent()` instead of `getDescent()`.

Character width

In the horizontal dimension, positioning characters is relatively simple: you don't have to worry about ascenders and descenders, you only have to worry about how far ahead to draw the next character after you have drawn the current one. The "how far" is called the *advance width* of a character. For most cases, the advance width is the actual width plus the intercharacter space. However, it's not a good idea to think in these terms; in many cases, the intercharacter space is actually negative (i.e., the bounding boxes for two adjacent characters overlap). For example, consider an italic font. The top right corner of one character probably extends beyond the character's advance width, overlapping the next character's bounding box. (To see this, look back at Figure 3-1; in particular, look at the *ll* in *O'Reilly*.) If you think purely in terms of the advance width (the amount to move horizontally after drawing a character), you won't run into trouble. Obviously, the advance width depends on the character, unless you're using a fixed width font.

public int charWidth (char character)

This version of the `charWidth()` method returns the advance width of the given `character` in pixels.

public int charWidth (int character)

The `charWidth()` method returns the advance width of the given `character` in pixels. Note that the argument has type `int` rather than `char`. This version is useful when overriding the `Component.keyDown()` method, which gets the integer value of the character pressed as a parameter. With the `KeyEvent` class, you should use the previous version with its `getKeyChar()` method.

public int stringWidth (String string)

The `stringWidth()` method calculates the advance width of the entire `string` in pixels. Among other things, you can use the results to underline or center text within an area of the screen. Example 3-1 and Figure 3-2 show an example that centers several text strings (taken from the command-line arguments) in a `Frame`.

Example 3–1: Centering Text in a Frame

```
import java.awt.*;
public class Center extends Frame {
    static String text[];
    private Dimension dim;
    static public void main (String args[]) {
        if (args.length == 0) {
            System.err.println ("Usage: java Center <some text>");
            return;
        }
        text = args;
        Center f = new Center();
        f.show();
    }
    public void addNotify() {
        super.addNotify();
        int maxWidth = 0;
        FontMetrics fm = getToolkit().getFontMetrics(getFont());
        for (int i=0;i<text.length;i++) {
            maxWidth = Math.max (maxWidth, fm.stringWidth(text[i]));
        }
        Insets inset = insets();
        dim = new Dimension (maxWidth + inset.left + inset.right,
            text.length*fm.getHeight() + inset.top + inset.bottom);
        resize (dim);
    }
    public void paint (Graphics g) {
        g.translate(insets().left, insets().top);
        FontMetrics fm = g.getFontMetrics();
        for (int i=0;i<text.length;i++) {
            int x,y;
            x = (size().width - fm.stringWidth(text[i]))/2;
            y = (i+1)*fm.getHeight()-1;
            g.drawString (text[i], x, y);
        }

    }
}
```

This application extends the `Frame` class. It stores its command-line arguments in the `String` array `text[]`. The `addNotify()` method sizes the frame appropriately. It computes the size needed to display the arguments and resizes the `Frame` accordingly. To compute the width, it takes the longest `stringWidth()` and adds the left and right insets. To compute the height, it takes the current font's height,

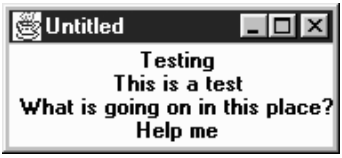

Figure 3–2: Centering text in a frame

multiplies it by the number of lines to display, and adds insets. Then it is up to the `paint()` method to use `stringWidth()` and `getHeight()` to figure out where to put each string.

public int charsWidth (char data[], int offset, int length)

The `charsWidth()` method allows you to calculate the advance width of the char array `data`, without first converting `data` to a `String` and calling the `stringWidth()` method. The `offset` specifies the element of `data` to start with; `length` specifies the number of elements to use. The first element of the array has an `offset` of zero. If `offset` or `length` is invalid, `charsWidth()` throws the run-time exception `ArrayIndexOutOfBoundsException`.

public int bytesWidth (byte data[], int offset, int length)

The `bytesWidth()` method allows you to calculate the advance width of the byte array `data`, without first converting `data` to a `String` and calling the `stringWidth()`method. The `offset` specifies the element of `data` to start with; `length` specifies the number of elements to use. The first element of the array has an `offset` of zero. If `offset` or `length` is invalid, `bytesWidth()` throws the run-time exception `ArrayIndexOutOfBoundsException`.

public int[] getWidths ()

The `getWidths()` method returns an integer array of the advance widths of the first 255 characters in the `FontMetrics` font. `getWidths()` is very useful if you are continually looking up the widths of ASCII characters. Obtaining the widths as an array and looking up individual character widths yourself results in less method invocation overhead than making many calls to `charWidth()`.

public int getMaxAdvance ()

The `getMaxAdvance()` method returns the advance pixel width of the widest character in the font. This allows you to reserve enough space for characters before you know what they are. If you know you are going to display only ASCII characters, you are better off calculating the maximum value returned from `getWidths()`. When unable to determine the width in advance, the method `getMaxAdvance()` returns –1.

Miscellaneous methods

public Font getFont ()

The `getFont()` method returns the specific font for this `FontMetrics` instance.

public String toString ()

The `toString()` method of `FontMetrics` returns a string displaying the current font, ascent, descent, and height. For example:

```
sun.awt.win32.Win32FontMetrics[font=java.awt.Font[family=TimesRoman,
name=TimesRoman,style=bolditalic,size=20]ascent=17, descent=6, height=24]
```

Because this is an abstract class, the concrete implementation could return something different.

3.2.2 Font Display Example

Example 3-2 displays all the available fonts in the different styles at 12 points. The code uses the `FontMetrics` methods to ensure that there is enough space for each line. Figure 3-3 shows the results, using the Java 1.0 font names, on several platforms.

Example 3–2: Font Display

```
import java.awt.*;
public class Display extends Frame {
    static String[] fonts;
    private Dimension dim;
    Display () {
        super ("Font Display");
        fonts = Toolkit.getDefaultToolkit().getFontList();
    }
    public void addNotify() {
        Font f;
        super.addNotify();
        int height   = 0;
        int maxWidth = 0;
        final int vMargin  = 5, hMargin = 5;
        for (int i=0;i<fonts.length;i++) {
            f = new Font (fonts[i], Font.PLAIN, 12);
            height += getHeight (f);
            f = new Font (fonts[i], Font.BOLD, 12);
            height += getHeight (f);
            f = new Font (fonts[i], Font.ITALIC, 12);
            height += getHeight (f);
            f = new Font (fonts[i], Font.BOLD | Font.ITALIC, 12);
            height += getHeight (f);
            maxWidth = Math.max (maxWidth, getWidth (f, fonts[i] + " BOLDITALIC"));
        }
        Insets inset = insets();
        dim = new Dimension (maxWidth + inset.left + inset.right + hMargin,
                             height + inset.top + inset.bottom + vMargin);
```

Example 3–2: Font Display (continued)

```
            resize (dim);
        }
        static public void main (String args[]) {
            Display f = new Display();
            f.show();
        }
        private int getHeight (Font f) {
            FontMetrics fm = Toolkit.getDefaultToolkit().getFontMetrics(f);
            return fm.getHeight();
        }
        private int getWidth (Font f, String s) {
            FontMetrics fm = Toolkit.getDefaultToolkit().getFontMetrics(f);
            return fm.stringWidth(s);
        }
        public void paint (Graphics g) {
            int x = 0;
            int y = 0;
            g.translate(insets().left, insets().top);
            for (int i=0;i<fonts.length;i++) {
                Font plain = new Font (fonts[i], Font.PLAIN, 12);
                Font bold = new Font (fonts[i], Font.BOLD, 12);
                Font italic = new Font (fonts[i], Font.ITALIC, 12);
                Font bolditalic = new Font (fonts[i], Font.BOLD | Font.ITALIC, 12);
                g.setFont (plain);
                y += getHeight (plain);
                g.drawString (fonts[i] + " PLAIN", x, y);
                g.setFont (bold);
                y += getHeight (bold);
                g.drawString (fonts[i] + " BOLD", x, y);
                g.setFont (italic);
                y += getHeight (italic);
                g.drawString (fonts[i] + " ITALIC", x, y);
                g.setFont (bolditalic);
                y += getHeight (bolditalic);
                g.drawString (fonts[i] + " BOLDITALIC", x, y);
            }
            resize (dim);
        }
    }
```

3.3 Color

Not so long ago, color was a luxury; these days, color is a requirement. A program that uses only black and white seems hopelessly old fashioned. AWT's `Color` class lets you define and work with `Color` objects. When we discuss the `Component` class (see Chapter 5, *Components*), you will see how to use these color objects, and our discussion of the `SystemColor` subclass (new to Java 1.1; discussed later in this chapter) shows you how to control the colors that are painted on the screen.

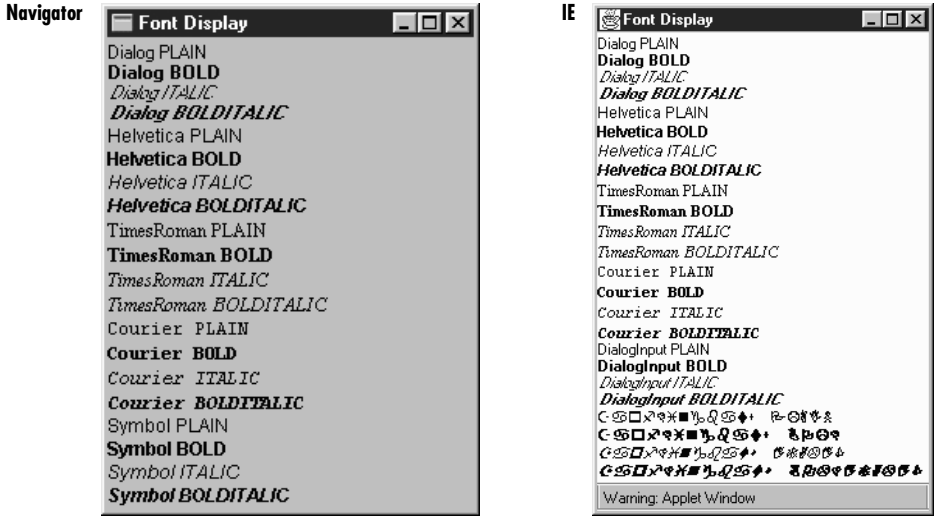

Figure 3–3: Fonts available with the Netscape Navigator 3.0 and Internet Explorer 3.0

A few words of warning: while colors give you the opportunity to make visually pleasing applications, they also let you do things that are incredibly ugly. Resist the urge to go overboard with your use of color; it's easy to make something hideous when you are trying to use every color in the palette. Also, realize that colors are fundamentally platform dependent, and in a very messy way. Java lets you use the same `Color` objects on any platform, but it can't guarantee that every display will treat the color the same way; the result depends on everything from your software to the age of your monitor. What looks pink on one monitor may be red on another. Furthermore, when running in an environment with a limited palette, AWT picks the available color that is closest to what you requested. If you really care about appearance, there is no substitute for testing.

3.3.1 Color Methods

Constants

The `Color` class has predefined constants (all of type `public static final Color`) for frequently used colors. These constants, their RGB values, and their HSB values (hue, saturation, brightness) are given in Table 3-1.

Table 3–1: Comparison of RGB and HSB Colors

Color	Red	Green	Blue	Hue	Saturation	Brightness
black	0	0	0	0	0	0
blue	0	0	255	.666667	1	1
cyan	0	255	255	.5	1	1
darkGray	64	64	64	0	0	.25098
gray	128	128	128	0	0	.501961
green	0	255	0	.333333	1	1
lightGray	192	192	192	0	0	.752941
magenta	255	0	255	.833333	1	1
orange	255	200	0	.130719	1	1
pink	255	175	175	0	.313726	1
red	255	0	0	0	1	1
white	255	255	255	0	0	1
yellow	255	255	0	.166667	1	1

These constants are used like any other class variable: for example, `Color.red` is a constant `Color` object representing the color red. Many other color constants are defined in the `SystemColor` class.

Constructors

When you're not using a predefined constant, you create `Color` objects by specifying the color's red, green, and blue components. Depending on which constructor you use, you can specify the components as integers between 0 and 255 (most intense) or as floating point intensities between 0.0 and 1.0 (most intense). The result is a 24-bit quantity that represents a color. The remaining 8 bits are used to represent transparency: that is, if the color is painted on top of something, does whatever was underneath show through? The `Color` class doesn't let you work with the transparency bits; all `Color` objects are opaque. However, you can use transparency when working with images; this topic is covered in Chapter 12, *Image Processing*.

public Color (int red, int green, int blue)

This constructor is the most commonly used. You provide the specific `red`, `green`, and `blue` values for the color. Valid values for `red`, `green`, and `blue` are between 0 and 255. The constructor examines only the low-order byte of the integer and ignores anything outside the range, including the sign bit.

public Color (int rgb)

This constructor allows you to combine all three variables in one parameter, `rgb`. Bits 16–23 represent the red component, and bits 8–15 represent the green component. Bits 0–7 represent the blue component. Bits 24–31 are ignored. Going from three bytes to one integer is fairly easy:

```
(((red & 0xFF) << 16 ) | ((green & 0xFF) << 8) | ((blue & 0xFF) << 0))
```

public Color (float red, float green, float blue)

This final constructor allows you to provide floating point values between 0.0 and 1.0 for each of `red`, `green`, and `blue`. Values outside of this range yield unpredictable results.

Settings

public int getRed ()

The `getRed()` method retrieves the current setting for the red component of the color.

public int getGreen ()

The `getGreen()` method retrieves the current setting for the green component of the color.

public int getBlue ()

The `getBlue()` method retrieves the current setting for the blue component of the color.

public int getRGB ()

The `getRGB()` method retrieves the current settings for red, green, and blue in one combined value. Bits 16–23 represent the red component. Bits 8–15 represent the green component. Bits 0–7 represent the blue component. Bits 24–31 are the transparency bits; they are always `0xff` (opaque) when using the default RGB `ColorModel`.

public Color brighter ()

The `brighter()` method creates a new `Color` that is somewhat brighter than the current color. This method is useful if you want to highlight something on the screen.

NOTE Black does not get any brighter.

---

public Color darker ()

The `darker()` method returns a new `Color` that is somewhat darker than the current color. This method is useful if you are trying to de-emphasize an object on the screen. If you are creating your own `Component`, you can use a

`darker()` `Color` to mark it inactive.

Color properties

`Color` properties are very similar to `Font` properties. You can use system properties (or resource files) to allow users to select colors for your programs. The settings have the form `0xRRGGBB`, where `RR` is the red component of the color, `GG` represents the green component, and `BB` represents the blue component. `0x` indicates that the number is in hexadecimal. If you (or your user) are comfortable using decimal values for colors (0x112233 is 1122867 in decimal), you can, but then it is harder to see the values of the different components.

NOTE The location of the system properties file depends on the run-time environment and version you are using. Ordinarily, the file will go into a subdirectory of the installation directory or, for environment's where users have home directories, in a subdirectory for the user. Sun's HotJava, JDK, and *appletviewer* tools use the *properties* file in the *.hotjava* directory.

Most browsers do not permit modifying properties, so there is no file.

Java 1.1 adds the idea of "resource files," which are syntactically similar to properties files. Resource files are then placed on the server or within a directory found in the `CLASSPATH`. Updating the properties file is no longer recommended.

For example, consider a screen that uses four colors: one each for the foreground, the background, inactive components, and highlighted text. In the system properties file, you allow users to select colors by setting the following properties:

```
myPackage.myClass.foreground
myPackage.myClass.background
myPackage.myClass.inactive
myPackage.myClass.highlight
```

One particular user set two:

```
myPackage.myClass.foreground=0xff00ff        #magenta
myPackage.myClass.background=0xe0e0e0        #light gray
```

These lines tell the program to use magenta as the foreground color and light gray for the background. The program will use its default colors for inactive components and highlighted text.

public static Color getColor (String name)

The `getColor()` method gets the color specified by the system property `name`. If `name` is not a valid system property, `getColor()` returns `null`. If the property value does not convert to an integer, `getColor()` returns `null`.

For the properties listed above, if you call `getColor()` with `name` set to the property `myPackage.myClass.foreground`, it returns a magenta `Color` object. If called with `name` set to `myPackage.myClass.inactive`, `getColor()` returns `null`.

public static Color getColor (String name, Color defaultColor)

The `getColor()` method gets the color specified by the system property `name`. This version of the `getColor()` method returns `defaultColor` if `name` is not a valid system property or the property's value does not convert to an integer.

For the previous example, if `getColor()` is called with `name` set to `myPackage.myClass.inactive`, the `getColor()` method returns the value of `defaultColor`. This allows you to provide defaults for properties the user doesn't wish to set explicitly.

public static Color getColor (String name, int defaultColor)

This `getColor()` method gets the color specified by the system property `name`. This version of the `getColor()` method returns `defaultColor` if `name` is not a valid system property or the property's value does not convert to an integer. The default color is specified as an integer in which bits 16–23 represent the red component, 8–15 represent the green component, and 0–7 represent the blue component. Bits 24–31 are ignored. If the property value does not convert to an integer, `defaultColor` is returned.

public static Color decode (String name) ★

The `decode()` method provides an explicit means to decipher color property settings, regardless of where the setting comes from. (The `getColor()` method can decipher settings but only if they're in the system properties file.) In particular, you can use `decode()` to look up color settings in a resource file. The format of `name` is the same as that used by `getColor()`. If the contents of `name` do not translate to a 24-bit integer, the `NumberFormatException` run-time exception is thrown. To perform the equivalent of `getColor("myPackage.myClass.foreground")`, without using system properties, see the following example. For a more extensive example using resource files, see Appendix A.

```
// Java 1.1 only
InputStream is = instance.getClass().getResourceAsStream("propfile");
Properties p = new Properties();
try {
    p.load (is);
    Color c = Color.decode(p.getProperty("myPackage.myClass.foreground"));
} catch (IOException e) {
```

```
        System.out.println ("error loading props...");
    }
```

Hue, saturation, and brightness

So far, the methods we have seen work with a color's red, green, and blue components. There are many other ways to represent colors. This group of methods allows you to work in terms of the HSB (hue, saturation, brightness) model. Hue represents the base color to work with: working through the colors of the rainbow, red is represented by numbers immediately above 0; magenta is represented by numbers below 1; white is 0; and black is 1. Saturation represents the color's purity, ranging from completely unsaturated (either white or black depending upon brightness) to totally saturated (just the base color present). Brightness is the desired level of luminance, ranging from black (0) to the maximum amount determined by the saturation level.

public static float[] RGBtoHSB (int red, int green, int blue, float[] hsbvalues)

The `RGBtoHSB()` method allows you to convert a specific `red`, `green`, `blue` value to the hue, saturation, and brightness equivalent. `RGBtoHSB()` returns the results in two different ways: the parameter `hsbvalues` and the method's return value. The values of these are the same. If you do not want to pass an `hsbvalues` array parameter, pass `null`. In both the parameter and the return value, the three components are placed in the array as follows:

`hsbvalues[0]`	*contains hue*
`hsbvalues[1]`	*contains saturation*
`hsbvalues[2]`	*contains brightness*

public static Color getHSBColor (float hue, float saturation, float brightness)

The `getHSBColor()` method creates a `Color` object by using `hue`, `saturation`, and `brightness` instead of `red`, `green`, and `blue` values.

public static int HSBtoRGB (float hue, float saturation, float brightness)

The `HSBtoRGB()` method converts a specific `hue`, `saturation`, and `brightness` to a `Color` and returns the `red`, `green`, and `blue` values as an integer. As with the constructor, bits 16–23 represent the red component, 8–15 represent the green component, and 0–7 represent the blue component. Bits 24–31 are ignored.

Miscellaneous methods

public int hashCode ()

The `hashCode()` method returns a hash code for the color. The hash code is used whenever a color is used as a key in a `Hashtable`.

public boolean equals (Object o)

The `equals()` method overrides the `equals()` method of the `Object` to define equality for `Color` objects. Two `Color` objects are equivalent if their `red`, `green`, and `blue` values are equal.

public String toString ()

The `toString()` method of `Color` returns a string showing the color's red, green, and blue settings. For example `System.out.println (Color.orange)` would result in the following:

```
java.awt.Color[r=255,g=200,b=0]
```

3.4 SystemColor

In Java 1.1, AWT provides access to desktop color schemes, or *themes*. To give you an idea of how these themes work, with the Windows Standard scheme for the Windows 95 desktop, buttons have a gray background with black text. If you use the control panel to change to a High Contrast Black scheme, the button's background becomes black and the text white. Prior to 1.1, Java didn't know anything about desktop colors: all color values were hard coded. If you asked for a particular shade of gray, you got that shade, and that was it; applets and applications had no knowledge of the desktop color scheme in effect, and therefore, wouldn't change in response to changes in the color scheme.

Starting with Java 1.1, you can write programs that react to changes in the color scheme: for example, a button's color will change automatically when you use the control panel to change the color scheme. To do so, you use a large number of constants that are defined in the `SystemColor` class. Although these constants are `public static final`, they actually have a very strange behavior. Your program is not allowed to modify them (like any other constant). However, their initial values are loaded at run-time, and their values may change, corresponding to changes in the color scheme. This has one important consequence for programmers: you should not use `equals()` to compare a `SystemColor` with a "regular" `Color`; use the `getRGB()` methods of the colors you are comparing to ensure that you compare the current color value.* Section 3.6 contains a usage example.

* The omission of an `equals()` method that can properly compare a `SystemColor` with a `Color` is unfortunate.

Because `SystemColor` is a subclass of `Color`, you can use a `SystemColor` anywhere you can use a `Color` object. You will never create your own `SystemColor` objects; there is no public constructor. The only objects in this class are the twenty or so `SystemColor` constants.

3.4.1 SystemColor Methods

Constants

There are two sets of constants within `SystemColor`. The first set provides names for indices into the internal system color lookup table; you will probably never need to use these. All of them have corresponding constants in the second set, except `SystemColor.NUM_COLORS`, which tells you how many `SystemColor` constants are in the second set.

public final static int ACTIVE_CAPTION ★
public final static int ACTIVE_CAPTION_BORDER ★
public final static int ACTIVE_CAPTION_TEXT ★
public final static int CONTROL ★
public final static int CONTROL_DK_SHADOW ★
public final static int CONTROL_HIGHLIGHT ★
public final static int CONTROL_LT_HIGHLIGHT ★
public final static int CONTROL_SHADOW ★
public final static int CONTROL_TEXT ★
public final static int DESKTOP ★
public final static int INACTIVE_CAPTION ★
public final static int INACTIVE_CAPTION_BORDER ★
public final static int INACTIVE_CAPTION_TEXT ★
public final static int INFO ★
public final static int INFO_TEXT ★
public final static int MENU ★
public final static int MENU_TEXT ★
public final static int NUM_COLORS ★
public final static int SCROLLBAR ★
public final static int TEXT ★
public final static int TEXT_HIGHLIGHT ★
public final static int TEXT_HIGHLIGHT_TEXT ★
public final static int TEXT_INACTIVE_TEXT ★
public final static int TEXT_TEXT ★
public final static int WINDOW ★

public final static int WINDOW_BORDER ★
public final static int WINDOW_TEXT ★

The second set of constants is the set of `SystemColors` you use when creating `Component` objects, to ensure they appear similar to other objects in the user's desktop environment. By using these symbolic constants, you can create new objects that are well integrated into the user's desktop environment, making it easier for the user to work with your program.

public final static SystemColor activeCaption ★
The `activeCaption` color represents the background color for the active window's title area. This is automatically set for you when you use `Frame`.

public final static SystemColor activeCaptionBorder ★
The `activeCaptionBorder` color represents the border color for the active window.

public final static SystemColor activeCaptionText ★
The `activeCaptionText` color represents the text color to use for the active window's title.

public final static SystemColor control ★
The `control` color represents the background color for the different components. If you are creating your own `Component` by subclassing `Canvas`, this should be the background color of the new object.

public final static SystemColor controlDkShadow ★
The `controlDkShadow` color represents a dark shadow color to be used with `control` and `controlShadow` to simulate a three-dimensional appearance. Ordinarily, when not depressed, the `controlDkShadow` should be used for the object's bottom and right edges. When depressed, `controlDkShadow` should be used for the top and left edges.

public final static SystemColor controlHighlight ★
The `controlHighlight` color represents an emphasis color for use in an area or an item of a custom component.

public final static SystemColor controlLtHighlight ★
The `controlLtHighlight` color represents a lighter emphasis color for use in an area or an item of a custom component.

public final static SystemColor controlShadow ★
The `controlShadow` color represents a light shadow color to be used with `control` and `controlDkShadow` to simulate a three-dimensional appearance. Ordinarily, when not depressed, the `controlShadow` should be used for the top and left edges. When depressed, `controlShadow` should be used for the bottom and right edges.

public final static SystemColor controlText ★

The `controlText` color represents the text color of a component. Before drawing any text in your own components, you should change the color to `controlText` with a statement like this:

```
g.setColor(SystemColor.controlText);
```

public final static SystemColor desktop ★

The `desktop` color represents the background color of the desktop workspace.

public final static SystemColor inactiveCaption ★

The `inactiveCaption` color represents the background color for an inactive window's title area.

public final static SystemColor inactiveCaptionBorder ★

The `inactiveCaptionBorder` color represents the border color for an inactive window.

public final static SystemColor inactiveCaptionText ★

The `inactiveCaptionText` color represents the text color to use for each inactive window's title.

public final static SystemColor info ★

The `info` color represents the background color for mouse-over help text. When a mouse dwells over an object, any pop-up help text should be displayed in an area of this color. In the Microsoft Windows world, these are also called "tool tips."

public final static SystemColor infoText ★

The `infoText` color represents the text color for mouse-over help text.

public final static SystemColor menu ★

The `menu` color represents the background color of deselected `MenuItem`-like objects. When the menu is selected, the `textHighlight` color is normally the background color.

public final static SystemColor menuText ★

The `menuText` color represents the color of the text on deselected `MenuItem`-like objects. When a menu is selected, the `textHighlightText` color is normally the text color. If the menu happens to be inactive, `textInactiveText` would be used.

public final static SystemColor scrollbar ★

The `scrollbar` color represents the background color for scrollbars. This color is used by default with `Scrollbar`, `ScrollPane`, `TextArea`, and `List` objects.

public final static SystemColor textHighlight ★

The `textHighlight` color represents the background color of highlighted text; for example, it is used for the selected area of a `TextField` or a selected `MenuItem`.

public final static SystemColor textHighlightText ★

The `textHighlightText` color represents the text color of highlighted text.

public final static SystemColor textInactiveText ★

The `textInactiveText` color represents the text color of an inactive component.

public final static SystemColor textText ★

The `textText` color represents the color of text in `TextComponent` objects.

public final static SystemColor window ★

The `window` color represents the background color of the window's display area. For an applet, this would be the display area specified by the `WIDTH` and `HEIGHT` values of the `<APPLET>` tag (`setBackground(SystemColor.window)`), although you would probably use it more for the background of a `Frame`.

public final static SystemColor windowBorder ★

The `windowBorder` color represents the color of the borders around a window. With AWT, instances of `Window` do not have borders, but instances of `Frame` and `Dialog` do.

public final static SystemColor windowText ★

The `windowText` color represents the color of the text drawn within the window.

NOTE Every platform does not fully support every system color. However, on platforms that do not provide natural values for some constants, Java selects reasonable alternate colors.

If you are going to be working only with Java's prefabricated components (`Button`, `List`, etc.), you don't have to worry about system colors; the component's default colors will be set appropriately. You are most likely to use system colors if you are creating your own components. In this case, you will use system colors to make your component emulate the behavior of other components; for example, you will use `controlText` as the color for drawing text, `activeCaption` as the background for the caption of an active window, and so on.

Constructors

There are no public constructors for `SystemColor`. If you need to create a new color, use the `Color` class described previously.

Miscellaneous methods

public int getRGB ()

The `getRGB()` method retrieves the current settings for red, green, and blue in one combined value, like `Color`. However, since the color value is dynamic, `getRGB()` needs to look up the value in an internal table. Therefore, `SystemColor` overrides `Color.getRGB()`.

public String toString ()

The `toString()` method of `SystemColor` returns a string showing the system color's index into its internal table. For example, the following string is returned by `SystemColor.text.toString()`:

```
java.awt.SystemColor[i=12]
```

3.5 Displaying Colors

Example 3-3 displays the predefined colors on the screen in a series of filled rectangles. When you press a mouse button, they appear brighter. When you press a key, they appear darker. (Event handling is fully explained in Chapter 4, *Events*.) Figure 3-4 shows the results, although it doesn't look very impressive in black and white.

Example 3–3: Color Display

```
import java.awt.*;
public class ColorDisplay extends Frame {
    int width, height;
    static Color colors[] =
                {Color.black, Color.blue, Color.cyan, Color.darkGray,
                Color.gray, Color.green, Color.lightGray, Color.magenta,
                Color.orange, Color.pink, Color.red, Color.white,
                Color.yellow};
    ColorDisplay () {
        super ("ColorDisplay");
        setBackground (Color.white);
    }
    static public void main (String args[]) {
        ColorDisplay f = new ColorDisplay();
        f.resize (300,300);
        f.show();
    }
    public void paint (Graphics g) {
        g.translate (insets().left, insets().top);
        if (width == 0) {
```

Example 3–3: Color Display (continued)

```
            Insets inset = insets();
            width  = (size().width - inset.right - inset.left) / 3;
            height = (size().height - inset.top - inset.bottom) / 5;
        }
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 5; j++) {
                if ((i == 2) && (j >= 3)) break;
                g.setColor (colors[i*5+j]);
                g.fillRect (i*width, j*height, width, height);
            }
        }
    }
    public boolean keyDown (Event e, int c) {
        for (int i=0;i<colors.length;i++)
            colors[i] = colors[i].darker();
        repaint();
        return true;
    }
    public boolean mouseDown (Event e, int x, int y) {
        for (int i=0;i<colors.length;i++)
            colors[i] = colors[i].brighter();
        repaint();
        return true;
    }
}
```

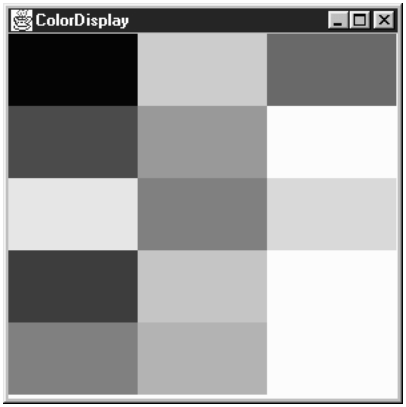

Figure 3–4: A color display

3.6 Using Desktop Colors

Example 3-4 demonstrates how to use the desktop color constants introduced in Java 1.1. If you run this example under an earlier release, an uncatchable class verifier error will occur.

NOTE Notice that the border lines are drawn from `0` to `width-1` or `height-1`. This is to draw lines of length `width` and `height`, respectively.

Example 3–4: Desktop Color Usage

```
// Java 1.1 only
import java.awt.*;
public class TextBox3D extends Canvas {
    String text;
    public TextBox3D (String s, int width, int height) {
        super();
        text=s;
        setSize(width, height);
    }
    public synchronized void paint (Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        Dimension size=getSize();
        int x = (size.width - fm.stringWidth(text))/2;
        int y = (size.height - fm.getHeight())/2;
        g.setColor (SystemColor.control);
        g.fillRect (0, 0, size.width, size.height);
        g.setColor (SystemColor.controlShadow);
        g.drawLine (0, 0, 0, size.height-1);
        g.drawLine (0, 0, size.width-1, 0);
        g.setColor (SystemColor.controlDkShadow);
        g.drawLine (0, size.height-1, size.width-1, size.height-1);
        g.drawLine (size.width-1, 0, size.width-1, size.height-1);
        g.setColor (SystemColor.controlText);
        g.drawString (text, x, y);
    }
}
```