



Graphics and Java2D

Objectives

- To understand graphics contexts and graphics objects.
- To understand and be able to manipulate colors.
- To understand and be able to manipulate fonts.
- To use **Graphics** methods to draw lines, rectangles, rectangles with rounded corners, three-dimensional rectangles, ovals, arcs and polygons.
- To use methods of class **Graphics2D** from the Java2D API to draw lines, rectangles, rectangles with rounded corners, ellipses, arcs and general paths.
- To be able to specify **Paint** and **Stroke** characteristics of shapes displayed with **Graphics2D**.

One picture is worth ten thousand words.

Chinese proverb

Treat nature in terms of the cylinder, the sphere, the cone, all in perspective.

Paul Cezanne

Nothing ever becomes real till it is experienced—even a proverb is no proverb to you till your life has illustrated it.

John Keats

A picture shows me at a glance what it takes dozens of pages of a book to expound.

Ivan Sergeyevich Turgenev



Outline

- 11.1 Introduction
- 11.2 Graphics Contexts and Graphics Objects
- 11.3 Color Control
- 11.4 Font Control
- 11.5 Drawing Lines, Rectangles and Ovals
- 11.6 Drawing Arcs
- 11.7 Drawing Polygons and Polylines
- 11.8 The Java2D API
- 11.9 Java2D Shapes
- 11.10 (Optional Case Study) Thinking About Objects: Designing Interfaces with the UML

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

11.1 Introduction

In this chapter, we overview several of Java's capabilities for drawing two-dimensional shapes, controlling colors and controlling fonts. One of Java's initial appeals was its support for graphics that enabled Java programmers to visually enhance their applets and applications. Java now contains many more sophisticated drawing capabilities as part of the *Java2D API*. This chapter begins with an introduction to many of the original drawing capabilities of Java. Next, we present several of the new and more powerful Java2D capabilities, such as controlling the style of lines used to draw shapes and controlling how shapes are filled with color and patterns.

Figure 11.1 shows a portion of the Java class hierarchy that includes several of the basic graphics classes and Java2D API classes and interfaces covered in this chapter. Class **Color** contains methods and constants for manipulating colors. Class **Font** contains methods and constants for manipulating fonts. Class **FontMetrics** contains methods for obtaining font information. Class **Polygon** contains methods for creating polygons. Class **Graphics** contains methods for drawing strings, lines, rectangles and other shapes. The bottom half of the figure lists several classes and interfaces from the Java2D API. Class **BasicStroke** helps specify the drawing characteristics of lines. Classes **GradientPaint** and **TexturePaint** help specify the characteristics for filling shapes with colors or patterns. Classes **GeneralPath**, **Arc2D**, **Ellipse2D**, **Line2D**, **Rectangle2D** and **RoundRectangle2D** define a variety of Java2D shapes.

To begin drawing in Java, we must first understand Java's *coordinate system* (Figure 11.2), which is a scheme for identifying every possible point on the screen. By default, the upper-left corner of a GUI component (such as an applet or a window) has the coordinates (0, 0). A coordinate pair is composed of an *x-coordinate* (the *horizontal coordinate*) and a *y-coordinate* (the *vertical coordinate*). The *x-coordinate* is the horizontal distance moving right from the upper-left corner. The *y-coordinate* is the vertical distance moving down from the upper-left corner. The *x-axis* describes every horizontal coordinate, and the *y-axis* describes every vertical coordinate.

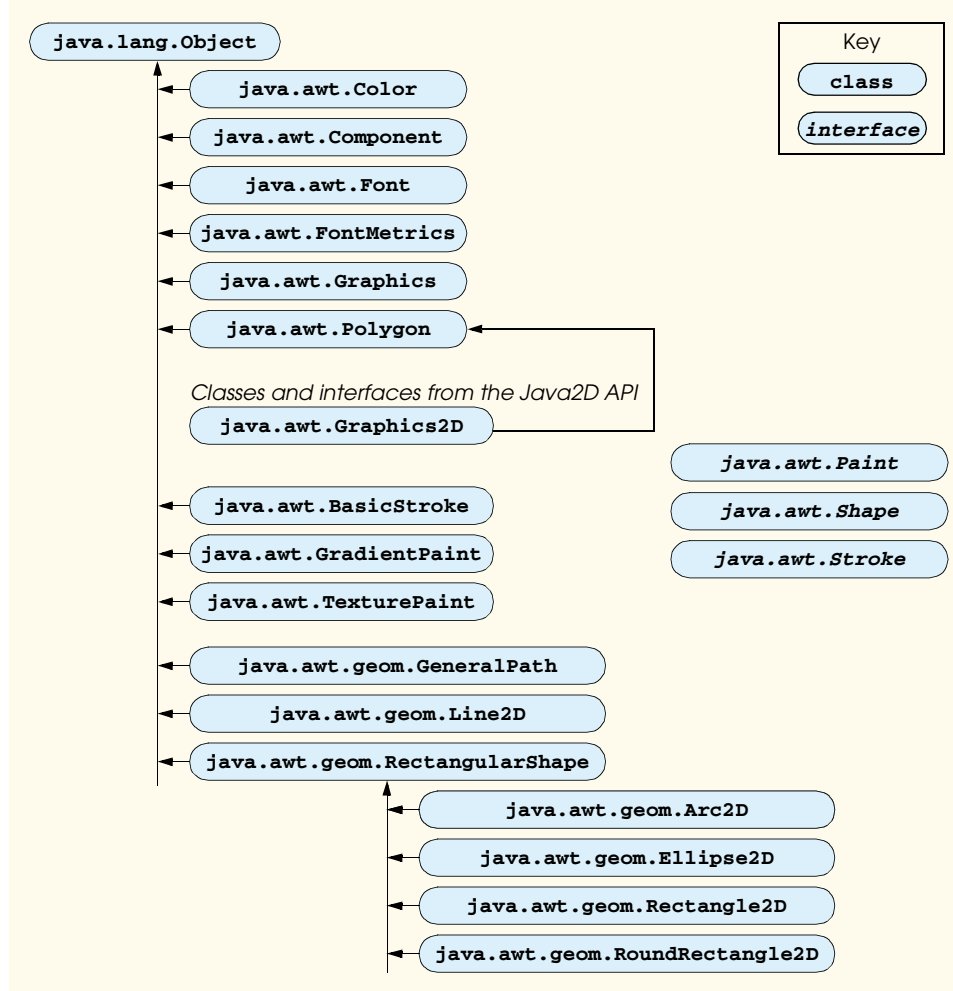


Fig. 11.1 Some classes and interfaces used in this chapter from Java's original graphics capabilities and from the Java2D API.



Software Engineering Observation 11.1

The upper-left coordinate (0, 0) of a window is actually behind the title bar of the window. For this reason, drawing coordinates should be adjusted to draw inside the borders of the window. Class **Container** (a superclass of all windows in Java) has method **getInsets**, which returns an **Insets** object (package **java.awt**) for this purpose. An **Insets** object has four **public** members—**top**, **bottom**, **left** and **right**—that represent the number of pixels from each edge of the window to the drawing area for the window.

Text and shapes are displayed on the screen by specifying coordinates. Coordinate units are measured in *pixels*. A pixel is a display monitor's smallest unit of resolution.

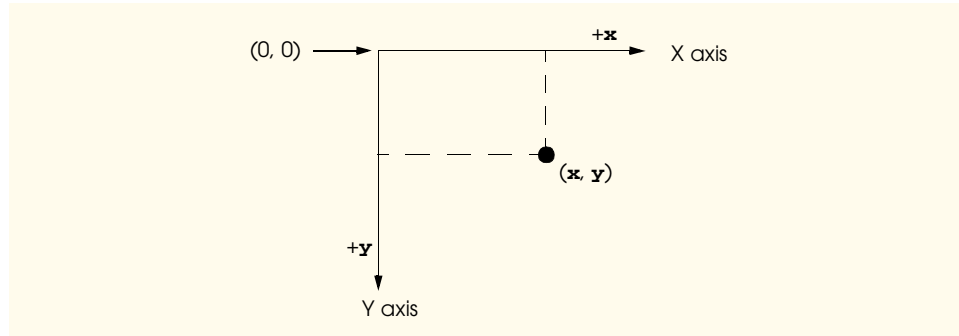


Fig. 11.2 Java coordinate system. Units are measured in pixels.



Portability Tip 11.1

Different display monitors have different resolutions (i.e., the density of pixels varies). This can cause graphics to appear to be different sizes on different monitors.

11.2 Graphics Contexts and Graphics Objects

A Java *graphics context* enables drawing on the screen. A **Graphics** object manages a graphics context by controlling how information is drawn. Graphics objects contain methods for drawing, font manipulation, color manipulation and the like. Every applet we have seen in the text that performs drawing on the screen has used the **Graphics** object **g** (the argument to the applet's **paint** method) to manage the applet's graphics context. In this chapter, we demonstrate drawing in applications. However, every technique shown here can be used in applets.

The **Graphics** class is an **abstract** class (i.e., **Graphics** objects cannot be instantiated). This contributes to Java's portability. Because drawing is performed differently on each platform that supports Java, there cannot be one class that implements drawing capabilities on all systems. For example, the graphics capabilities that enable a PC running Microsoft Windows to draw a rectangle are different from the graphics capabilities that enable a UNIX workstation to draw a rectangle—and those are both different from the graphics capabilities that enable a Macintosh to draw a rectangle. When Java is implemented on each platform, a derived class of **Graphics** is created that actually implements all the drawing capabilities. This implementation is hidden from us by the **Graphics** class, which supplies the interface that enables us to write programs that use graphics in a platform-independent manner.

Class **Component** is the superclass for many of the classes in the **java.awt** package (we discuss class **Component** in Chapter 12). **Component** method **paint** takes a **Graphics** object as an argument. This object is passed to the **paint** method by the system when a **paint** operation is required for a **Component**. The header for the **paint** method is

```
public void paint( Graphics g )
```

The **Graphics** object **g** receives a reference to an object of the system's derived **Graphics** class. The preceding method header should look familiar to you—it is the same one

we have been using in our applet classes. Actually, the **Component** class is an indirect base class of class **JApplet**—the superclass of every applet in this book. Many capabilities of class **JApplet** are inherited from class **Component**. The paint method defined in class **Component** does nothing by default—it must be overridden by the programmer.

The paint method is seldom called directly by the programmer because drawing graphics is an *event-driven process*. When an applet executes, the paint method is automatically called (after calls to the **JApplet**'s **init** and **start** methods). For paint to be called again, an *event* must occur (such as covering and uncovering the applet). Similarly, when any **Component** is displayed, that **Component**'s **paint** method is called.

If the programmer needs to call **paint**, a call is made to the **Component** class **repaint** method. Method **repaint** requests a call to the **Component** class **update** method as soon as possible to clear the **Component**'s background of any previous drawing, then **update** calls **paint** directly. The repaint method is frequently called by the programmer to force a paint operation. Method **repaint** should not be overridden, because it performs some system-dependent tasks. The **update** method is seldom called directly and sometimes overridden. Overriding the **update** method is useful for “smoothing” animations (i.e., reducing “flicker”) as we will discuss in Chapter 18, Multimedia. The headers for **repaint** and **update** are

```
public void repaint()
public void update( Graphics g )
```

Method **update** takes a **Graphics** object as an argument, which is supplied automatically by the system when **update** is called.

In this chapter, we focus on the **paint** method. In the next chapter, we concentrate more on the event-driven nature of graphics and discuss the **repaint** and **update** methods in more detail. We also discuss in that chapter class **JComponent**—a superclass of many GUI components in package **javax.swing**. Subclasses of **JComponent** typically paint from their **paintComponent** methods.

11.3 Color Control

Colors enhance the appearance of a program and help convey meaning. For example, a traffic light has three different color lights—red indicates stop, yellow indicates caution and green indicates go.

Class **Color** defines methods and constants for manipulating colors in a Java program. The predefined color constants are summarized in Fig. 11.3, and several color methods and constructors are summarized in Fig. 11.4. Note that two of the methods in Fig. 11.4 are **Graphics** methods that are specific to colors.

Every color is created from a red, a green and a blue component. Together these components are called *RGB values*. All three RGB components can be integers in the range from 0 to 255, or all three RGB parts can be floating-point values in the range 0.0 to 1.0. The first RGB part defines the amount of red, the second defines the amount of green and the third defines the amount of blue. The larger the RGB value, the greater the amount of that particular color. Java enables the programmer to choose from $256 \times 256 \times 256$ (or approximately 16.7 million) colors. However, not all computers are capable of displaying all these colors. If this is the case, the computer will display the closest color it can.

Color Constant	Color	RGB value
<code>public final static Color orange</code>	orange	255, 200, 0
<code>public final static Color pink</code>	pink	255, 175, 175
<code>public final static Color cyan</code>	cyan	0, 255, 255
<code>public final static Color magenta</code>	magenta	255, 0, 255
<code>public final static Color yellow</code>	yellow	255, 255, 0
<code>public final static Color black</code>	black	0, 0, 0
<code>public final static Color white</code>	white	255, 255, 255
<code>public final static Color gray</code>	gray	128, 128, 128
<code>public final static Color lightGray</code>	light gray	192, 192, 192
<code>public final static Color darkGray</code>	dark gray	64, 64, 64
<code>public final static Color red</code>	red	255, 0, 0
<code>public final static Color green</code>	green	0, 255, 0
<code>public final static Color blue</code>	blue	0, 0, 255

Fig. 11.3 `Color` class `static` constants and RGB values

Method	Description
<code>public Color(int r, int g, int b)</code>	Creates a color based on red, green and blue contents expressed as integers from 0 to 255.
<code>public Color(float r, float g, float b)</code>	Creates a color based on red, green and blue contents expressed as floating-point values from 0.0 to 1.0.
<code>public int getRed() // Color class</code>	Returns a value between 0 and 255 representing the red content.
<code>public int getGreen() // Color class</code>	Returns a value between 0 and 255 representing the green content.
<code>public int getBlue() // Color class</code>	Returns a value between 0 and 255 representing the blue content.
<code>public Color getColor() // Graphics class</code>	Returns a <code>Color</code> object representing the current color for the graphics context.
<code>public void setColor(Color c) // Graphics class</code>	Sets the current color for drawing with the graphics context.

Fig. 11.4 `Color` methods and color-related `Graphics` methods.



Common Programming Error 11.1

Spelling any **static Color** class constant with an initial capital letter is a syntax error.

Two **Color** constructors are shown in Fig. 11.4—one that takes three **int** arguments, and one that takes three **float** arguments, with each argument specifying the amount of red, green and blue, respectively. The **int** values must be between 0 and 255 and the **float** values must be between 0.0 and 1.0. The new **Color** object will have the specified amounts of red, green and blue. **Color** methods **getRed**, **getGreen** and **getBlue** return integer values from 0 to 255 representing the amount of red, green and blue, respectively. **Graphics** method **getColor** returns a **Color** object representing the current drawing color. **Graphics** method **setColor** sets the current drawing color.

The application of Fig. 11.5 demonstrates several methods from Fig. 11.4 by drawing filled rectangles and strings in several different colors.

```

1  // Fig. 11.5: ShowColors.java
2  // Demonstrating Colors.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class ShowColors extends JFrame {
12
13     // constructor sets window's title bar string and dimensions
14     public ShowColors()
15     {
16         super( "Using colors" );
17
18         setSize( 400, 130 );
19         setVisible( true );
20     }
21
22     // draw rectangles and Strings in different colors
23     public void paint( Graphics g )
24     {
25         // call superclass's paint method
26         super.paint( g );
27
28         // set new drawing color using integers
29         g.setColor( new Color( 255, 0, 0 ) );
30         g.fillRect( 25, 25, 100, 20 );
31         g.drawString( "Current RGB: " + g.getColor(), 130, 40 );
32
33         // set new drawing color using floats
34         g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
35         g.fillRect( 25, 50, 100, 20 );
36         g.drawString( "Current RGB: " + g.getColor(), 130, 65 );

```

Fig. 11.5 Demonstrating setting and getting a **Color** (part 1 of 2).

```

37
38     // set new drawing color using static Color objects
39     g.setColor( Color.blue );
40     g.fillRect( 25, 75, 100, 20 );
41     g.drawString( "Current RGB: " + g.getColor(), 130, 90 );
42
43     // display individual RGB values
44     Color color = Color.magenta;
45     g.setColor( color );
46     g.fillRect( 25, 100, 100, 20 );
47     g.drawString( "RGB values: " + color.getRed() + ", " +
48         color.getGreen() + ", " + color.getBlue(), 130, 115 );
49 }
50
51 // execute application
52 public static void main( String args[] )
53 {
54     ShowColors application = new ShowColors();
55
56     application.setDefaultCloseOperation(
57         JFrame.EXIT_ON_CLOSE );
58 }
59
60 } // end class ShowColors

```



Fig. 11.5 Demonstrating setting and getting a **Color** (part 2 of 2).

When the application begins execution, class **ShowColors**'s **paint** method (lines 23–49) is called to paint the window. Line 29 uses **Graphics** method **setColor** to set the current drawing color. Method **setColor** receives a **Color** object. The expression **new Color(255, 0, 0)** creates a new **Color** object that represents red (red value 255, and 0 for the green and blue values). Line 30 uses **Graphics** method **fillRect** to draw a filled rectangle in the current color. Method **fillRect** receives the same parameters as method **drawRect** (discussed in Chapter 3). Line 31 uses **Graphics** method **drawString** to draw a **String** in the current color. The expression **g.getColor()** retrieves the current color from the **Graphics** object. The returned **Color** is concatenated with string **"Current RGB: "**, resulting in an implicit call to class **Color**'s **toString** method. Notice that the **String** representation of the **Color** object contains the class name and package (**java.awt.Color**), and the red, green and blue values.

Lines 34–36 and lines 39–41 perform the same tasks again. Line 34 uses the **Color** constructor with three **float** arguments to create the color green (**0.0f** for red, **1.0f** for green and **0.0f** for blue). Note the syntax of the constants. The letter **f** appended to a floating-point constant indicates that the constant should be treated as type **float**. Normally, floating-point constants are treated as type **double**.

Line 39 sets the current drawing color to one of the predefined **Color** constants (**Color.blue**). Note that the new operator is not needed to create the constant. The **Color** constants are **static**, so they are defined when class **Color** is loaded into memory at execution time.

The statement at lines 47–48 demonstrates **Color** methods **getRed**, **getGreen** and **getBlue** on the predefined **Color.magenta** object.

Notice lines 56–57 in **main**. **JFrame** method **setDefaultCloseOperation** specifies the default action to take when the user clicks the close box on an application window. In this case, we specify **JFrame.EXIT_ON_CLOSE** to indicate that the program should terminate when the user clicks the close box. Other options are **DO_NOTHING_ON_CLOSE** (to ignore the window-closing event), **HIDE_ON_CLOSE** (to hide the window, such that it can be redisplayed later) and **DISPOSE_ON_CLOSE** (to dispose of the window, such that it cannot be redisplayed later). From this point forward, we implement our own **WindowListener** only if the program should perform additional tasks when the user clicks the window's close box. Otherwise, we use method **setDefaultCloseOperation** to specify that the program should terminate when the user clicks the close box.



Software Engineering Observation 11.2

*To change the color, you must create a new **Color** object (or use one of the predefined **Color** constants); there are no set methods in class **Color** to change the characteristics of the current color.*

One of the newer features of Java is the predefined GUI component **JColorChooser** (package **javax.swing**) for selecting colors. The application of Fig. 11.6 enables you to press a button to display a **JColorChooser** dialog. When you select a color and press the dialog's **OK** button, the background color of the application window changes colors.

```

1  // Fig. 11.6: ShowColors2.java
2  // Demonstrating JColorChooser.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class ShowColors2 extends JFrame {
12     private JButton changeColorButton;
13     private Color color = Color.lightGray;
14     private Container container;
15
16     // set up GUI
17     public ShowColors2()
18     {
19         super( "Using JColorChooser" );
20     }

```

Fig. 11.6 Demonstrating the **JColorChooser** dialog (part 1 of 3).

```

21     container = getContentPane();
22     container.setLayout( new FlowLayout() );
23
24     // set up changeColorButton and register its event handler
25     changeColorButton = new JButton( "Change Color" );
26
27     changeColorButton.addActionListener(
28
29         // anonymous inner class
30         new ActionListener() {
31
32             // display JColorChooser when user clicks button
33             public void actionPerformed( ActionEvent event )
34             {
35                 color = JColorChooser.showDialog(
36                     ShowColors2.this, "Choose a color", color );
37
38                 // set default color, if no color is returned
39                 if ( color == null )
40                     color = Color.lightGray;
41
42                 // change content pane's background color
43                 container.setBackground( color );
44             }
45         } // end anonymous inner class
46     ); // end call to addActionListener
47
48     container.add( changeColorButton );
49
50     setSize( 400, 130 );
51     setVisible( true );
52 }
53
54 // execute application
55 public static void main( String args[] )
56 {
57     ShowColors2 application = new ShowColors2();
58
59     application.setDefaultCloseOperation(
60         JFrame.EXIT_ON_CLOSE );
61 }
62
63 } // end class ShowColors2

```

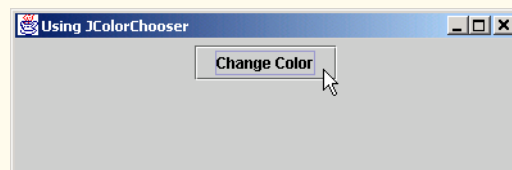


Fig. 11.6 Demonstrating the **JColorChooser** dialog (part 2 of 3).

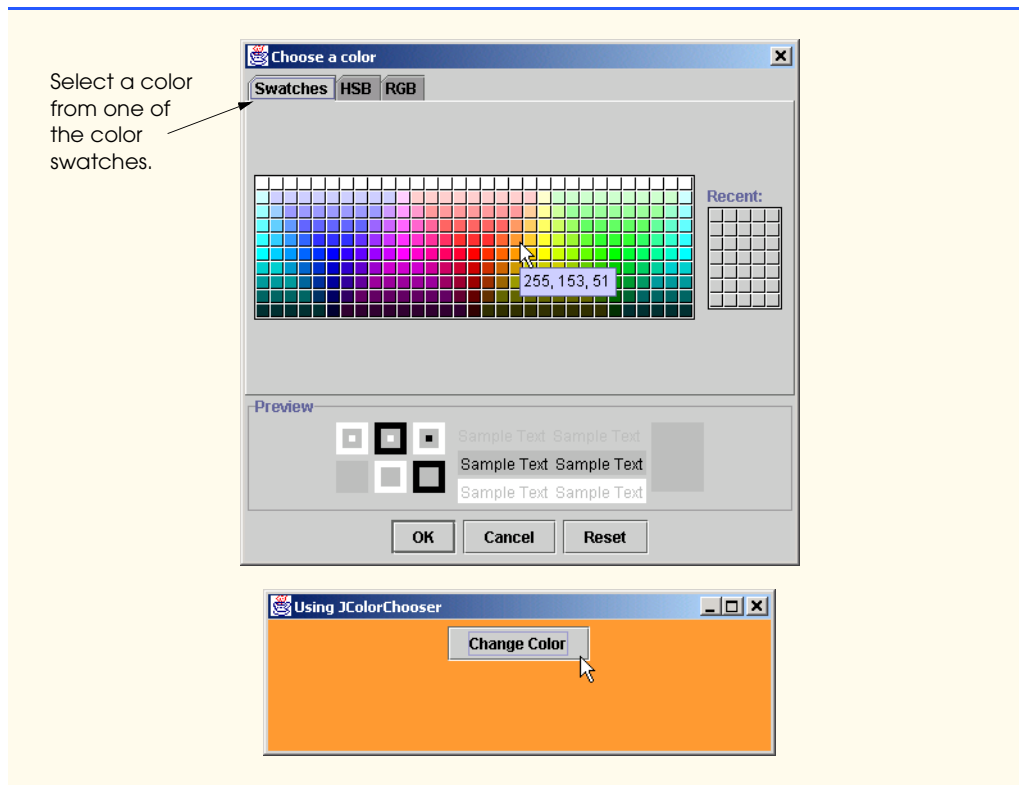


Fig. 11.6 Demonstrating the `JColorChooser` dialog (part 3 of 3).

Lines 35–36 (from method `actionPerformed` for `changeColor`) use `static` method `showDialog` of class `JColorChooser` to display the color chooser dialog. This method returns the selected `Color` object (`null`, if the user presses **Cancel** or closes the dialog without pressing **OK**).

Method `showDialog` takes three arguments—a reference to its parent `Component`, a `String` to display in the title bar of the dialog and the initial selected `Color` for the dialog. The parent component is the window from which the dialog is displayed. While the color chooser dialog is on the screen, the user cannot interact with the parent component. This type of dialog is called a *modal dialog* and is discussed in Chapter 13. Notice the special syntax `ShowColors2.this` used in the preceding statement. When using an inner class, you can access the outer class object's `this` reference by qualifying `this` with the name of the outer class and the dot (`.`) operator.

After the user selects a color, lines 39–40 determine whether `color` is `null`, and, if so, set `color` to the default `Color.lightGray`. Line 43 uses method `setBackground` to change the background color of the content pane (represented by `container` in this program). Method `setBackground` is one of the many `Component` methods that can be used on most GUI components.

The second screen capture of Fig. 11.6 demonstrates the default `JColorChooser` dialog that allows the user to select a color from a variety of *color swatches*. Notice that there are actually three tabs across the top of the dialog—**Swatches**, **HSB** and **RGB**.

These represent three different ways to select a color. The **HSB** tab allows you to select a color based on *hue*, *saturation* and *brightness*. The **RGB** tab allows you to select a color by using sliders to select the red, green and blue components of the color. The **HSB** and **RGB** tabs are shown in Fig. 11.7.

11.4 Font Control

This section introduces methods and constants for font control. Most font methods and font constants are part of class **Font**. Some methods of class **Font** and class **Graphics** are summarized in Fig. 11.8.

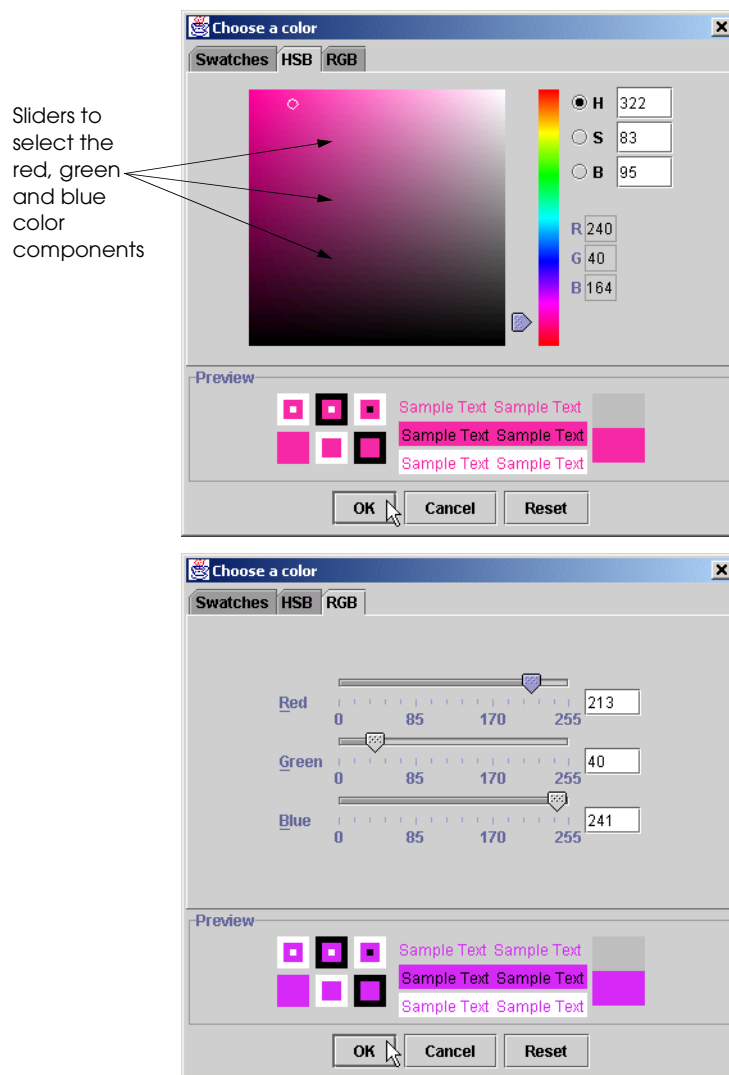


Fig. 11.7 The **HSB** and **RGB** tabs of the **JColorChooser** dialog.

Method or constant	Description
<code>public final static int PLAIN // Font class</code>	A constant representing a plain font style.
<code>public final static int BOLD // Font class</code>	A constant representing a bold font style.
<code>public final static int ITALIC // Font class</code>	A constant representing an italic font style.
<code>public Font(String name, int style, int size)</code>	Creates a Font object with the specified font, style and size.
<code>public int getStyle() // Font class</code>	Returns an integer value indicating the current font style.
<code>public int getSize() // Font class</code>	Returns an integer value indicating the current font size.
<code>public String getName() // Font class</code>	Returns the current font name as a string.
<code>public String getFamily() // Font class</code>	Returns the font's family name as a string.
<code>public boolean isPlain() // Font class</code>	Tests a font for a plain font style. Returns true if the font is plain.
<code>public boolean isBold() // Font class</code>	Tests a font for a bold font style. Returns true if the font is bold.
<code>public boolean isItalic() // Font class</code>	Tests a font for an italic font style. Returns true if the font is italic.
<code>public Font getFont() // Graphics class</code>	Returns a Font object reference representing the current font.
<code>public void setFont(Font f) // Graphics class</code>	Sets the current font to the font, style and size specified by the Font object reference f .

Fig. 11.8 **Font** methods, constants and font-related **Graphics** methods.

Class **Font**'s constructor takes three arguments—the *font name*, *font style* and *font size*. The font name is any font currently supported by the system where the program is running, such as standard Java fonts **Monospaced**, **SansSerif** and **Serif**. The font style is **Font.PLAIN**, **Font.ITALIC** or **Font.BOLD** (**static** constants of class **Font**). Font styles can be used in combination (e.g., **Font.ITALIC + Font.BOLD**). The font size is measured in points. A *point* is 1/72 of an inch. **Graphics** method **setFont** sets the current drawing font—the font in which text will be displayed—to its **Font** argument.

**Portability Tip 11.2**

*The number of fonts varies greatly across systems. The J2SDK guarantees that the fonts **Serif**, **Monospaced**, **SansSerif**, **Dialog** and **DialogInput** will be available.*

**Common Programming Error 11.2**

Specifying a font that is not available on a system is a logic error. Java will substitute that system's default font.

The program of Fig. 11.9 displays text in four different fonts, with each font in a different size. The program uses the **Font** constructor to initialize **Font** objects on lines 30, 35, 40 and 47 (each in a call to **Graphics** method **setFont** to change the drawing font). Each call to the **Font** constructor passes a font name (**Serif**, **Monospaced** or **SansSerif**) as a **String**, a font style (**Font.PLAIN**, **Font.ITALIC** or **Font.BOLD**) and a font size. Once **Graphics** method **setFont** is invoked, all text displayed following the call will appear in the new font until the font is changed. Note that line 35 changes the drawing color to red, so the next string displayed appears in red.

```

1  // Fig. 11.9: Fonts.java
2  // Using fonts
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class Fonts extends JFrame {
12
13     // set window's title bar and dimensions
14     public Fonts()
15     {
16         super( "Using fonts" );
17
18         setSize( 420, 125 );
19         setVisible( true );
20     }
21
22     // display Strings in different fonts and colors
23     public void paint( Graphics g )
24     {
25         // call superclass's paint method
26         super.paint( g );
27
28         // set current font to Serif (Times), bold, 12pt
29         // and draw a string
30         g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
31         g.drawString( "Serif 12 point bold.", 20, 50 );
32

```

Fig. 11.9 Using **Graphics** method **setFont** to change **FontS** (part 1 of 2).

```

33 // set current font to Monospaced (Courier),
34 // italic, 24pt and draw a string
35 g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
36 g.drawString( "Monospaced 24 point italic.", 20, 70 );
37
38 // set current font to SansSerif (Helvetica),
39 // plain, 14pt and draw a string
40 g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
41 g.drawString( "SansSerif 14 point plain.", 20, 90 );
42
43 // set current font to Serif (times), bold/italic,
44 // 18pt and draw a string
45 g.setColor( Color.red );
46 g.setFont(
47     new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
48 g.drawString( g.getFont().getName() + " " +
49     g.getFont().getSize() +
50     " point bold italic.", 20, 110 );
51 }
52
53 // execute application
54 public static void main( String args[] )
55 {
56     Fonts application = new Fonts();
57
58     application.setDefaultCloseOperation(
59         JFrame.EXIT_ON_CLOSE );
60 }
61
62 } // end class Fonts

```

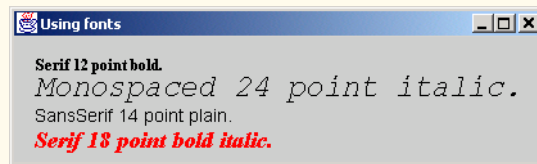


Fig. 11.9 Using **Graphics** method **setFont** to change **Fonts** (part 2 of 2).



Software Engineering Observation 11.3

To change the font, you must create a new **Font** object; there are no set methods in class **Font** to change the characteristics of the current font.

Often, it is necessary to get information about the current font, such as the font name, the font style and the font size. Several **Font** methods used to get font information are summarized in Fig. 11.8. Method **getStyle** returns an integer value representing the current style. The integer value returned is either **Font.PLAIN**, **Font.ITALIC**, **Font.BOLD** or any combination of **Font.PLAIN**, **Font.ITALIC** and **Font.BOLD**.

Method **getSize** returns the font size in points. Method **getName** returns the current font name as a **String**. Method **getFamily** returns the name of the font family to which the current font belongs. The name of the font family is platform specific.



Portability Tip 11.3

Java provides several standardized font names and maps these into system-specific font names for portability. This is transparent to the programmer.

Font methods are also available to test the style of the current font and are summarized in Fig. 11.8. The **isPlain** method returns **true** if the current font style is plain. The **isBold** method returns **true** if the current font style is bold. The **isItalic** method returns **true** if the current font style is italic.

Sometimes precise information about a font's metrics must be known—such as *height*, *descent* (the amount a character dips below the baseline), *ascent* (the amount a character rises above the baseline) and *leading* (the difference between the descent of one line of text and the ascent of the line of text below it—i.e., the interline spacing). Figure 11.10 illustrates some of the common *font metrics*. Note that the coordinate passed to **drawString** corresponds to the lower-left corner of the baseline of the font.

Class **FontMetrics** defines several methods for obtaining font metrics. These methods and **Graphics** method **getFontMetrics** are summarized in Fig. 11.11.

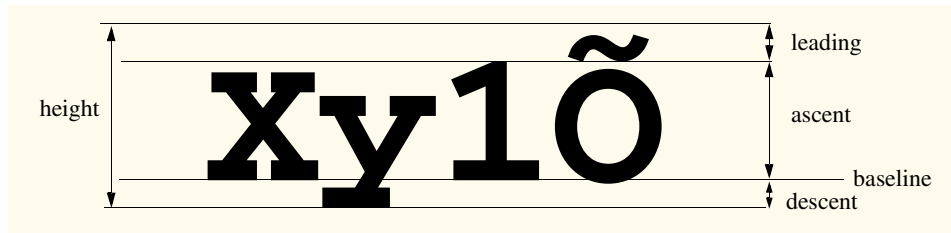


Fig. 11.10 Font metrics.

Method	Description
<code>public int getAscent()</code>	<code>// FontMetrics class</code> Returns a value representing the ascent of a font in points.
<code>public int getDescent()</code>	<code>// FontMetrics class</code> Returns a value representing the descent of a font in points.
<code>public int getLeading()</code>	<code>// FontMetrics class</code> Returns a value representing the leading of a font in points.
<code>public int getHeight()</code>	<code>// FontMetrics class</code> Returns a value representing the height of a font in points.
<code>public FontMetrics getFontMetrics()</code>	<code>// Graphics class</code> Returns the FontMetrics object for the current drawing Font .
<code>public FontMetrics getFontMetrics(Font f)</code>	<code>// Graphics class</code> Returns the FontMetrics object for the specified Font argument.

Fig. 11.11 **FontMetrics** and **Graphics** methods for obtaining font metrics.

The program of Fig. 11.12 uses the methods of Fig. 11.11 to obtain font metric information for two fonts.

```

1  // Fig. 11.12: Metrics.java
2  // Demonstrating methods of class FontMetrics and
3  // class Graphics useful for obtaining font metrics.
4
5  // Java core packages
6  import java.awt.*;
7  import java.awt.event.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class Metrics extends JFrame {
13
14     // set window's title bar String and dimensions
15     public Metrics()
16     {
17         super( "Demonstrating FontMetrics" );
18
19         setSize( 510, 210 );
20         setVisible( true );
21     }
22
23     // display font metrics
24     public void paint( Graphics g )
25     {
26         // call superclass's paint method
27         super.paint( g );
28
29         g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
30         FontMetrics metrics = g.getFontMetrics();
31         g.drawString( "Current font: " + g.getFont(), 10, 40 );
32         g.drawString( "Ascent: " + metrics.getAscent(), 10, 55 );
33         g.drawString( "Descent: " + metrics.getDescent(), 10, 70 );
34         g.drawString( "Height: " + metrics.getHeight(), 10, 85 );
35         g.drawString( "Leading: " + metrics.getLeading(), 10, 100);
36
37         Font font = new Font( "Serif", Font.ITALIC, 14 );
38         metrics = g.getFontMetrics( font );
39         g.setFont( font );
40         g.drawString( "Current font: " + font, 10, 130 );
41         g.drawString( "Ascent: " + metrics.getAscent(), 10, 145 );
42         g.drawString( "Descent: " + metrics.getDescent(), 10, 160);
43         g.drawString( "Height: " + metrics.getHeight(), 10, 175 );
44         g.drawString( "Leading: " + metrics.getLeading(), 10, 190);
45     }
46
47     // execute application
48     public static void main( String args[] )
49     {
50         Metrics application = new Metrics();

```

Fig. 11.12 Obtaining font metric information (part 1 of 2).

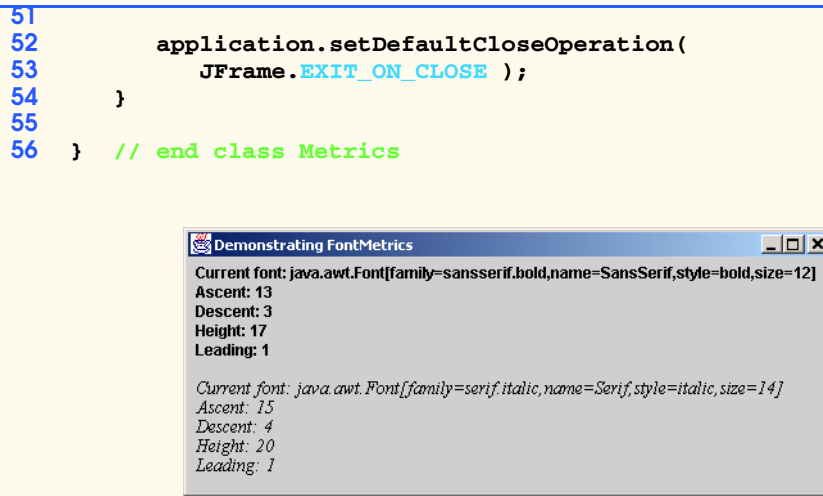


Fig. 11.12 Obtaining font metric information (part 2 of 2).

Line 29 creates and sets the current drawing font to a **SansSerif**, bold, 12-point font. Line 30 uses **Graphics** method **getFontMetrics** to obtain the **FontMetrics** object for the current font. Line 31 uses an implicit call to class **Font**'s **toString** method to output the string representation of the font. Lines 32–35 use **FontMetric** methods to obtain the ascent, descent, height and leading for the font.

Line 37 creates a new **Serif**, italic, 14-point font. Line 38 uses a second version of **Graphics** method **getFontMetrics**, which receives a **Font** argument and returns a corresponding **FontMetrics** object. Lines 41–44 obtain the ascent, descent, height and leading for the font. Notice that the font metrics are slightly different for the two fonts.

11.5 Drawing Lines, Rectangles and Ovals

This section presents a variety of **Graphics** methods for drawing lines, rectangles and ovals. The methods and their parameters are summarized in Fig. 11.13. For each drawing method that requires a **width** and **height** parameter, the **width** and **height** must be nonnegative values. Otherwise, the shape will not display..

Method	Description
<code>public void drawLine(int x1, int y1, int x2, int y2)</code>	Draws a line between the point (x1, y1) and the point (x2, y2).
<code>public void drawRect(int x, int y, int width, int height)</code>	Draws a rectangle of the specified width and height . The top-left corner of the rectangle has the coordinates (x, y).

Fig. 11.13 **Graphics** methods that draw lines, rectangles and ovals (part 1 of 2).

Method	Description
<code>public void fillRect(int x, int y, int width, int height)</code>	Draws a solid rectangle with the specified width and height . The top-left corner of the rectangle has the coordinate (x , y).
<code>public void clearRect(int x, int y, int width, int height)</code>	Draws a solid rectangle with the specified width and height in the current background color. The top-left corner of the rectangle has the coordinate (x , y).
<code>public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</code>	Draws a rectangle with rounded corners in the current color with the specified width and height . The arcWidth and arcHeight determine the rounding of the corners (see Fig. 11.15).
<code>public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</code>	Draws a solid rectangle with rounded corners in the current color with the specified width and height . The arcWidth and arcHeight determine the rounding of the corners (see Fig. 11.15).
<code>public void draw3DRect(int x, int y, int width, int height, boolean b)</code>	Draws a three-dimensional rectangle in the current color with the specified width and height . The top-left corner of the rectangle has the coordinates (x , y). The rectangle appears raised when b is true and is lowered when b is false.
<code>public void fill3DRect(int x, int y, int width, int height, boolean b)</code>	Draws a filled three-dimensional rectangle in the current color with the specified width and height . The top-left corner of the rectangle has the coordinates (x , y). The rectangle appears raised when b is true and is lowered when b is false.
<code>public void drawOval(int x, int y, int width, int height)</code>	Draws an oval in the current color with the specified width and height . The bounding rectangle's top-left corner is at the coordinates (x , y). The oval touches all four sides of the bounding rectangle at the center of each side (see Fig. 11.16).
<code>public void fillOval(int x, int y, int width, int height)</code>	Draws a filled oval in the current color with the specified width and height . The bounding rectangle's top-left corner is at the coordinates (x , y). The oval touches all four sides of the bounding rectangle at the center of each side (see Fig. 11.16).

Fig. 11.13 Graphics methods that draw lines, rectangles and ovals (part 2 of 2).

The application of Fig. 11.14 demonstrates drawing a variety of lines, rectangles, 3D rectangles, rounded rectangles and ovals.

```
1 // Fig. 11.14: LinesRectsOvals.java
2 // Drawing lines, rectangles and ovals
3
4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Java extension packages
9 import javax.swing.*;
10
11 public class LinesRectsOvals extends JFrame {
12
13     // set window's title bar String and dimensions
14     public LinesRectsOvals()
15     {
16         super( "Drawing lines, rectangles and ovals" );
17
18         setSize( 400, 165 );
19         setVisible( true );
20     }
21
22     // display various lines, rectangles and ovals
23     public void paint( Graphics g )
24     {
25         // call superclass's paint method
26         super.paint( g );
27
28         g.setColor( Color.red );
29         g.drawLine( 5, 30, 350, 30 );
30
31         g.setColor( Color.blue );
32         g.drawRect( 5, 40, 90, 55 );
33         g.fillRect( 100, 40, 90, 55 );
34
35         g.setColor( Color.cyan );
36         g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
37         g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
38
39         g.setColor( Color.yellow );
40         g.draw3DRect( 5, 100, 90, 55, true );
41         g.fill3DRect( 100, 100, 90, 55, false );
42
43         g.setColor( Color.magenta );
44         g.drawOval( 195, 100, 90, 55 );
45         g.fillOval( 290, 100, 90, 55 );
46     }
47 }
```

Fig. 11.14 Demonstrating **Graphics** method **drawLine** (part 1 of 2).

```

48 // execute application
49 public static void main( String args[] )
50 {
51     LinesRectsOvals application = new LinesRectsOvals();
52
53     application.setDefaultCloseOperation(
54         JFrame.EXIT_ON_CLOSE );
55 }
56
57 } // end class LinesRectsOvals

```

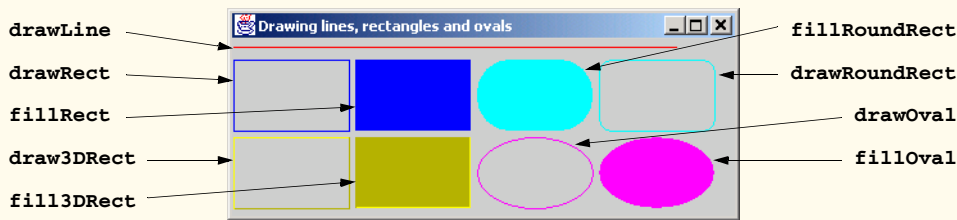


Fig. 11.14 Demonstrating **Graphics** method **drawLine** (part 2 of 2).

Methods **fillRoundRect** (line 36) and **drawRoundRect** (line 37) draw rectangles with rounded corners. Their first two arguments specify the coordinates of the upper-left corner of the *bounding rectangle*—the area in which the rounded rectangle will be drawn. Note that the upper-left corner coordinates are not the edge of the rounded rectangle, but the coordinates where the edge would be if the rectangle had square corners. The third and fourth arguments specify the **width** and **height** of the rectangle. Their last two arguments—**arcWidth** and **arcHeight**—determine the horizontal and vertical diameters of the arcs used to represent the corners.

Methods **draw3DRect** (line 40) and **fill3DRect** (line 41) take the same arguments. The first two arguments specify the top-left corner of the rectangle. The next two arguments specify the **width** and **height** of the rectangle, respectively. The last argument determines whether the rectangle is *raised* (**true**) or *lowered* (**false**). The three-dimensional effect of **draw3DRect** appears as two edges of the rectangle in the original color and two edges in a slightly darker color. The three-dimensional effect of **fill3DRect** appears as two edges of the rectangle in the original drawing color and the fill and other two edges in a slightly darker color. Raised rectangles have the original drawing color edges at the top and left of the rectangle. Lowered rectangles have the original drawing color edges at the bottom and right of the rectangle. The three-dimensional effect is difficult to see in some colors.

Figure 11.15 labels the arc width, arc height, width and height of a rounded rectangle. Using the same value for **arcWidth** and **arcHeight** produces a quarter circle at each corner. When **width**, **height**, **arcWidth** and **arcHeight** have the same values, the result is a circle. If the values for **width** and **height** are the same and the values of **arcWidth** and **arcHeight** are 0, the result is a square.

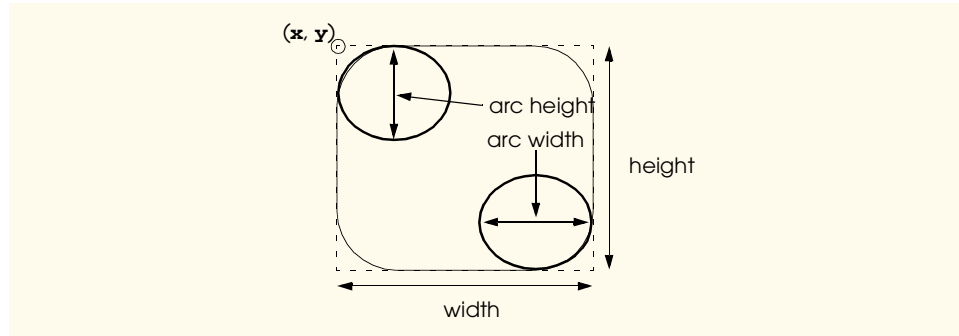


Fig. 11.15 The arc width and arc height for rounded rectangles.

The **drawOval** and **fillOval** methods take the same four arguments. The first two arguments specify the top-left coordinate of the bounding rectangle that contains the oval. The last two arguments specify the **width** and **height** of the bounding rectangle, respectively. Figure 11.16 shows an oval bounded by a rectangle. Note that the oval touches the center of all four sides of the bounding rectangle (the bounding rectangle is not displayed on the screen).

11.6 Drawing Arcs

An *arc* is a portion of an oval. Arc angles are measured in degrees. Arcs *sweep* from a *starting angle* the number of degrees specified by their *arc angle*. The starting angle indicates in degrees where the arc begins. The arc angle specifies the total number of degrees through which the arc sweeps. Figure 11.17 illustrates two arcs. The left set of axes shows an arc sweeping from zero degrees to approximately 110 degrees. Arcs that sweep in a counter-clockwise direction are measured in *positive degrees*. The right set of axes shows an arc sweeping from zero degrees to approximately -110 degrees. Arcs that sweep in a clockwise direction are measured in *negative degrees*. Notice the dashed boxes around the arcs in Fig. 11.17. When drawing an arc, we specify a bounding rectangle for an oval. The arc will sweep along part of the oval. The **Graphics** methods **drawArc** and **fillArc** for drawing arcs are summarized in Fig. 11.18.

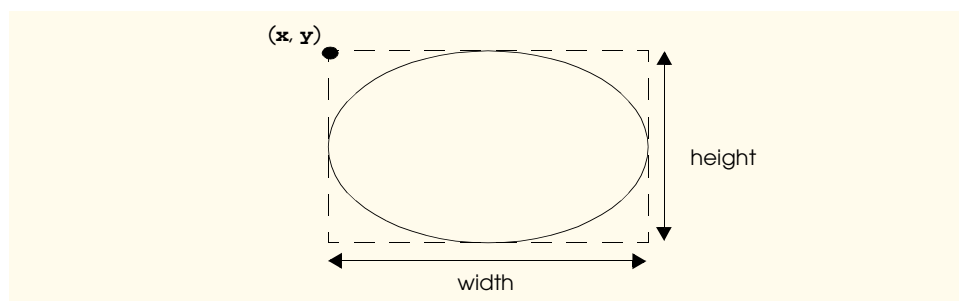


Fig. 11.16 An oval bounded by a rectangle.

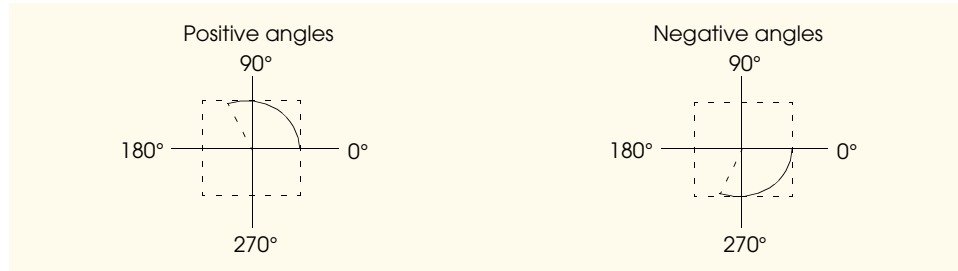


Fig. 11.17 Positive and negative arc angles.

Method	Description
<pre>public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)</pre>	Draws an arc relative to the bounding rectangle's top-left coordinates (x, y) with the specified width and height. The arc segment is drawn starting at startAngle and sweeps arcAngle degrees.
<pre>public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)</pre>	Draws a solid arc (i.e., a sector) relative to the bounding rectangle's top-left coordinates (x, y) with the specified width and height. The arc segment is drawn starting at startAngle and sweeps arcAngle degrees.

Fig. 11.18 **Graphics** methods for drawing arcs.

The program of Fig. 11.19 demonstrates the arc methods of Fig. 11.18. The program draws six arcs (three unfilled and three filled). To illustrate the bounding rectangle that helps determine where the arc appears, the first three arcs are displayed inside a yellow rectangle that has the same **x**, **y**, **width** and **height** arguments as the arcs.

```

1  // Fig. 11.19: DrawArcs.java
2  // Drawing arcs
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class DrawArcs extends JFrame {
12
```

Fig. 11.19 Demonstrating **drawArc** and **fillArc** (part 1 of 3).

```

13 // set window's title bar String and dimensions
14 public DrawArcs()
15 {
16     super( "Drawing Arcs" );
17
18     setSize( 300, 170 );
19     setVisible( true );
20 }
21
22 // draw rectangles and arcs
23 public void paint( Graphics g )
24 {
25     // call superclass's paint method
26     super.paint( g );
27
28     // start at 0 and sweep 360 degrees
29     g.setColor( Color.yellow );
30     g.drawRect( 15, 35, 80, 80 );
31     g.setColor( Color.black );
32     g.drawArc( 15, 35, 80, 80, 0, 360 );
33
34     // start at 0 and sweep 110 degrees
35     g.setColor( Color.yellow );
36     g.drawRect( 100, 35, 80, 80 );
37     g.setColor( Color.black );
38     g.drawArc( 100, 35, 80, 80, 0, 110 );
39
40     // start at 0 and sweep -270 degrees
41     g.setColor( Color.yellow );
42     g.drawRect( 185, 35, 80, 80 );
43     g.setColor( Color.black );
44     g.drawArc( 185, 35, 80, 80, 0, -270 );
45
46     // start at 0 and sweep 360 degrees
47     g.fillArc( 15, 120, 80, 40, 0, 360 );
48
49     // start at 270 and sweep -90 degrees
50     g.fillArc( 100, 120, 80, 40, 270, -90 );
51
52     // start at 0 and sweep -270 degrees
53     g.fillArc( 185, 120, 80, 40, 0, -270 );
54 }
55
56 // execute application
57 public static void main( String args[] )
58 {
59     DrawArcs application = new DrawArcs();
60
61     application.setDefaultCloseOperation(
62         JFrame.EXIT_ON_CLOSE );
63 }
64
65 } // end class DrawArcs

```

Fig. 11.19 Demonstrating **drawArc** and **fillArc** (part 2 of 3).

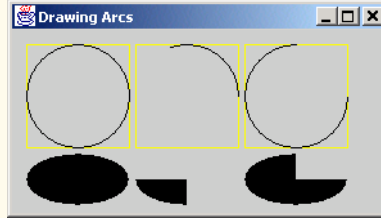


Fig. 11.19 Demonstrating **drawArc** and **fillArc** (part 3 of 3).

11.7 Drawing Polygons and Polylines

Polygons are multisided shapes. *Polylines* are a series of connected points. Graphics methods for drawing polygons and polylines are discussed in Fig. 11.20. Note that some methods require a **Polygon** object (package **java.awt**). Class **Polygon**'s constructors are also described in Fig. 11.20.

Method	Description
public void drawPolygon(int xPoints[], int yPoints[], int points)	Draws a polygon. The <i>x</i> -coordinate of each point is specified in the xPoints array and the <i>y</i> -coordinate of each point is specified in the yPoints array. The last argument specifies the number of points . This method draws a closed polygon—even if the last point is different from the first point.
public void drawPolyline(int xPoints[], int yPoints[], int points)	Draws a series of connected lines. The <i>x</i> -coordinate of each point is specified in the xPoints array and the <i>y</i> -coordinate of each point is specified in the yPoints array. The last argument specifies the number of points . If the last point is different from the first point, the polyline is not closed.
public void drawPolygon(Polygon p)	Draws the specified closed polygon.
public void fillPolygon(int xPoints[], int yPoints[], int points)	Draws a solid polygon. The <i>x</i> -coordinate of each point is specified in the xPoints array and the <i>y</i> -coordinate of each point is specified in the yPoints array. The last argument specifies the number of points . This method draws a closed polygon—even if the last point is different from the first point.
public void fillPolygon(Polygon p)	Draws the specified solid polygon. The polygon is closed.

Fig. 11.20 **Graphics** methods for drawing polygons and class **Polygon** constructors (part 1 of 2).

Method	Description
<code>public Polygon()</code>	<code>// Polygon class</code> Constructs a new polygon object. The polygon does not contain any points.
<code>public Polygon(int xValues[], int yValues[], int numberOfPoints)</code>	<code>// Polygon class</code> Constructs a new polygon object. The polygon has numberOfPoints sides, with each point consisting of an <i>x</i> -coordinate from xValues and a <i>y</i> -coordinate from yValues .

Fig. 11.20 **Graphics** methods for drawing polygons and class **Polygon** constructors (part 2 of 2).

The program of Fig. 11.21 draws polygons and polylines, using the methods and constructors in Fig. 11.20.

```

1  // Fig. 11.21: DrawPolygons.java
2  // Drawing polygons
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class DrawPolygons extends JFrame {
12
13     // set window's title bar String and dimensions
14     public DrawPolygons()
15     {
16         super( "Drawing Polygons" );
17
18         setSize( 275, 230 );
19         setVisible( true );
20     }
21
22     // draw polygons and polylines
23     public void paint( Graphics g )
24     {
25         // call superclass's paint method
26         super.paint( g );
27
28         int xValues[] = { 20, 40, 50, 30, 20, 15 };
29         int yValues[] = { 50, 50, 60, 80, 80, 60 };
30         Polygon polygon1 = new Polygon( xValues, yValues, 6 );
31
32         g.drawPolygon( polygon1 );

```

Fig. 11.21 Demonstrating **drawPolygon** and **fillPolygon**.

```

33
34     int xValues2[] = { 70, 90, 100, 80, 70, 65, 60 };
35     int yValues2[] = { 100, 100, 110, 110, 130, 110, 90 };
36
37     g.drawPolyline( xValues2, yValues2, 7 );
38
39     int xValues3[] = { 120, 140, 150, 190 };
40     int yValues3[] = { 40, 70, 80, 60 };
41
42     g.fillPolygon( xValues3, yValues3, 4 );
43
44     Polygon polygon2 = new Polygon();
45     polygon2.addPoint( 165, 135 );
46     polygon2.addPoint( 175, 150 );
47     polygon2.addPoint( 270, 200 );
48     polygon2.addPoint( 200, 220 );
49     polygon2.addPoint( 130, 180 );
50
51     g.fillPolygon( polygon2 );
52 }
53
54 // execute application
55 public static void main( String args[] )
56 {
57     DrawPolygons application = new DrawPolygons();
58
59     application.setDefaultCloseOperation(
60         JFrame.EXIT_ON_CLOSE );
61 }
62
63 } // end class DrawPloygons

```

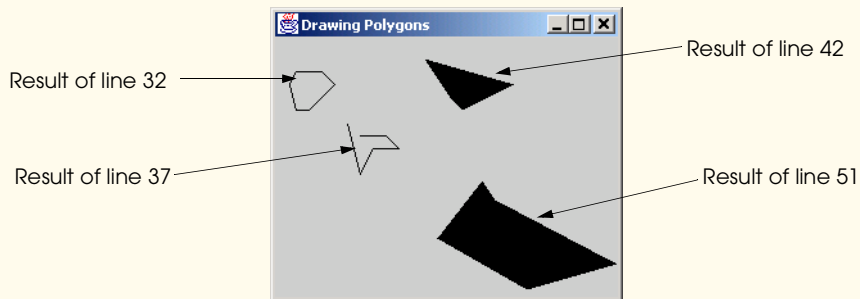


Fig. 11.21 Demonstrating `drawPolygon` and `fillPolygon`.

Lines 28–29 create two `int` arrays and use them to specify the points for **Polygon** `polygon1`. The **Polygon** constructor call at line 30 receives array `xValues`, which contains the *x*-coordinate of each point, array `yValues`, which contains the *y*-coordinate of each point, and 6 (the number of points in the polygon). Line 32 displays `polygon1` by passing it as an argument to **Graphics** method `drawPolygon`.

Lines 34–35 create two `int` arrays and use them to specify the points for a series of connected lines. Array `xValues2` contains the *x*-coordinate of each point and array

yValues2 contains the y-coordinate of each point. Line 37 uses **Graphics** method **drawPolyline** to display the series of connected lines specified with the arguments **xValues2**, **yValues2** and **7** (the number of points).

Lines 39–40 create two **int** arrays and use them to specify the points of a polygon. Array **xValues3** contains the x-coordinate of each point and array **yValues3** contains the y-coordinate of each point. Line 42 displays a polygon by passing to **Graphics** method **fillPolygon** the two arrays (**xValues3** and **yValues3**) and the number of points to draw (**4**).



Common Programming Error 11.3

An **ArrayIndexOutOfBoundsException** is thrown if the number of points specified in the third argument to method **drawPolygon** or method **fillPolygon** is greater than the number of elements in the arrays of coordinates that define the polygon to display.

Line 44 creates **Polygon polygon2** with no points. Lines 45–49 use **Polygon** method **addPoint** to add pairs of x- and y-coordinates to the **Polygon**. Line 51 displays **Polygon polygon2** by passing it to **Graphics** method **fillPolygon**.

11.8 The Java2D API

The new *Java2D API* provides advanced two-dimensional graphics capabilities for programmers who require detailed and complex graphical manipulations. The API includes features for processing line art, text and images in packages **java.awt**, **java.awt.image**, **java.awt.color**, **java.awt.font**, **java.awt.geom**, **java.awt.print** and **java.awt.image.renderable**. The capabilities of the API are far too broad to cover in this textbook. For an overview of the capabilities, see the Java2D demo (demonstrated in Chapter 3). In this section, we present an overview of several Java2D capabilities.

Drawing with the Java2D API is accomplished with an instance of class **Graphics2D** (package **java.awt**). Class **Graphics2D** is a subclass of class **Graphics**, so it has all the graphics capabilities demonstrated earlier in this chapter. In fact, the actual object we have used to draw in every **paint** method is a **Graphics2D** object that is passed to method **paint** and accessed via the superclass **Graphics** reference **g**. To access the **Graphics2D** capabilities, we must downcast the **Graphics** reference passed to **paint** to a **Graphics2D** reference with a statement such as

```
Graphics2D g2d = ( Graphics2D ) g;
```

The programs of the next several sections use this technique.

11.9 Java2D Shapes

Next, we present several Java2D shapes from package **java.awt.geom**, including **Ellipse2D.Double**, **Rectangle2D.Double**, **RoundRectangle2D.Double**, **Arc2D.Double** and **Line2D.Double**. Note the syntax of each class name. Each of these classes represents a shape with dimensions specified as double-precision floating-point values. There is a separate version of each represented with single-precision floating-point values (such as **Ellipse2D.Float**). In each case, **Double** is a **static** inner class of the class to the left of the dot operator (e.g., **Ellipse2D**). To use the **static** inner class, we simply qualify its name with the outer class name.

The program of Fig. 11.22 demonstrates several Java2D shapes and drawing characteristics, such as thickening lines, filling shapes with patterns and drawing dashed lines. These are just a few of the many capabilities provided by Java2D.

```

1  // Fig. 11.22: Shapes.java
2  // Demonstrating some Java2D shapes
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7  import java.awt.geom.*;
8  import java.awt.image.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class Shapes extends JFrame {
14
15     // set window's title bar String and dimensions
16     public Shapes()
17     {
18         super( "Drawing 2D shapes" );
19
20         setSize( 425, 160 );
21         setVisible( true );
22     }
23
24     // draw shapes with Java2D API
25     public void paint( Graphics g )
26     {
27         // call superclass's paint method
28         super.paint( g );
29
30         // create 2D by casting g to Graphics2D
31         Graphics2D g2d = ( Graphics2D ) g;
32
33         // draw 2D ellipse filled with a blue-yellow gradient
34         g2d.setPaint( new GradientPaint( 5, 30, Color.blue, 35,
35             100, Color.yellow, true ) );
36         g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
37
38         // draw 2D rectangle in red
39         g2d.setPaint( Color.red );
40         g2d.setStroke( new BasicStroke( 10.0f ) );
41         g2d.draw( new Rectangle2D.Double( 80, 30, 65, 100 ) );
42
43         // draw 2D rounded rectangle with a buffered background
44         BufferedImage buffImage = new BufferedImage(
45             10, 10, BufferedImage.TYPE_INT_RGB );
46
47         Graphics2D gg = buffImage.createGraphics();
48         gg.setColor( Color.yellow ); // draw in yellow

```

Fig. 11.22 Demonstrating some Java2D shapes.

```

49     gg.fillRect( 0, 0, 10, 10 ); // draw a filled rectangle
50     gg.setColor( Color.black ); // draw in black
51     gg.drawRect( 1, 1, 6, 6 ); // draw a rectangle
52     gg.setColor( Color.blue ); // draw in blue
53     gg.fillRect( 1, 1, 3, 3 ); // draw a filled rectangle
54     gg.setColor( Color.red ); // draw in red
55     gg.fillRect( 4, 4, 3, 3 ); // draw a filled rectangle
56
57     // paint buffImage onto the JFrame
58     g2d.setPaint( new TexturePaint(
59         buffImage, new Rectangle( 10, 10 ) ) );
60     g2d.fill( new RoundRectangle2D.Double(
61         155, 30, 75, 100, 50, 50 ) );
62
63     // draw 2D pie-shaped arc in white
64     g2d.setPaint( Color.white );
65     g2d.setStroke( new BasicStroke( 6.0f ) );
66     g2d.draw( new Arc2D.Double(
67         240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
68
69     // draw 2D lines in green and yellow
70     g2d.setPaint( Color.green );
71     g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
72
73     float dashes[] = { 10 };
74
75     g2d.setPaint( Color.yellow );
76     g2d.setStroke( new BasicStroke( 4, BasicStroke.CAP_ROUND,
77         BasicStroke.JOIN_ROUND, 10, dashes, 0 ) );
78     g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
79 }
80
81 // execute application
82 public static void main( String args[] )
83 {
84     Shapes application = new Shapes();
85
86     application.setDefaultCloseOperation(
87         JFrame.EXIT_ON_CLOSE );
88 }
89
90 } // end class Shapes

```

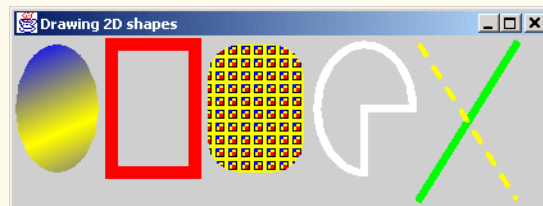


Fig. 11.22 Demonstrating some Java2D shapes.

Line 31 casts the **Graphics** reference received by **paint** to a **Graphics2D** reference and assigns it to **g2d** to allow access to the **Java2D** features.

The first shape we draw is an oval filled with gradually changing colors. Lines 34–35 invoke **Graphics2D** method **setPaint** to set the **Paint** object that determines the color for the shape to display. A **Paint** object is an object of any class that implements interface **java.awt.Paint**. The **Paint** object can be something as simple as one of the predefined **Color** objects introduced in Section 11.3 (class **Color** implements **Paint**), or the **Paint** object can be an instance of the Java2D API's **GradientPaint**, **SystemColor** or **TexturePaint** classes. In this case, we use a **GradientPaint** object.

Class **GradientPaint** helps draw a shape in a gradually changing colors—called a *gradient*. The **GradientPaint** constructor used here requires seven arguments. The first two arguments specify the starting coordinate for the gradient. The third argument specifies the starting **Color** for the gradient. The fourth and fifth arguments specify the ending coordinate for the gradient. The sixth argument specifies the ending **Color** for the gradient. The last argument specifies if the gradient is cyclic (**true**) or acyclic (**false**). The two coordinates determine the direction of the gradient. Because the second coordinate (35, 100) is down and to the right of the first coordinate (5, 30), the gradient goes down and to the right at an angle. Because this gradient is cyclic (**true**), the color starts with blue, gradually becomes yellow, then gradually returns to blue. If the gradient is acyclic, the color transitions from the first color specified (e.g., blue) to the second color (e.g., yellow).

Line 36 uses **Graphics2D** method **fill** to draw a filled **Shape** object. The **Shape** object is an instance of any class that implements interface **Shape** (package **java.awt**)—in this case, an instance of class **Ellipse2D.Double**. The **Ellipse2D.Double** constructor receives four arguments specifying the bounding rectangle for the ellipse to display.

Next we draw a red rectangle with a thick border. Line 39 uses **setPaint** to set the **Paint** object to **Color.red**. Line 40 uses **Graphics2D** method **setStroke** to set the characteristics of the rectangle's border (or the lines for any other shape). Method **setStroke** requires a **Stroke** object as its argument. The **Stroke** object is an instance of any class that implements interface **Stroke** (package **java.awt**)—in this case, an instance of class **BasicStroke**. Class **BasicStroke** provides a variety of constructors to specify the width of the line, how the line ends (called the *end caps*), how lines join together (called *line joins*) and the dash attributes of the line (if it is a dashed line). The constructor here specifies that the line should be 10 pixels wide.

Line 41 uses **Graphics2D** method **draw** to draw a **Shape** object—in this case, an instance of class **Rectangle2D.Double**. The **Rectangle2D.Double** constructor receives four arguments specifying the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height of the rectangle.

Next we draw a rounded rectangle filled with a pattern created in a **BufferedImage** (package **java.awt.image**) object. Lines 44–45 create the **BufferedImage** object. Class **BufferedImage** can be used to produce images in color and gray scale. This particular **BufferedImage** is 10 pixels wide and 10 pixels tall. The third constructor argument **BufferedImage.TYPE_INT_RGB** indicates that the image is stored in color using the RGB color scheme.

To create the fill pattern for the rounded rectangle, we must first draw into the **BufferedImage**. Line 47 creates a **Graphics2D** object that can be used to draw into the **BufferedImage**. Lines 48–55 use methods **setColor**, **fillRect** and **drawRect** (discussed earlier in this chapter) to create the pattern.

Lines 58–59 set the **Paint** object to a new **TexturePaint** (package **java.awt**) object. A **TexturePaint** object uses the image stored in its associated **BufferedImage** as the fill texture for a filled-in shape. The second argument specifies the **Rectangle** area from the **BufferedImage** that will be replicated through the texture. In this case, the **Rectangle** is the same size as the **BufferedImage**. However, a smaller portion of the **BufferedImage** can be used.

Lines 60–61 use **Graphics2D** method **fill** to draw a filled **Shape** object—in this case, an instance of class **RoundRectangle2D.Double**. The constructor for class **RoundRectangle2D.Double** receives six arguments specifying the rectangle dimensions and the arc width and arc height used to determine the rounding of the corners.

Next we draw a pie-shaped arc with a thick white line. Line 64 sets the **Paint** object to **Color.white**. Line 65 sets the **Stroke** object to a new **BasicStroke** for a line 6 pixels wide. Lines 66–67 use **Graphics2D** method **draw** to draw a **Shape** object—in this case, an **Arc2D.Double**. The **Arc2D.Double** constructor's first four arguments specifying the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height of the bounding rectangle for the arc. The fifth argument specifies the start angle. The sixth argument specifies the arc angle. The last argument specifies how the arc is closed. Constant **Arc2D.PIE** indicates that the arc is closed by drawing two lines. One line from the arc's starting point to the center of the bounding rectangle and one line from the center of the bounding rectangle to the ending point. Class **Arc2D** provides two other **static** constants for specifying how the arc is closed. Constant **Arc2D.CHORD** draws a line from the starting point to the ending point. Constant **Arc2D.OPEN** specifies that the arc is not closed.

Finally, we draw two lines using **Line2D** objects—one solid and one dashed. Line 70 sets the **Paint** object to **Color.green**. Line 71 uses **Graphics2D** method **draw** to draw a **Shape** object—in this case, an instance of class **Line2D.Double**. The **Line2D.Double** constructor's arguments specify starting coordinates and ending coordinates of the line.

Line 73 defines a one-element **float** array containing the value 10. This array will be used to describe the dashes in the dashed line. In this case, each dash will be 10 pixels long. To create dashes of different lengths in a pattern, simply provide the lengths of each dash as an element in the array. Line 75 sets the **Paint** object to **Color.yellow**. Lines 76–77 set the **Stroke** object to a new **BasicStroke**. The line will be 4 pixels wide and will have rounded ends (**BasicStroke.CAP_ROUND**). If lines join together (as in a rectangle at the corners), the joining of the lines will be rounded (**BasicStroke.JOIN_ROUND**). The **dashes** argument specifies the dash lengths for the line. The last argument indicates the starting subscript in the **dashes** array for the first dash in the pattern. Line 78 then draws a line with the current **Stroke**.

Next we present a general path—a shape constructed from straight lines and complex curves. A general path is represented with an object of class **GeneralPath** (package **java.awt.geom**). The program of Fig. 11.23 demonstrates drawing a general path in the shape of a five-pointed star.

```
1  // Fig. 11.23: Shapes2.java
2  // Demonstrating a general path
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7  import java.awt.geom.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class Shapes2 extends JFrame {
13
14     // set window's title bar String, background color
15     // and dimensions
16     public Shapes2()
17     {
18         super( "Drawing 2D Shapes" );
19
20         getContentPane().setBackground( Color.yellow );
21         setSize( 400, 400 );
22         setVisible( true );
23     }
24
25     // draw general paths
26     public void paint( Graphics g )
27     {
28         // call superclass's paint method
29         super.paint( g );
30
31         int xPoints[] =
32             { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
33         int yPoints[] =
34             { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
35
36         Graphics2D g2d = ( Graphics2D ) g;
37
38         // create a star from a series of points
39         GeneralPath star = new GeneralPath();
40
41         // set the initial coordinate of the General Path
42         star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );
43
44         // create the star--this does not draw the star
45         for ( int count = 1; count < xPoints.length; count++ )
46             star.lineTo( xPoints[ count ], yPoints[ count ] );
47
48         // close the shape
49         star.closePath();
50
51         // translate the origin to (200, 200)
52         g2d.translate( 200, 200 );
```

Fig. 11.23 Demonstrating some Java2D shapes

```

53
54     // rotate around origin and draw stars in random colors
55     for ( int count = 1; count <= 20; count++ ) {
56
57         // rotate coordinate system
58         g2d.rotate( Math.PI / 10.0 );
59
60         // set random drawing color
61         g2d.setColor( new Color(
62             ( int ) ( Math.random() * 256 ),
63             ( int ) ( Math.random() * 256 ),
64             ( int ) ( Math.random() * 256 ) ) );
65
66         // draw filled star
67         g2d.fill( star );
68     }
69
70 } // end method paint
71
72 // execute application
73 public static void main( String args[] )
74 {
75     Shapes2 application = new Shapes2();
76
77     application.setDefaultCloseOperation(
78         JFrame.EXIT_ON_CLOSE );
79 }
80
81 } // end class Shapes2

```

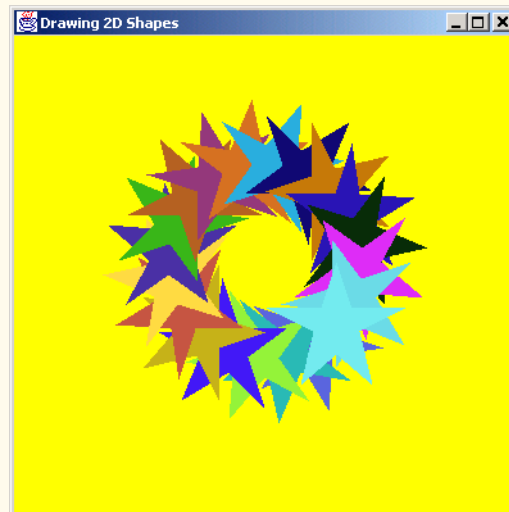


Fig. 11.23 Demonstrating some Java2D shapes

Lines 31–34 define two **int** arrays representing the *x*- and *y*-coordinates of the points in the star. Line 39 defines **GeneralPath** object **star**.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

Line 42 uses **GeneralPath** method **moveTo** to specify the first point in the **star**. The **for** structure at lines 45–46 use **GeneralPath** method **lineTo** to draw a line to the next point in the **star**. Each new call to **lineTo** draws a line from the previous point to the current point. Line 49 uses **GeneralPath** method **closePath** to draw a line from the last point to the point specified in the last call to **moveTo**. This completes the general path.

Line 52 uses **Graphics2D** method **translate** to move the drawing origin to location (200, 200). All drawing operations now use location (200, 200) as (0, 0).

The **for** structure at line 55–68 draws the **star** 20 times by rotating it around the new origin point. Line 58 uses **Graphics2D** method **rotate** to rotate the next displayed shape. The argument specifies the rotation angle in radians (with $360^\circ = 2\pi$ radians). Line 67 uses **Graphics2D** method **fill** to draw a filled version of the **star**.

11.10 (Optional Case Study) Thinking About Objects: Designing Interfaces with the UML

In Section 10.22, we incorporated event handling into our simulation by modifying the collaboration diagram that deals with passengers entering and exiting the elevator. We included both event handling and inheritance in that diagram. The **Elevator** informs its **Door** of the **Elevator**'s arrival. This **Door** opens the arrival **Floor**'s **Door** by obtaining its handle through a **Location** object (which was included in the arrival event), and potentially two **Person** objects exit and enter the **Elevator** after both **Doors** open. We also discussed listener interfaces. In this section, we represent our listener interface with the UML.

Realizations

The UML expresses the relationship between a class and an interface through a *realization*. A class *realizes*, or implements, the behaviors of an interface. A class diagram can show a realization between classes and interfaces. As mentioned in “Thinking About Objects” Section 3.8, the UML provides two notations to draw a class diagram—the complete diagram and the elided (condensed) diagram. Figure 11.24 shows the complete class diagram that models the realization between class **Person** and interface **DoorListener**. The diagram is similar to the generalization diagram, except that the arrow expressing the relationship is dashed instead of solid. Note that the middle compartment in interface **DoorListener** is empty, because interfaces do not contain variables—interfaces can contain constants, but interface **DoorListener** does not contain any constants. Lastly, note the word **interface** placed in guillemets (« ») located in the first compartment of interface **DoorListener**. This notation distinguishes interface **DoorListener** as an interface in our system. Items placed in guillemets are called *stereotypes* in the UML. A stereotype indicates an element's role—or purpose—in a UML diagram.

Figure 11.25 shows the alternate way to represent the realization of class **Person** and interface **DoorListener** in the UML. Figure 11.25 is the elided diagram of Fig. 11.24. The small circle represents the interface, and the solid line represents the realization. By hiding its operations, we condense the interface, making it easier to read; however, in doing so, we sacrifice the information about its behaviors. When constructing an elided diagram, common practice is to place the information regarding any behavior in a separate diagram—for example, we place the full **DoorListener** class in the class diagram of Fig. 11.28.

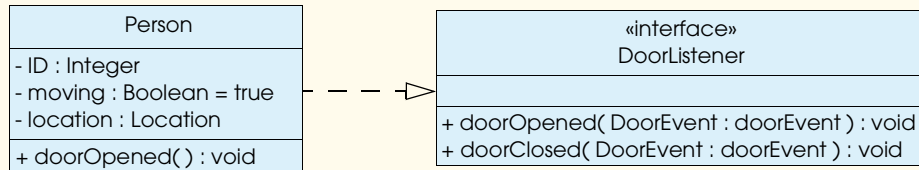


Fig. 11.24 Class diagram that models class **Person** realizing interface **DoorListener**.



Fig. 11.25 Elided class diagram that models class **Person** realizing interface **DoorListener**.

Forward engineering from the UML to implemented Java code benefits from well-constructed realization diagrams. When declaring any class, specify the realization between that class and its interface—that class will “implement” the interface and override the interface’s methods. For example, we use Fig. 11.24 to begin constructing **Person.java**:

```

public class Person implements DoorListener {

    // constructor
    public Person() {}

    // methods of DoorListener
    public void doorOpened( DoorEvent doorEvent ) {}
    public void doorClosed( DoorEvent doorEvent ) {}
}
  
```

Figure 11.26 shows the Java complete implementation for Fig. 11.24. Lines 6–8 and lines 14–15 include the attributes and operations of **Person**, respectively—in this case, the **doorOpened** operation (line 14) was already included when we implemented the **DoorListener** interface, so we include only the attributes of **Person**:

```

1 // Person.java
2 // Generated from Fig. 11.24
3 public class Person implements DoorListener {
  
```

Fig. 11.26 Class **Person** is generated from Fig. 11.24 (part 1 of 2).

```

4
5  // attributes
6  private int ID;
7  private boolean moving = true;
8  private Location location;
9
10 // constructor
11 public Person() {}
12
13 // methods of DoorListener
14 public void doorOpened( DoorEvent doorEvent ) {}
15 public void doorClosed( DoorEvent doorEvent ) {}
16 }

```

Fig. 11.26 Class **Person** is generated from Fig. 11.24 (part 2 of 2).

When a **Door** opens or closes, that **Door** invokes only those methods declared in interface **DoorListener**, but only if the **Person** has registered with that **Door** to receive **DoorEvents**. Finally, we present the elided class diagram that models the realizations in our elevator model in Fig. 11.27—the elided diagram does not contain any interface methods (making the diagram easier to read), so we present the class diagram for interfaces in Fig. 11.28, which shows all interface methods. Refer to these diagrams when studying the elevator simulation implementation in Appendices G, H and I.

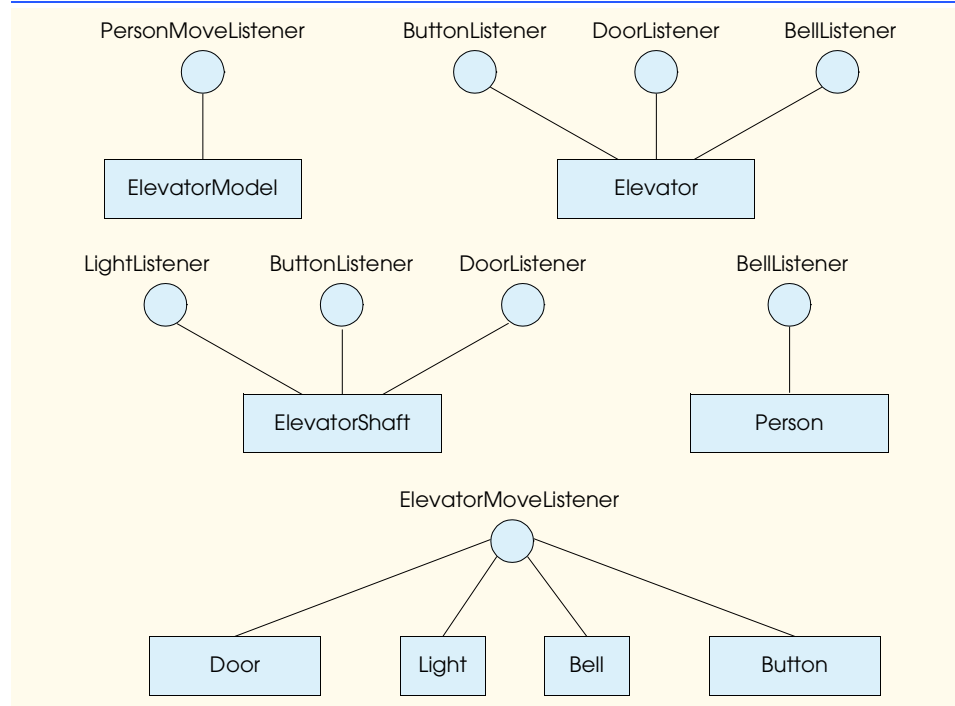


Fig. 11.27 Class diagram that models realizations in the elevator model.

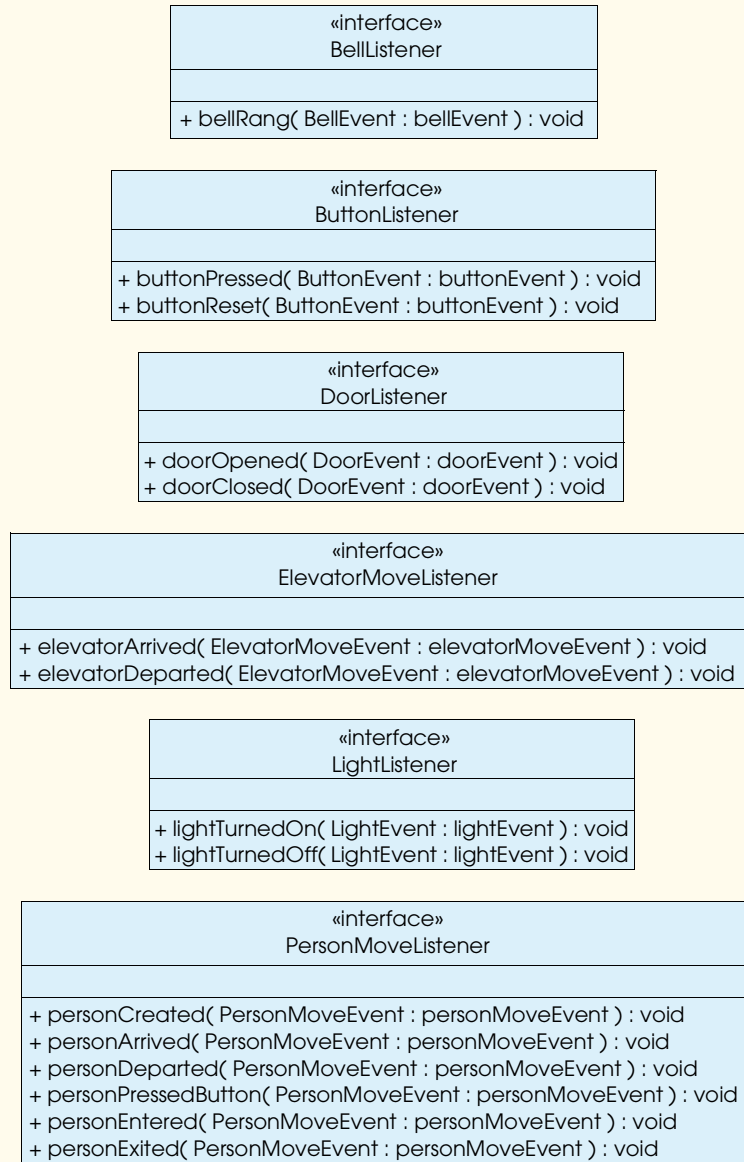


Fig. 11.28 Class diagram for listener interfaces.

According to Fig. 11.27, classes **Door**, **Light**, **Bell** and **Button** implement interface **ElevatorMoveListener**. Class **Elevator** implements interfaces **ButtonListener**, **DoorListener** and **BellListener**. Class **ElevatorModel** implements interface **PersonMoveListener**. Class **ElevatorShaft** implements interfaces **LightListener**, **ButtonListener** and **DoorListener**. Lastly, class

Person implements interface **DoorListener**. We reexamine Fig. 11.27 in Appendix H when we begin coding our model.

In this section we showed how to represent interfaces and realizations with the UML. We also presented class diagrams showing the listener interfaces and their realizations for our elevator simulation. In “Thinking About Objects” Section 12.16, we model how the user interacts with our simulation.

SUMMARY

- A coordinate system is a scheme for identifying every possible point on the screen.
- The upper-left corner of a GUI component has the coordinates $(0, 0)$. A coordinate pair is composed of an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate).
- Coordinate units are measured in pixels. A pixel is a display monitor’s smallest unit of resolution.
- A graphics context enables drawing on the screen in Java. A **Graphics** object manages a graphics context by controlling how information is drawn.
- **Graphics** objects contain methods for drawing, font manipulation, color manipulation and so on.
- Method **paint** is normally called in response to an *event*, such as uncovering a window.
- Method **repaint** requests a call to **Component** method **update** as soon as possible to clear the **Component**’s background of any previous drawing, then **update** calls **paint** directly.
- Class **Color** defines methods and constants for manipulating colors in a Java program.
- Java uses RGB colors in which the red, green and blue color components are integers in the range from 0 to 255 or floating-point values in the range from 0.0 to 1.0. The larger the RGB value, the greater the amount of that particular color.
- **Color** methods **getRed**, **getGreen** and **getBlue** return integer values from 0 to 255 representing the amount of red, green and blue in a **Color**.
- Class **Color** provides 13 predefined **Color** objects.
- **Graphics** method **getColor** returns a **Color** object representing the current drawing color. **Graphics** method **setColor** sets the current drawing color.
- Java provides class **JColorChooser** to display a dialog for selecting colors.
- **static** method **showDialog** of class **JColorChooser** displays a color chooser dialog. This method returns the selected **Color** object (**null**, if none is selected).
- The default **JColorChooser** dialog allows you to select a color from a variety of color swatches. The **HSB** tab allows you to select a color based on hue, saturation and brightness. The **RGB** tab allows you to select a color by using sliders for the red, green and blue components of the color.
- **Component** method **setBackground** (one of the many **Component** methods that can be used on most GUI components) changes the background color of a component.
- Class **Font**’s constructor takes three arguments—the font name, the font style and the font size. The font name is any font currently supported by the system. The font style is **Font.PLAIN**, **Font.ITALIC** or **Font.BOLD**. The font size is measured in points.
- **Graphics** method **setFont** sets the drawing font.
- Class **FontMetrics** defines several methods for obtaining font metrics.
- **Graphics** method **getFontMetrics** with no arguments obtains the **FontMetrics** object for the current font. **Graphics** method **getFontMetrics** that receives a **Font** argument returns a corresponding **FontMetrics** object.

- Methods **draw3DRect** and **fill3DRect** take five arguments specifying the top-left corner of the rectangle, the **width** and **height** of the rectangle and whether the rectangle is raised (**true**) or lowered (**false**).
- Methods **drawRoundRect** and **fillRoundRect** draw rectangles with rounded corners. Their first two arguments specify the upper-left corner, the third and fourth arguments specify the **width** and **height**, and the last two arguments—**arcWidth** and **arcHeight**—determine the horizontal and vertical diameters of the arcs used to represent the corners.
- Methods **drawOval** and **fillOval** take the same arguments—the top-left coordinate and the **width** and the **height** of the bounding rectangle that contains the oval.
- An arc is a portion of an oval. Arcs sweep from a starting angle the number of degrees specified by their arc angle. The starting angle specifies where the arc begins and the arc angle specifies the total number of degrees through which the arc sweeps. Arcs that sweep counterclockwise are measured in positive degrees and arcs that sweep clockwise are measured in negative degrees.
- Methods **drawArc** and **fillArc** take the same arguments—the top-left coordinate, the **width** and the **height** of the bounding rectangle that contains the arc and the **startAngle** and **arcAngle** that define the sweep of the arc.
- Polygons are multisided shapes. Polylines are a series of connected points.
- One **Polygon** constructor receives an array containing the *x*-coordinate of each point, an array containing the *y*-coordinate of each point and the number of points in the polygon.
- One version of **Graphics** method **drawPolygon** displays a **Polygon** object. Another version receives an array containing the *x*-coordinate of each point, an array containing the *y*-coordinate of each point and the number of points in the polygon and displays the corresponding polygon.
- **Graphics** method **drawPolyline** displays a series of connected lines specified by its arguments (an array containing the *x*-coordinate of each point, an array containing the *y*-coordinate of each point and the number of points).
- **Polygon** method **addPoint** adds pairs of *x*- and *y*-coordinates to a **Polygon**.
- The Java2D API provides advanced two-dimensional graphics capabilities for processing line art, text and images.
- To access the **Graphics2D** capabilities, downcast the **Graphics** reference passed to **paint** to a **Graphics2d** reference.
- **Graphics2D** method **setPaint** sets the **Paint** object that determines the color and texture for the shape to display. A **Paint** object is an object of any class that implements interface **java.awt.Paint**. The **Paint** object can be a **Color** or an instance of the Java2D API's **GradientPaint**, **SystemColor** or **TexturePaint** classes.
- Class **GradientPaint** draws a shape in a gradually changing color called a gradient.
- **Graphics2D** method **fill** draws a filled **Shape** object. The **Shape** object is an instance of any class that implements interface **Shape**.
- The **Ellipse2D.Double** constructor receives four arguments specifying the bounding rectangle for the ellipse to display.
- **Graphics2D** method **setStroke** sets the characteristics of the lines used to draw a shape. Method **setStroke** requires a **Stroke** object as its argument. The **Stroke** object is an instance of any class that implements interface **Stroke**, such as a **BasicStroke**.
- **Graphics2D** method **draw** draws a **Shape** object. The **Shape** object is an instance of any class that implements interface **Shape**.
- The **Rectangle2D.Double** constructor receives four arguments specifying the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height of the rectangle.

- Class **BufferedImage** can be used to produce images in color and gray scale.
- A **TexturePaint** object uses the image stored in its associated **BufferedImage** as the fill texture for a filled-in shape.
- The **RoundRectangle2D.Double** constructor receives six arguments specifying the rectangle's dimensions and the arc width and arc height used to determine the rounding of the corners.
- The **Arc2D.Double** constructor's first four arguments specify the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height of the bounding rectangle for the arc. The fifth argument specifies the start angle. The sixth argument specifies the end angle. The last argument specifies the type of arc (**Arc2D.PIE**, **Arc2D.CHORD** or **Arc2D.OPEN**).
- The **Line2D.Double** constructor's arguments specify starting and ending line coordinates.
- A general path is a shape constructed from straight lines and complex curves represented with an object of class **GeneralPath** (package **java.awt.geom**).
- **GeneralPath** method **moveTo** specifies the first point in a general path. **GeneralPath** method **lineTo** draws a line to the next point in the general path. Each new call to **lineTo** draws a line from the previous point to the current point. **GeneralPath** method **closePath** draws a line from the last point to the point specified in the last call to **moveTo**.
- **Graphics2D** method **translate** moves the drawing origin to a new location. All drawing operations now use that location as $(0, 0)$.
- **Graphics2D** method **rotate** to rotate the next that is displayed. Its argument specifies the rotation angle in radians (with $360^\circ = 2\pi$ radians).

TERMINOLOGY

addPoint method

angle

arc bounded by a rectangle

arc height

arc sweeping through an angle

arc width

Arc2D.Double class

ascent

background color

baseline

bounding rectangle

BufferedImage class

closed polygon

closePath method

Color class

Component class

coordinate

coordinate system

degree

descent

draw an arc

draw method

draw3DRect method

drawArc method

drawLine method

drawOval method

drawPolygon method

drawPolyline method

drawRect method

drawRoundRect method

Ellipse2D.Double class

event

event-driven process

fill method

fill3DRect method

fillArc method

filled polygon

fillOval method

fillPolygon method

fillRect method

fillRoundRect method

font

Font class

font metrics

font name

font style

FontMetrics class

GeneralPath class

getAscent method

getBlue method

getDescent method

getFamily method

getFont method	pixel
getFontList method	point
getFontMetrics method	polygon
getGreen method	Polygon class
getHeight method	positive degrees
getLeading method	Rectangle2D.Double class
getName method	repaint method
getRed method	RGB value
getSize method	rotate method
getStyle method	RoundRectangle2D.Double class
GradientPaint class	SansSerif font
Graphics class	Serif font
graphics context	setColor method
graphics object	setFont method
Graphics2D class	setPaint method
isBold method	setStroke method
isItalic method	Shape interface
isPlain method	Stroke interface
Java2D API	SystemColor class
leading	TexturePaint class
Line2D.Double class	translate method
lineTo method	update method
Monospaced font	vertical component
moveTo method	x-axis
negative degrees	x-coordinate
Paint interface	y-axis
paint method	y-coordinate

SELF-REVIEW EXERCISES

11.1 Fill in the blanks in each of the following statements:

- In Java2D, method _____ of class _____ sets the characteristics of a line used to draw a shape.
- Class _____ helps define the fill for a shape such that the fill gradually changes from one color to another.
- The _____ method of class **Graphics** draws a line between two points.
- RGB is short for _____, _____ and _____.
- Font sizes are measured in units called _____.
- Class _____ helps define the fill for a shape using a pattern drawn in a **BufferedImage**.

11.2 State whether each of the following is *true* or *false*. If *false*, explain why.

- The first two arguments of **Graphics** method **drawOval** specify the center coordinate of the oval.
- In the Java coordinate system, *x* values increase from left to right.
- Method **fillPolygon** draws a solid polygon in the current color.
- Method **drawArc** allows negative angles.
- Method **getSize** returns the size of the current font in centimeters.
- Pixel coordinate (0, 0) is located at the exact center of the monitor.

11.3 Find the error(s) in each of the following and explain how to correct the error(s). Assume that **g** is a **Graphics** object.

- a) `g.setFont( "SansSerif" );`
- b) `g.erase( x, y, w, h );` // clear rectangle at (x, y)
- c) `Font f = new Font( "Serif", Font.BOLDITALIC, 12 );`
- d) `g.setColor( Color.Yellow );` // change color to yellow

ANSWERS TO SELF-REVIEW EXERCISES

- 11.1** a) `setStroke`, `Graphics2D`. b) `GradientPaint`. c) `drawLine`. d) red, green, blue. e) points. f) `TexturePaint`.
- 11.2** a) False. The first two arguments specify the upper-left corner of the bounding rectangle.
 b) True.
 c) True.
 d) True.
 e) False. Font sizes are measured in points.
 f) False. The coordinate (0,0) corresponds to the upper-left corner of a GUI component on which drawing occurs.
- 11.3** a) The `setFont` method takes a `Font` object as an argument—not a `String`.
 b) The `Graphics` class does not have an `erase` method. The `clearRect` method should be used.
 c) `Font.BOLDITALIC` is not a valid font style. To get a bold italic font, use `Font.BOLD + Font.ITALIC`.
 d) `Yellow` should begin with a lowercase letter: `g.setColor( Color.yellow );`.

EXERCISES

- 11.4** Fill in the blanks in each of the following statements:
- a) Class _____ of the Java2D API is used to define ovals.
 - b) Methods `draw` and `fill` of class `Graphics2D` require an object of type _____ as their argument.
 - c) The three constants that specify font style are _____, _____ and _____.
 - d) `Graphics2D` method _____ sets the painting color for Java2D shapes.
- 11.5** State whether each of the following is *true* or *false*. If *false*, explain why.
- a) The `drawPolygon` method automatically connects the endpoints of the polygon.
 - b) The `drawLine` method draws a line between two points.
 - c) The `fillArc` method uses degrees to specify the angle.
 - d) In the Java coordinate system, y values increase from top to bottom.
 - e) The `Graphics` class inherits directly from class `Object`.
 - f) The `Graphics` class is an `abstract` class.
 - g) The `Font` class inherits directly from class `Graphics`.
- 11.6** Write a program that draws a series of eight concentric circles. The circles should be separated by 10 pixels. Use the `drawOval` method of class `Graphics`.
- 11.7** Write a program that draws a series of eight concentric circles. The circles should be separated by 10 pixels. Use the `drawArc` method.
- 11.8** Modify your solution to Exercise 11.6 to draw the ovals by using instances of class `Ellipse2D.Double` and method `draw` of class `Graphics2D`.
- 11.9** Write a program that draws lines of random lengths in random colors.
- 11.10** Modify your solution to Exercise 11.9 to draw random lines, in random colors and random line thicknesses. Use class `Line2D.Double` and method `draw` of class `Graphics2D` to draw the lines.

11.11 Write a program that displays randomly generated triangles in different colors. Each triangle should be filled with a different color. Use class **GeneralPath** and method **fill** of class **Graphics2D** to draw the triangles.

11.12 Write a program that randomly draws characters in different font sizes and colors.

11.13 Write a program that draws an 8-by-8 grid. Use the **drawLine** method.

11.14 Modify your solution to Exercise 11.13 to draw the grid using instances of class **Line2D.Double** and method **draw** of class **Graphics2D**.

11.15 Write a program that draws a 10-by-10 grid. Use the **drawRect** method.

11.16 Modify your solution to Exercise 11.15 to draw the grid by using instances of class **Rectangle2D.Double** and method **draw** of class **Graphics2D**.

11.17 Write a program that draws a tetrahedron (a pyramid). Use class **GeneralPath** and method **draw** of class **Graphics2D**.

11.18 Write a program that draws a cube. Use class **GeneralPath** and method **draw** of class **Graphics2D**.

11.19 In Exercise 3.9, you wrote an applet that input the radius of a circle from the user and displayed the circle's diameter, circumference and area. Modify your solution to Exercise 3.9 to read a set of coordinates in addition to the radius. Then draw the circle and display the circle's diameter, circumference and area, using an **Ellipse2D.Double** object to represent the circle and method **draw** of class **Graphics2D** to display the circle.

11.20 Write an application that simulates a screen saver. The application should randomly draw lines using method **drawLine** of class **Graphics**. After drawing 100 lines, the application should clear itself and start drawing lines again. To allow the program to draw continuously, place a call to **repaint** as the last line in method **paint**. Do you notice any problems with this on your system?

11.21 Here is a peek ahead. Package **javax.swing** contains a class called **Timer** that is capable of calling method **actionPerformed** of interface **ActionListener** at a fixed time interval (specified in milliseconds). Modify your solution to Exercise 11.20 to remove the call to **repaint** from method **paint**. Define your class so it implements **ActionListener**. (The **actionPerformed** method should simply call **repaint**.) Define an instance variable of type **Timer** called **timer** in your class. In the constructor for your class, write the following statements:

```
timer = new Timer( 1000, this );
timer.start();
```

This creates an instance of class **Timer** that will call **this** object's **actionPerformed** method every 1000 milliseconds (i.e., every second).

11.22 Modify your solution to Exercise 11.21 to enable the user to enter the number of random lines that should be drawn before the application clears itself and starts drawing lines again. Use a **JTextField** to obtain the value. The user should be able to type a new number into the **JTextField** at any time during the program's execution. Use an inner class definition to perform event handling for the **JTextField**.

11.23 Modify your solution to Exercise 11.21 such that it uses random number generation to choose different shapes to display (use methods of class **Graphics**).

11.24 Modify your solution to Exercise 11.23 to use classes and drawing capabilities of the Java2D API. For shapes such as rectangles and ellipses, draw them with randomly generated gradients (use class **GradientPaint** to generate the gradient).

11.25 Write a graphical version of your solution to Exercise 6.37—the *Towers of Hanoi*. After studying Chapter 18, you will be able to implement a version of this exercise using Java's image, animation and audio capabilities.

11.26 Modify the die-rolling program of Fig. 7.8 so that it updates the counts for each side of the die after each roll. Convert the application into a windowed application (i.e., a subclass of **JFrame**) and use **Graphics** method **drawString** to output the totals.

11.27 Modify your solution to Exercise 7.21—*Turtle Graphics*—to add a graphical user interface using **JTextFields** and **JButtons**. Also, draw lines rather than drawing asterisks (*). When the turtle graphics program specifies a move, translate the number of positions into a number of pixels on the screen by multiplying the number of positions by 10 (or any value you choose). Implement the drawing with Java2D API features.

11.28 Produce a graphical version of the Knight's Tour problem (Exercises 7.22, 7.23 and 7.26). As each move is made, the appropriate cell of the chessboard should be updated with the proper move number. If the result of the program is a *full tour* or a *closed tour*, the program should display an appropriate message. If you would like, use class **Timer** (see Exercise 11.24) to help animate the Knight's Tour. Every second, the next move should be made.

11.29 Produce a graphical version of the *Tortoise and the Hare* simulation (Exercise 7.41). Simulate the mountain by drawing an arc that extends from the bottom-left of the window to the top-right of the window. The tortoise and the hare should race up the mountain. Implement the graphical output so the tortoise and the hare are actually printed on the arc every move. [Note: Extend the length of the race from 70 to 300 to allow yourself a larger graphics area.]

11.30 Produce a graphical version of the *Maze Traversal* problem (Exercises 7.38-7.40). Use the mazes you produced as guides for creating the graphical versions. While the maze is being solved, a small circle should be displayed in the maze indicating the current position. If you would like, use class **Timer** (see Exercise 11.24) to help animate the traversal of the maze. Every second, the next move should be made.

11.31 Produce a graphical version of the *Bucket Sort* (Exercise 7.28) that shows each value being placed into the appropriate bucket and eventually being copied back to the original array.

11.32 Write a program that uses method **drawPolyline** to draw a spiral.

11.33 Write a program that inputs four numbers and graphs the numbers as a pie chart. Use class **Arc2D.Double** and method **fill** of class **Graphics2D** to perform the drawing. Draw each piece of the pie in a separate color.

11.34 Write an applet that inputs four numbers and graphs the numbers as a bar graph. Use class **Rectangle2D.Double** and method **fill** of class **Graphics2D** to perform the drawing. Draw each bar in a different color.