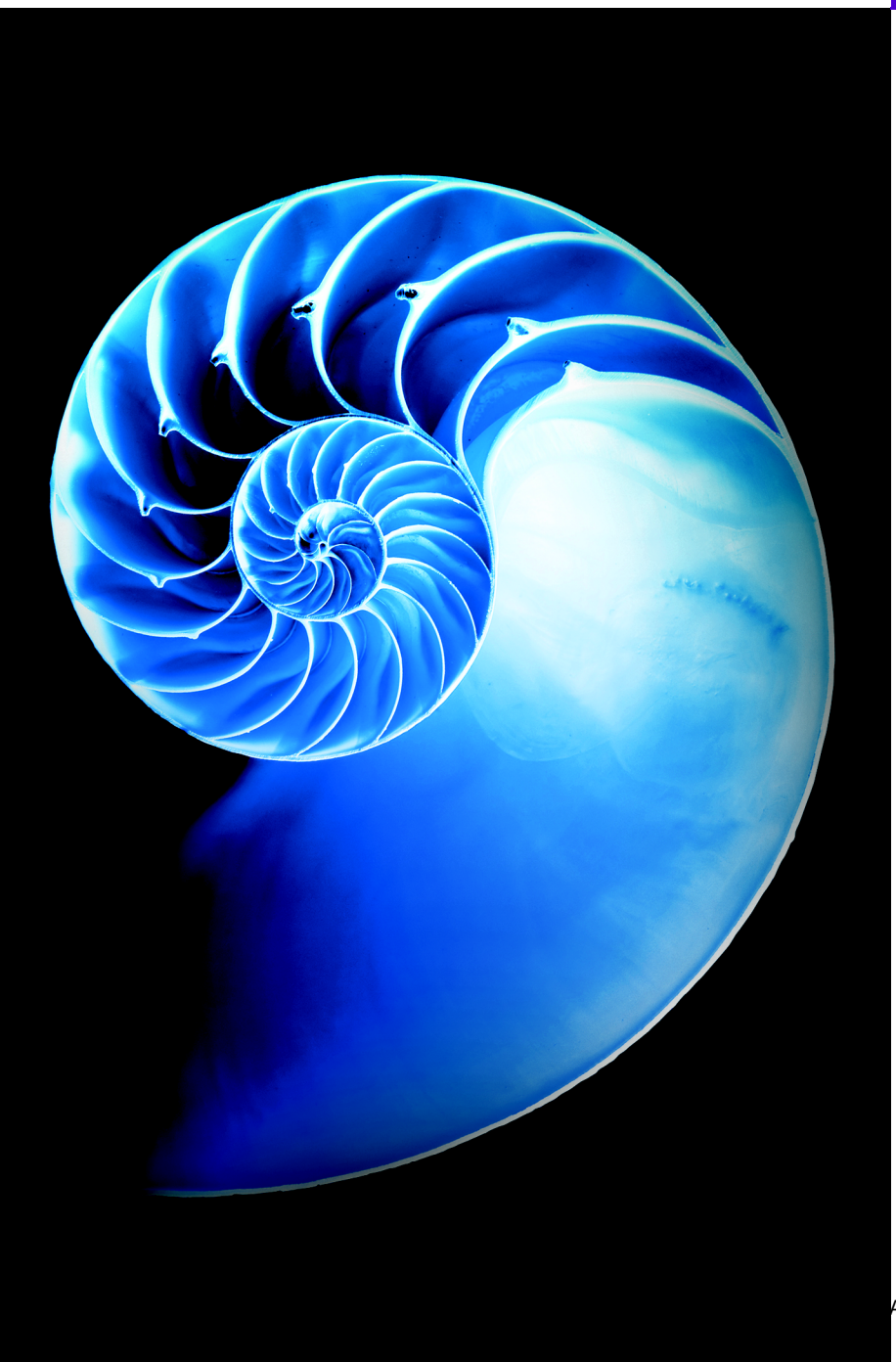
27

Graphics and Java 2D



Objectives

In this chapter you'll:

- Understand graphics contexts and graphics objects.
- Manipulate colors and fonts.
- Use methods of class `Graphics` to draw various shapes.
- Use methods of class `Graphics2D` from the `Java 2D` API to draw various shapes.
- Specify `Paint` and `Stroke` characteristics of shapes displayed with `Graphics2D`.

Outline

27.1 Introduction

[*Note:* JavaFX (Chapters 12, 13 and 22) is Java's GUI, graphics and multimedia API of the future. This chapter is provided *as is* for those still interested in Swing GUIs.]

In this chapter, we overview several of Java's capabilities for drawing two-dimensional shapes, controlling colors and controlling fonts. Part of Java's initial appeal was its support for graphics that enabled programmers to visually enhance their applications. Java contains more sophisticated drawing capabilities as part of the Java 2D API (presented in this chapter) and its successor technology JavaFX (presented in Chapter 12 and two online chapters). This chapter begins by introducing many of Java's original drawing capabilities. Next we present several of the more powerful Java 2D capabilities, such as controlling the *style* of lines used to draw shapes and the way shapes are *filled* with *colors* and *patterns*. The classes that were part of Java's original graphics capabilities are now considered to be part of the Java 2D API.

Figure 27.1 shows a portion of the class hierarchy that includes various graphics classes and Java 2D API classes and interfaces covered in this chapter. Class **`Color`** contains methods and constants for manipulating colors. Class `JComponent` contains method `paintComponent`, which is used to draw graphics on a component. Class **`Font`** contains methods and constants for manipulating fonts. Class **`FontMetrics`** contains methods for obtaining *font* information. Class **`Graphics`** contains methods for drawing strings, lines, rectangles and other shapes. Class **`Graphics2D`**, which extends class `Graphics`, is used for drawing with the Java 2D API. Class **`Polygon`** contains methods for creating *polygons*. The bottom half of the figure lists several classes and interfaces from the Java 2D 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`, `Line2D`, `Arc2D`, `Ellipse2D`, `Rectangle2D` and `RoundRectangle2D` represent several Java 2D shapes.

To begin drawing in Java, we must first understand Java's **coordinate system** (Fig. 27.2), which is a scheme for identifying every *point* on the screen. By default, the *upper-left corner* of a GUI component (e.g., 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 left edge of the screen. The *y*-coordinate is the vertical distance moving *down* from the *top* of the screen. The ***x*-axis** describes every horizontal coordinate, and the ***y*-axis** every vertical coordinate. The coordinates are used to indicate where graphics should be displayed on a screen. Coordinate units are measured in **pixels** (which stands for "picture elements"). A pixel is a display monitor's *smallest unit of resolution*.

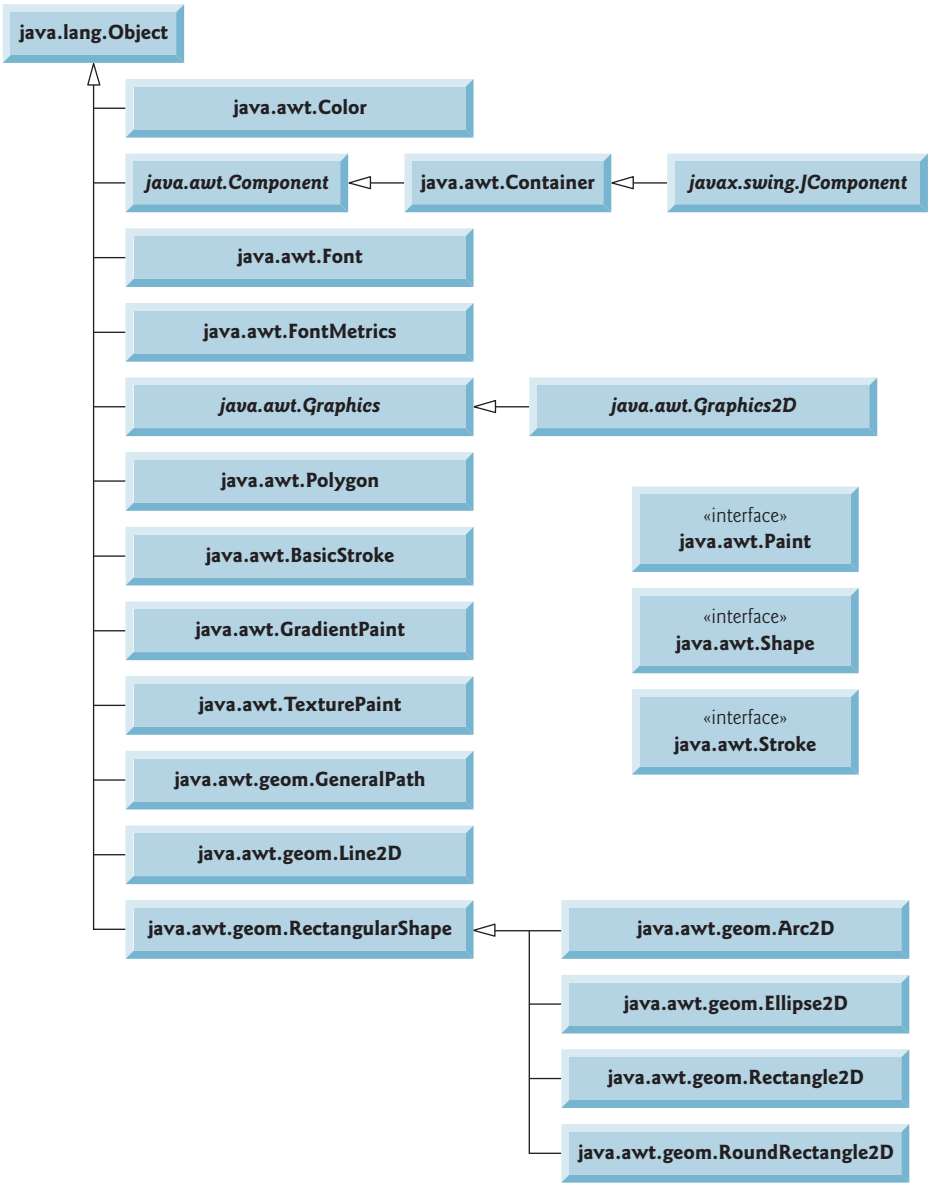


Fig. 27.1 | Classes and interfaces used in this chapter from Java's original graphics capabilities and from the Java 2D API.

Portability Tip 27.1

Different display monitors have different resolutions (i.e., the density of the pixels varies). This can cause graphics to appear in different sizes on different monitors or on the same monitor with different settings.

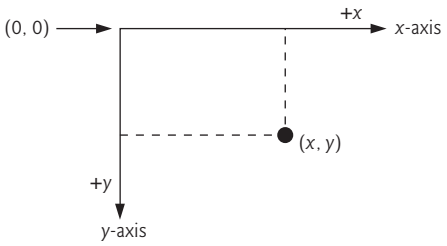


Fig. 27.2 | Java coordinate system. Units are measured in pixels.

27.2 Graphics Contexts and Graphics Objects

A **graphics context** enables drawing on the screen. A `Graphics` object manages a graphics context and draws pixels on the screen that represent *text* and other graphical objects (e.g., *lines*, *ellipses*, *rectangles* and other *polygons*). `Graphics` objects contain methods for *drawing*, *font manipulation*, *color manipulation* and the like.

Class `Graphics` is an `abstract` class (i.e., you cannot instantiate `Graphics` objects). This contributes to Java's portability. Because drawing is performed *differently* on every platform that supports Java, there cannot be only one implementation of the drawing capabilities across all systems. When Java is implemented on a particular platform, a subclass of `Graphics` is created that implements the drawing capabilities. This implementation is hidden by class `Graphics`, which supplies the interface that enables us to use graphics in a *platform-independent* manner.

Recall from Chapter 26 that class `Component` is the *superclass* for many of the classes in package `java.awt`. Class `JComponent` (package `javax.swing`), which inherits indirectly from class `Component`, contains a `paintComponent` method that can be used to draw graphics. Method `paintComponent` takes a `Graphics` object as an argument. This object is passed to the `paintComponent` method by the system when a lightweight Swing component needs to be repainted. The header for the `paintComponent` method is

```
public void paintComponent(Graphics g)
```

Parameter `g` receives a reference to an instance of the system-specific subclass of `Graphics`. The preceding method header should look familiar to you—it's the same one we used in some of the applications in Chapter 26. Actually, class `JComponent` is a *superclass* of `JPanel`. Many capabilities of class `JPanel` are inherited from class `JComponent`.

You seldom call method `paintComponent` directly, because drawing graphics is an *event-driven* process. As we mentioned in Chapter 11, Java uses a *multithreaded* model of program execution. Each thread is a *parallel* activity. Each program can have many threads. When you create a GUI-based application, one of those threads is known as the **event-dispatch thread (EDT)**—it's used to process all GUI events. All manipulation of the on-screen GUI components must be performed in that thread. When a GUI application executes, the application container calls method `paintComponent` (in the event-dispatch thread) for each lightweight component as the GUI is displayed. For `paintComponent` to be called again, an event must occur (such as *covering* and *uncovering* the component with another window).

If you need `paintComponent` to execute (i.e., if you want to update the graphics drawn on a Swing component), you can call method **`repaint`**, which returns `void`, takes no arguments and is inherited by all `JComponents` indirectly from class `Component` (package `java.awt`).

27.3 Color Control

Class `Color` declares methods and constants for manipulating colors in a Java program. The predeclared color constants are summarized in Fig. 27.3, and several color methods and constructors are summarized in Fig. 27.4. Two of the methods in Fig. 27.4 are `Graphics` methods that are specific to colors.

Color constant	RGB value
`public static final Color RED`	255, 0, 0
`public static final Color GREEN`	0, 255, 0
`public static final Color BLUE`	0, 0, 255
`public static final Color ORANGE`	255, 200, 0
`public static final Color PINK`	255, 175, 175
`public static final Color CYAN`	0, 255, 255
`public static final Color MAGENTA`	255, 0, 255
`public static final Color YELLOW`	255, 255, 0
`public static final Color BLACK`	0, 0, 0
`public static final Color WHITE`	255, 255, 255
`public static final Color GRAY`	128, 128, 128
`public static final Color LIGHT_GRAY`	192, 192, 192
`public static final Color DARK_GRAY`	64, 64, 64

Fig. 27.3 | `Color` constants and their RGB values.

Method	Description
`Color` constructors and methods	
`public Color(int r, int g, int b)`	Creates a color based on red, green and blue components expressed as integers from 0 to 255.
`public Color(float r, float g, float b)`	Creates a color based on red, green and blue components expressed as floating-point values from 0.0 to 1.0.
`public int getRed()`	Returns a value between 0 and 255 representing the red content.

Fig. 27.4 | `Color` methods and color-related `Graphics` methods. (Part 1 of 2.)

Method	Description
`public int getGreen()`	Returns a value between 0 and 255 representing the green content.
`public int getBlue()`	Returns a value between 0 and 255 representing the blue content.
`Graphics` methods for manipulating `Color`s	
`public Color getColor()`	Returns `Color` object representing current color for the graphics context.
`public void setColor(Color c)`	Sets the current color for drawing with the graphics context.

Fig. 27.4 | `Color` methods and color-related `Graphics` methods. (Part 2 of 2.)

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

Two of class `Color`'s constructors are shown in Fig. 27.4—one that takes three `int` arguments and one that takes three `float` arguments, with each argument specifying the amount of red, green and blue. The `int` values must be in the range 0–255 and the `float` values in the range 0.0–1.0. 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 amounts of red, green and blue, respectively. `Graphics` method **`getColor`** returns a `Color` object representing the `Graphics` object's current drawing color. `Graphics` method **`setColor`** sets the current drawing color.

Drawing in Different Colors

Figures 27.5–27.6 demonstrate several methods from Fig. 27.4 by drawing *filled rectangles* and `String`s in several different colors. When the application begins execution, class `ColorJPanel`'s `paintComponent` method (lines 10–37 of Fig. 27.5) is called to paint the window. Line 17 uses `Graphics` method `setColor` to set the 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 18 uses `Graphics` method **`fillRect`** to draw a *filled rectangle* in the current color. Method `fillRect` draws a rectangle based on its four arguments. The first two integer values represent the upper-left *x*-coordinate and upper-left *y*-coordinate, where the `Graphics` object begins drawing the rectangle. The third and fourth arguments are nonnegative integers that represent the width and the height of the rectangle in pixels, respectively. A rectangle drawn using method `fillRect` is filled by the current color of the `Graphics` object.

Line 19 uses `Graphics` method **`drawString`** to draw a `String` in the current color. The expression `g.getColor()` retrieves the current color from the `Graphics` object. We

```
1  // Fig. 13.5: ColorJPanel.java
2  // Changing drawing colors.
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import javax.swing.JPanel;
6
7  public class ColorJPanel extends JPanel
8  {
9     // draw rectangles and Strings in different colors
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14       this.setBackground(Color.WHITE);
15
16       // set new drawing color using integers
17       g.setColor(new Color(255, 0, 0));
18       g.fillRect(15, 25, 100, 20);
19       g.drawString("Current RGB: " + g.getColor(), 130, 40);
20
21       // set new drawing color using floats
22       g.setColor(new Color(0.50f, 0.75f, 0.0f));
23       g.fillRect(15, 50, 100, 20);
24       g.drawString("Current RGB: " + g.getColor(), 130, 65);
25
26       // set new drawing color using static Color objects
27       g.setColor(Color.BLUE);
28       g.fillRect(15, 75, 100, 20);
29       g.drawString("Current RGB: " + g.getColor(), 130, 90);
30
31       // display individual RGB values
32       Color color = Color.MAGENTA;
33       g.setColor(color);
34       g.fillRect(15, 100, 100, 20);
35       g.drawString("RGB values: " + color.getRed() + ", " +
36          color.getGreen() + ", " + color.getBlue(), 130, 115);
37    }
38 } // end class ColorJPanel
```

Fig. 27.5 | Changing drawing colors.

```
1  // Fig. 13.6: ShowColors.java
2  // Demonstrating Colors.
3  import javax.swing.JFrame;
4
5  public class ShowColors
6  {
7     // execute application
8     public static void main(String[] args)
9     {
10       // create frame for ColorJPanel
11       JFrame frame = new JFrame("Using colors");
```

Fig. 27.6 | Demonstrating `Colors`. (Part 1 of 2.)

```
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14         ColorJPanel colorJPanel = new ColorJPanel();
15         frame.add(colorJPanel);
16         frame.setSize(400, 180);
17         frame.setVisible(true);
18      }
19   } // end class ShowColors
```

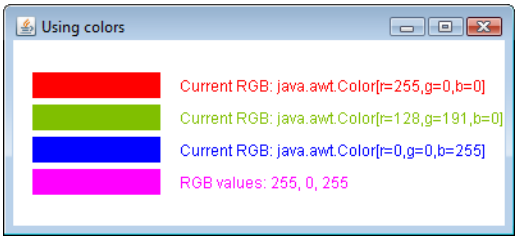


Fig. 27.6 | Demonstrating `Colors`. (Part 2 of 2.)

then concatenate the `Color` with string `"Current RGB: "`, resulting in an *implicit* call to class `Color`'s `toString` method. The `String` representation of a `Color` contains the class name and package (`java.awt.Color`) and the red, green and blue values.

Look-and-Feel Observation 27.1

People perceive colors differently. Choose your colors carefully to ensure that your application is readable, both for people who can perceive color and for those who are color blind. Try to avoid using many different colors in close proximity.

Lines 22–24 and 27–29 perform the same tasks again. Line 22 uses the `Color` constructor with three `float` arguments to create a dark green color (`0.50f` for red, `0.75f` for green and `0.0f` for blue). Note the syntax of the values. The letter `f` appended to a floating-point literal indicates that the literal should be treated as type `float`. Recall that by default, floating-point literals are treated as type `double`.

Line 27 sets the current drawing color to one of the predeclared `Color` constants (`Color.BLUE`). The `Color` constants are `static`, so they're created when class `Color` is loaded into memory at execution time.

The statement in lines 35–36 makes calls to `Color` methods `getRed`, `getGreen` and `getBlue` on the predeclared `Color.MAGENTA` constant. Method `main` of class `ShowColors` (lines 8–18 of Fig. 27.6) creates the `JFrame` that will contain a `ColorJPanel` object where the colors will be displayed.

Software Engineering Observation 27.1

To change the color, you must create a new `Color` object (or use one of the predeclared `Color` constants). Like `String` objects, `Color` objects are immutable (not modifiable).

The **`JColorChooser`** component (package `javax.swing`) enables application users to select colors. Figures 27.7–27.8 demonstrate a `JColorChooser` dialog. When you click the

Change Color button, a `JColorChooser` dialog appears. When you select a color and press the dialog's **OK** button, the background color of the application window changes.

```
1  // Fig. 13.7: ShowColors2JFrame.java
2  // Choosing colors with JColorChooser.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JColorChooser;
10 import javax.swing.JPanel;
11
12 public class ShowColors2JFrame extends JFrame
13 {
14    private final JButton changeColorJButton;
15    private Color color = Color.LIGHT_GRAY;
16    private final JPanel colorJPanel;
17
18    // set up GUI
19    public ShowColors2JFrame()
20    {
21       super("Using JColorChooser");
22
23       // create JPanel for display color
24       colorJPanel = new JPanel();
25       colorJPanel.setBackground(color);
26
27       // set up changeColorJButton and register its event handler
28       changeColorJButton = new JButton("Change Color");
29       changeColorJButton.addActionListener(
30          new ActionListener() // anonymous inner class
31          {
32             // display JColorChooser when user clicks button
33             @Override
34             public void actionPerformed(ActionEvent event)
35             {
36                color = JColorChooser.showDialog(
37                   ShowColors2JFrame.this, "Choose a color", color);
38
39                // set default color, if no color is returned
40                if (color == null)
41                   color = Color.LIGHT_GRAY;
42
43                // change content pane's background color
44                colorJPanel.setBackground(color);
45             } // end method actionPerformed
46          } // end anonymous inner class
47       ); // end call to addActionListener
48
49       add(colorJPanel, BorderLayout.CENTER);
```

Fig. 27.7 | Choosing colors with `JColorChooser`. (Part 1 of 2.)

```
50          add(changeColorJButton, BorderLayout.SOUTH);
51
52          setSize(400, 130);
53          setVisible(true);
54       } // end ShowColor2JFrame constructor
55    } // end class ShowColors2JFrame
```

Fig. 27.7 | Choosing colors with `JColorChooser`. (Part 2 of 2.)

```
1  // Fig. 13.8: ShowColors2.java
2  // Choosing colors with JColorChooser.
3  import javax.swing.JFrame;
4
5  public class ShowColors2
6  {
7     // execute application
8     public static void main(String[] args)
9     {
10       ShowColors2JFrame application = new ShowColors2JFrame();
11       application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12    }
13 } // end class ShowColors2
```

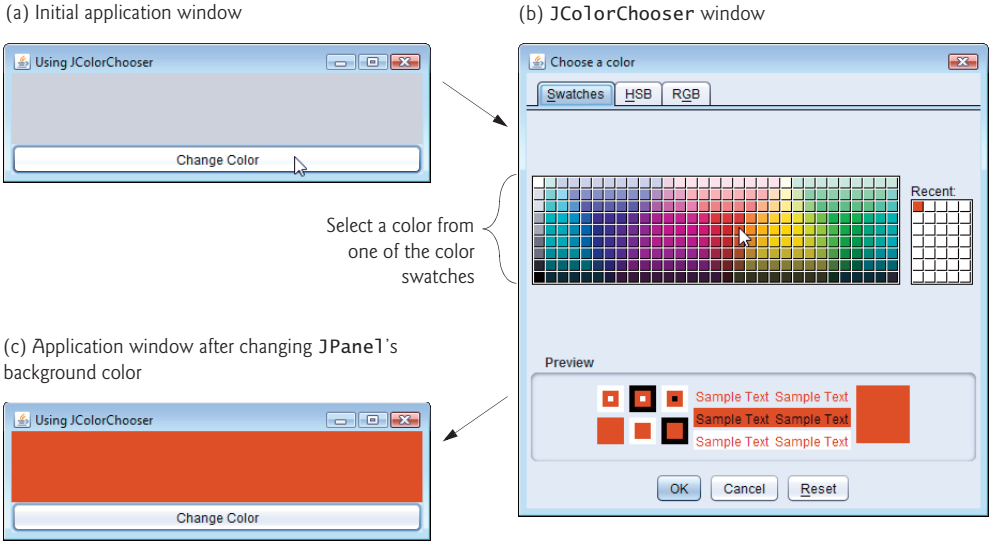


Fig. 27.8 | Choosing colors with `JColorChooser`.

Class `JColorChooser` provides `static` method **`showDialog`**, which creates a `JColorChooser` object, attaches it to a dialog box and displays the dialog. Lines 36–37 of Fig. 27.7 invoke this method to display the color-chooser dialog. Method `showDialog` returns the selected `Color` object, or `null` if the user presses **Cancel** or closes the dialog without pressing **OK**. The method 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 a reference to the window from which the dialog is displayed (in this case the `JFrame`, with the reference name `frame`). The dialog will be centered on the parent. If the parent is `null`, the dialog is centered on the screen. While the color-chooser dialog is on the screen, the user cannot interact with the parent component until the dialog is dismissed. This type of dialog is called a modal dialog.

After the user selects a color, lines 40–41 determine whether `color` is `null`, and, if so, set `color` to `Color.LIGHT_GRAY`. Line 44 invokes method `setBackground` to change the background color of the `JPanel`. Method `setBackground` is one of the many `Component` methods that can be used on most GUI components. The user can continue to use the **Change Color** button to change the background color of the application. Figure 27.8 contains method `main`, which executes the program.

Figure 27.8(b) shows the default `JColorChooser` dialog that allows the user to select a color from a variety of **color swatches**. There are 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**—values that are used to define the amount of light in a color. Visit `http://en.wikipedia.org/wiki/HSL_and_HSV` for more information on HSB. The **RGB** tab allows you to select a color by using sliders to select the red, green and blue components. The **HSB** and **RGB** tabs are shown in Fig. 27.9.

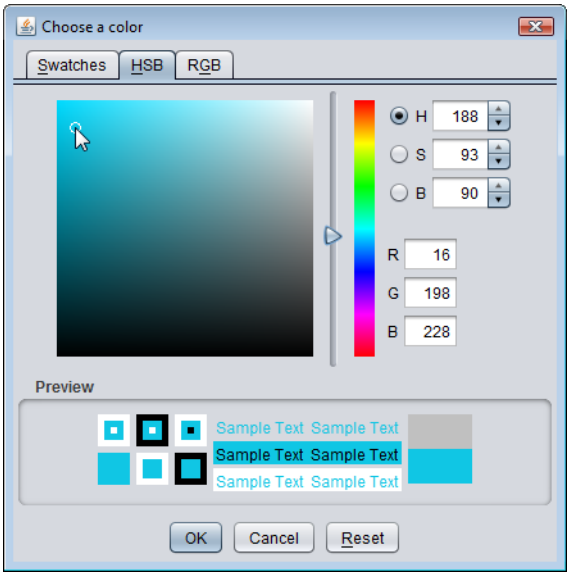


Fig. 27.9 | **HSB** and **RGB** tabs of the `JColorChooser` dialog. (Part 1 of 2.)

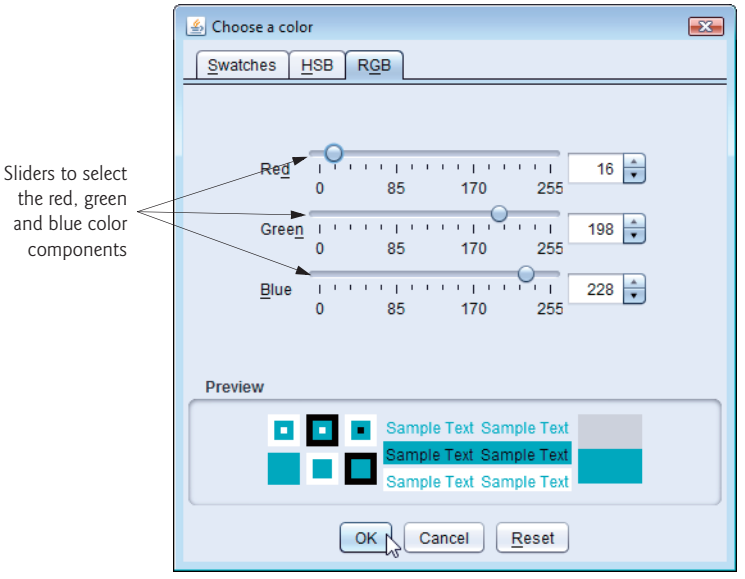


Fig. 27.9 | **HSB** and **RGB** tabs of the `JColorChooser` dialog. (Part 2 of 2.)

27.4 Manipulating Fonts

This section introduces methods and constants for manipulating fonts. Most font methods and font constants are part of class `Font`. Some constructors, methods and constants of class `Font` and class `Graphics` are summarized in Fig. 27.10.

Method or constant	Description
`Font` constants, constructors and methods	
`public static final int PLAIN`	A constant representing a plain font style.
`public static final int BOLD`	A constant representing a bold font style.
`public static final int ITALIC`	A constant representing an italic font style.
`public Font(String name, int style, int size)`	Creates a `Font` object with the specified font name, style and size.
`public int getStyle()`	Returns an `int` indicating the current font style.
`public int getSize()`	Returns an `int` indicating the current font size.
`public String getName()`	Returns the current font name as a string.
`public String getFamily()`	Returns the font's family name as a string.
`public boolean isPlain()`	Returns `true` if the font is plain, else `false`.
`public boolean isBold()`	Returns `true` if the font is bold, else `false`.
`public boolean isItalic()`	Returns `true` if the font is italic, else `false`.

Fig. 27.10 | `Font`-related methods and constants. (Part 1 of 2.)

Method or constant	Description
`Graphics` methods for manipulating `Font`s	
`public Font getFont()`	Returns a `Font` object reference representing the current font.
`public void setFont(Font f)`	Sets the current font to the font, style and size specified by the `Font` object reference `f`.

Fig. 27.10 | `Font`-related methods and constants. (Part 2 of 2.)

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 on which the program is running, such as standard Java fonts `Monospaced`, `SansSerif` and `Serif`. The font style is **`Font.PLAIN`**, **`Font.ITALIC`** or **`Font.BOLD`** (each is a `static` field 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 27.2

The number of fonts varies across systems. Java provides five font names—`Serif`, `Monospaced`, `SansSerif`, `Dialog` and `DialogInput`—that can be used on all Java platforms. The Java runtime environment (JRE) on each platform maps these logical font names to actual fonts installed on the platform. The actual fonts used may vary by platform.

The application of Figs. 27.11–27.12 displays text in four different fonts, with each font in a different size. Figure 27.11 uses the `Font` constructor to initialize `Font` objects (in lines 17, 21, 25 and 30) that are each passed 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. Each font's information is displayed in lines 18, 22, 26 and 31–32 using method `drawString`. The coordinates passed to `drawString` correspond to the lower-left corner of the baseline of the font. Line 29 changes the drawing color to red, so the next string displayed appears in red. Lines 31–32 display information about the final `Font` object. Method **`getFont`** of class `Graphics` returns a `Font` object representing the current font. Method **`getName`** returns the current font name as a string. Method **`getSize`** returns the font size in points.

Software Engineering Observation 27.2

To change the font, you must create a new `Font` object. `Font` objects are immutable—class `Font` has no set methods to change the characteristics of the current font.

```
1  // Fig. 13.11: FontJPanel.java
2  // Display strings in different fonts and colors.
3  import java.awt.Font;
```

Fig. 27.11 | Display strings in different fonts and colors. (Part 1 of 2.)

```
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class FontJPanel extends JPanel
9  {
10    // display Strings in different fonts and colors
11    @Override
12    public void paintComponent(Graphics g)
13    {
14       super.paintComponent(g);
15
16       // set font to Serif (Times), bold, 12pt and draw a string
17       g.setFont(new Font("Serif", Font.BOLD, 12));
18       g.drawString("Serif 12 point bold.", 20, 30);
19
20       // set font to Monospaced (Courier), italic, 24pt and draw a string
21       g.setFont(new Font("Monospaced", Font.ITALIC, 24));
22       g.drawString("Monospaced 24 point italic.", 20, 50);
23
24       // set font to SansSerif (Helvetica), plain, 14pt and draw a string
25       g.setFont(new Font("SansSerif", Font.PLAIN, 14));
26       g.drawString("SansSerif 14 point plain.", 20, 70);
27
28       // set font to Serif (Times), bold/italic, 18pt and draw a string
29       g.setColor(Color.RED);
30       g.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 18));
31       g.drawString(g.getFont().getName() + " " + g.getFont().getSize() +
32          " point bold italic.", 20, 90);
33    }
34 } // end class FontJPanel
```

Fig. 27.11 | Display strings in different fonts and colors. (Part 2 of 2.)

Figure 27.12 contains the `main` method, which creates a `JFrame` to display a `FontJPanel`. We add a `FontJPanel` object to this `JFrame` (line 15), which displays the graphics created in Fig. 27.11.

```
1  // Fig. 13.12: Fonts.java
2  // Using fonts.
3  import javax.swing.JFrame;
4
5  public class Fonts
6  {
7     // execute application
8     public static void main(String[] args)
9     {
10       // create frame for FontJPanel
11       JFrame frame = new JFrame("Using fonts");
12       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
```

Fig. 27.12 | Using fonts. (Part 1 of 2.)

```
14          FontJPanel fontJPanel = new FontJPanel();
15          frame.add(fontJPanel);
16          frame.setSize(420, 150);
17          frame.setVisible(true);
18       }
19    } // end class Fonts
```

Fig. 27.12 | Using fonts. (Part 2 of 2.)

Font Metrics

Sometimes it's necessary to get information about the current drawing font, such as its name, style and size. Several `Font` methods used to get font information are summarized in Fig. 27.10. Method **`getStyle`** returns an integer value representing the current style. The integer value returned is either `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD` or the combination of `Font.ITALIC` and `Font.BOLD`. Method **`getFamily`** returns the name of the font family to which the current font belongs. The name of the font family is platform specific. `Font` methods are also available to test the style of the current font, and these too are summarized in Fig. 27.10. Methods **`isPlain`**, **`isBold`** and **`isItalic`** return `true` if the current font style is plain, bold or italic, respectively.

Figure 27.13 illustrates some of the common **font metrics**, which provide precise information about a font, 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—that is, the interline spacing).

Fig. 27.13 | Font metrics.

Class **`FontMetrics`** declares several methods for obtaining font metrics. These methods and `Graphics` method **`getFontMetrics`** are summarized in Fig. 27.14. The application of Figs. 27.15–27.16 uses the methods of Fig. 27.14 to obtain font metric information for two fonts.

Method	Description
FontMetrics methods	
`public int getAscent()`	Returns the ascent of a font in points.
`public int getDescent()`	Returns the descent of a font in points.
`public int getLeading()`	Returns the leading of a font in points.
`public int getHeight()`	Returns the height of a font in points.
Graphics methods for getting a Font's FontMetrics	
`public FontMetrics getFontMetrics()`	Returns the `FontMetrics` object for the current drawing `Font`.
`public FontMetrics getFontMetrics(Font f)`	Returns the `FontMetrics` object for the specified `Font` argument.

Fig. 27.14 | `FontMetrics` and `Graphics` methods for obtaining font metrics.

```
1  // Fig. 13.15: MetricsJPanel.java
2  // FontMetrics and Graphics methods useful for obtaining font metrics.
3  import java.awt.Font;
4  import java.awt.FontMetrics;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class MetricsJPanel extends JPanel
9  {
10    // display font metrics
11    @Override
12    public void paintComponent(Graphics g)
13    {
14       super.paintComponent(g);
15
16       g.setFont(new Font("SansSerif", Font.BOLD, 12));
17       FontMetrics metrics = g.getFontMetrics();
18       g.drawString("Current font: " + g.getFont(), 10, 30);
19       g.drawString("Ascent: " + metrics.getAscent(), 10, 45);
20       g.drawString("Descent: " + metrics.getDescent(), 10, 60);
21       g.drawString("Height: " + metrics.getHeight(), 10, 75);
22       g.drawString("Leading: " + metrics.getLeading(), 10, 90);
23
24       Font font = new Font("Serif", Font.ITALIC, 14);
25       metrics = g.getFontMetrics(font);
26       g.setFont(font);
27       g.drawString("Current font: " + font, 10, 120);
28       g.drawString("Ascent: " + metrics.getAscent(), 10, 135);
29       g.drawString("Descent: " + metrics.getDescent(), 10, 150);
30       g.drawString("Height: " + metrics.getHeight(), 10, 165);
31       g.drawString("Leading: " + metrics.getLeading(), 10, 180);
32    }
33 } // end class MetricsJPanel
```

Fig. 27.15 | `FontMetrics` and `Graphics` methods useful for obtaining font metrics.

```
1  // Fig. 13.16: Metrics.java
2  // Displaying font metrics.
3  import javax.swing.JFrame;
4
5  public class Metrics
6  {
7     // execute application
8     public static void main(String[] args)
9     {
10       // create frame for MetricsJPanel
11       JFrame frame = new JFrame("Demonstrating FontMetrics");
12       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       MetricsJPanel metricsJPanel = new MetricsJPanel();
15       frame.add(metricsJPanel);
16       frame.setSize(510, 240);
17       frame.setVisible(true);
18    }
19 } // end class Metrics
```

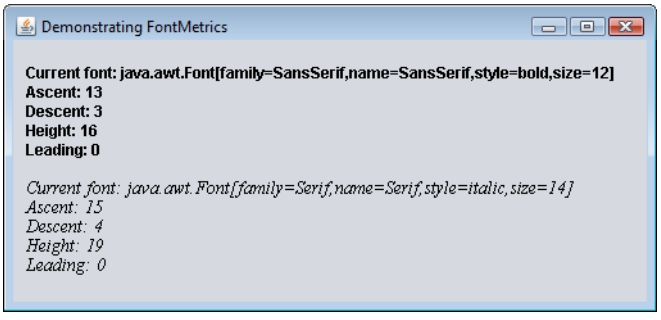


Fig. 27.16 | Displaying font metrics.

Line 16 of Fig. 27.15 creates and sets the current drawing font to a `SansSerif`, bold, 12-point font. Line 17 uses `Graphics` method `getFontMetrics` to obtain the `FontMetrics` object for the current font. Line 18 outputs the `String` representation of the `Font` returned by `g.getFont()`. Lines 19–22 use `FontMetric` methods to obtain the ascent, descent, height and leading for the font.

Line 24 creates a new `Serif`, italic, 14-point font. Line 25 uses a second version of `Graphics` method `getFontMetrics`, which accepts a `Font` argument and returns a corresponding `FontMetrics` object. Lines 28–31 obtain the ascent, descent, height and leading for the font. The font metrics are slightly different for the two fonts.

27.5 Drawing Lines, Rectangles and Ovals

This section presents `Graphics` methods for drawing lines, rectangles and ovals. The methods and their parameters are summarized in Fig. 27.17. 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
`public void drawLine(int x1, int y1, int x2, int y2)`	Draws a line between the point (`x1`, `y1`) and the point (`x2`, `y2`).
`public void drawRect(int x, int y, int width, int height)`	Draws a rectangle of the specified `width` and `height`. The rectangle's top-left corner is located at (`x`, `y`). Only the outline of the rectangle is drawn using the `Graphics` object's color—the body of the rectangle is not filled with this color.
`public void fillRect(int x, int y, int width, int height)`	Draws a filled rectangle in the current color with the specified `width` and `height`. The rectangle's top-left corner is located at (`x`, `y`).
`public void clearRect(int x, int y, int width, int height)`	Draws a filled rectangle with the specified `width` and `height` in the current background color. The rectangle's *top-left* corner is located at (`x`, `y`). This method is useful if you want to remove a portion of an image.
`public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	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. 27.20). Only the outline of the shape is drawn.
`public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	Draws a filled rectangle in the current color with rounded corners with the specified `width` and `height`. The `arcWidth` and `arcHeight` determine the rounding of the corners (see Fig. 27.20).
`public void draw3DRect(int x, int y, int width, int height, boolean b)`	Draws a three-dimensional rectangle in the current color with the specified `width` and `height`. The rectangle's *top-left* corner is located at (`x`, `y`). The rectangle appears raised when `b` is true and lowered when `b` is false. Only the outline of the shape is drawn.
`public void fill3DRect(int x, int y, int width, int height, boolean b)`	Draws a filled three-dimensional rectangle in the current color with the specified `width` and `height`. The rectangle's *top-left* corner is located at (`x`, `y`). The rectangle appears raised when `b` is true and lowered when `b` is false.
`public void drawOval(int x, int y, int width, int height)`	Draws an oval in the current color with the specified `width` and `height`. The bounding rectangle's *top-left* corner is located at (`x`, `y`). The oval touches all four sides of the bounding rectangle at the center of each side (see Fig. 27.21). Only the outline of the shape is drawn.
`public void fillOval(int x, int y, int width, int height)`	Draws a filled oval in the current color with the specified `width` and `height`. The bounding rectangle's *top-left* corner is located at (`x`, `y`). The oval touches the center of all four sides of the bounding rectangle (see Fig. 27.21).

Fig. 27.17 | `Graphics` methods that draw lines, rectangles and ovals.

The application of Figs. 27.18–27.19 demonstrates drawing a variety of lines, rectangles, three-dimensional rectangles, rounded rectangles and ovals. In Fig. 27.18, line 17 draws a red line, line 20 draws an empty blue rectangle and line 21 draws a filled blue rectangle. Methods **`fillRoundRect`** (line 24) and **`drawRoundRect`** (line 25) 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. 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. The last two arguments determine the horizontal and vertical diameters of the arc (i.e., the arc width and arc height) used to represent the corners.

```
1  // Fig. 13.18: LinesRectsOvalsJPanel.java
2  // Drawing lines, rectangles and ovals.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class LinesRectsOvalsJPanel extends JPanel
8  {
9     // display various lines, rectangles and ovals
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14       this.setBackground(Color.WHITE);
15
16       g.setColor(Color.RED);
17       g.drawLine(5, 30, 380, 30);
18
19       g.setColor(Color.BLUE);
20       g.drawRect(5, 40, 90, 55);
21       g.fillRect(100, 40, 90, 55);
22
23       g.setColor(Color.CYAN);
24       g.fillRoundRect(195, 40, 90, 55, 50, 50);
25       g.drawRoundRect(290, 40, 90, 55, 20, 20);
26
27       g.setColor(Color.GREEN);
28       g.draw3DRect(5, 100, 90, 55, true);
29       g.fill3DRect(100, 100, 90, 55, false);
30
31       g.setColor(Color.MAGENTA);
32       g.drawOval(195, 100, 90, 55);
33       g.fillOval(290, 100, 90, 55);
34    }
35 } // end class LinesRectsOvalsJPanel
```

Fig. 27.18 | Drawing lines, rectangles and ovals.

```
1  // Fig. 13.19: LinesRectsOvals.java
2  // Testing LinesRectsOvalsJPanel.
3  import java.awt.Color;
4  import javax.swing.JFrame;
5
6  public class LinesRectsOvals
7  {
8     // execute application
9     public static void main(String[] args)
10    {
11       // create frame for LinesRectsOvalsJPanel
12       JFrame frame =
13          new JFrame("Drawing lines, rectangles and ovals");
14       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15
16       LinesRectsOvalsJPanel linesRectsOvalsJPanel =
17          new LinesRectsOvalsJPanel();
18       linesRectsOvalsJPanel.setBackground(Color.WHITE);
19       frame.add(linesRectsOvalsJPanel);
20       frame.setSize(400, 210);
21       frame.setVisible(true);
22    }
23 } // end class LinesRectsOvals
```

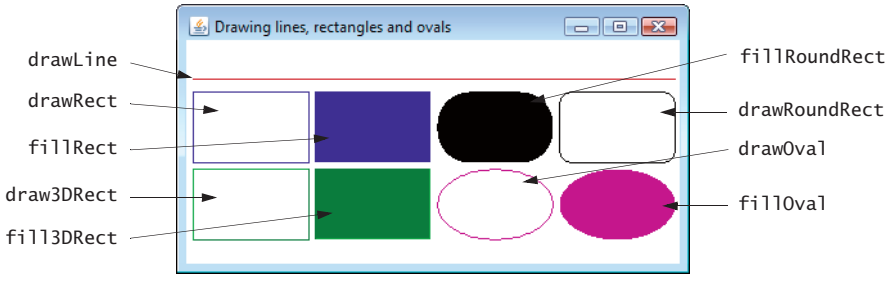


Fig. 27.19 | Testing `LinesRectsOvalsJPanel`.

Figure 27.20 labels the arc width, arc height, width and height of a rounded rectangle. Using the same value for the arc width and arc height produces a quarter-circle at each corner. When the arc width, arc height, width and height 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.

Methods **`draw3DRect`** (Fig. 27.18, line 28) and **`fill3DRect`** (line 29) take the same arguments. The first two 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

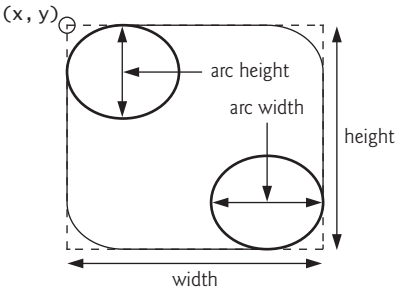


Fig. 27.20 | Arc width and arc height for rounded rectangles.

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.

Methods `drawOval` and `fillOval` (lines 32–33) take the same four arguments. The first two specify the top-left coordinate of the bounding rectangle that contains the oval. The last two specify the width and height of the bounding rectangle, respectively. Figure 27.21 shows an oval bounded by a rectangle. The oval touches the *center* of all four sides of the bounding rectangle. (The bounding rectangle is *not* displayed on the screen.)

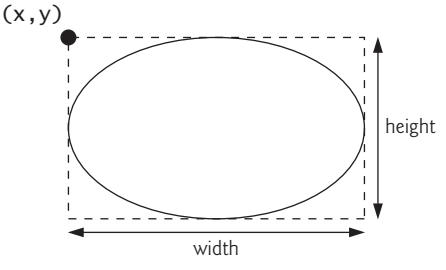


Fig. 27.21 | Oval bounded by a rectangle.

27.6 Drawing Arcs

An **arc** is drawn as a portion of an oval. Arc angles are measured in degrees. Arcs **sweep** (i.e., move along a curve) from a **starting angle** through 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 27.22 illustrates two arcs. The left set of axes shows an arc sweeping from zero degrees to approximately 110 degrees. Arcs that sweep in a *counterclockwise* direction are measured in **positive degrees**. The set of axes on the right shows an arc sweeping from zero degrees to approximately –110 degrees. Arcs that sweep in a *clockwise* direction are measured in **negative degrees**. Note the dashed boxes around the arcs in Fig. 27.22. When drawing an arc,

we specify a bounding rectangle for an oval. The arc will sweep along part of the oval. `Graphics` methods **`drawArc`** and **`fillArc`** for drawing arcs are summarized in Fig. 27.23.

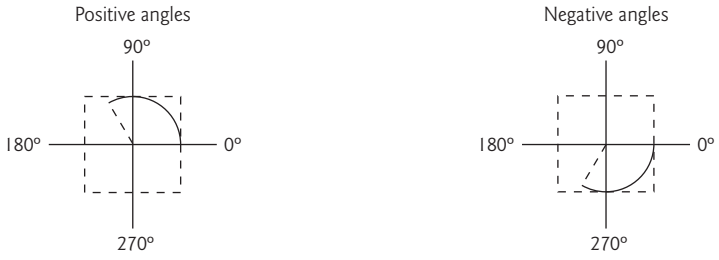


Fig. 27.22 | Positive and negative arc angles.

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

Fig. 27.23 | `Graphics` methods for drawing arcs.

Figures 27.24–27.25 demonstrate the arc methods of Fig. 27.23. The application 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 red rectangle that has the same `x`, `y`, `width` and `height` arguments as the arcs.

```
1  // Fig. 13.24: ArcsJPanel.java
2  // Arcs displayed with drawArc and fillArc.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class ArcsJPanel extends JPanel
8  {
```

Fig. 27.24 | Arcs displayed with `drawArc` and `fillArc`. (Part 1 of 2.)

```
9       // draw rectangles and arcs
10      @Override
11      public void paintComponent(Graphics g)
12      {
13         super.paintComponent(g);
14
15         // start at 0 and sweep 360 degrees
16         g.setColor(Color.RED);
17         g.drawRect(15, 35, 80, 80);
18         g.setColor(Color.BLACK);
19         g.drawArc(15, 35, 80, 80, 0, 360);
20
21         // start at 0 and sweep 110 degrees
22         g.setColor(Color.RED);
23         g.drawRect(100, 35, 80, 80);
24         g.setColor(Color.BLACK);
25         g.drawArc(100, 35, 80, 80, 0, 110);
26
27         // start at 0 and sweep -270 degrees
28         g.setColor(Color.RED);
29         g.drawRect(185, 35, 80, 80);
30         g.setColor(Color.BLACK);
31         g.drawArc(185, 35, 80, 80, 0, -270);
32
33         // start at 0 and sweep 360 degrees
34         g.fillArc(15, 120, 80, 40, 0, 360);
35
36         // start at 270 and sweep -90 degrees
37         g.fillArc(100, 120, 80, 40, 270, -90);
38
39         // start at 0 and sweep -270 degrees
40         g.fillArc(185, 120, 80, 40, 0, -270);
41      }
42   } // end class ArcsJPanel
```

Fig. 27.24 | Arcs displayed with `drawArc` and `fillArc`. (Part 2 of 2.)

```
1   // Fig. 13.25: DrawArcs.java
2   // Drawing arcs.
3   import javax.swing.JFrame;
4
5   public class DrawArcs
6   {
7      // execute application
8      public static void main(String[] args)
9      {
10        // create frame for ArcsJPanel
11        JFrame frame = new JFrame("Drawing Arcs");
12        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
```

Fig. 27.25 | Drawing arcs. (Part 1 of 2.)

```
14          ArcsJPanel arcsJPanel = new ArcsJPanel();
15          frame.add(arcsJPanel);
16          frame.setSize(300, 210);
17          frame.setVisible(true);
18       }
19    } // end class DrawArcs
```

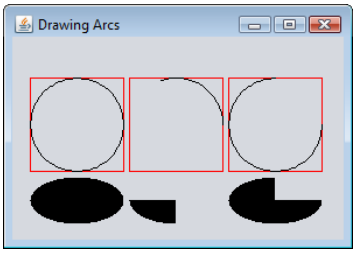


Fig. 27.25 | Drawing arcs. (Part 2 of 2.)

27.7 Drawing Polygons and Polylines

Polygons are *closed multisided shapes* composed of straight-line segments. **Polylines** are *sequences of connected points*. Figure 27.26 discusses methods for drawing polygons and polylines. Some methods require a **`Polygon`** object (package `java.awt`). Class `Polygon`'s constructors are also described in Fig. 27.26. The application of Figs. 27.27–27.28 draws polygons and polylines.

Method	Description
`Graphics` methods for drawing polygons	
`public void drawPolygon(int[] xPoints, int[] yPoints, int points)`	Draws a polygon. The *x*-coordinate of each point is specified in the `xPoints` array and the *y*-coordinate of each point in the `yPoints` array. The last argument specifies the number of `points`. This method draws a *closed polygon*. If the last point is different from the first, the polygon is *closed* by a line that connects the last point to the first.
`public void drawPolyline(int[] xPoints, int[] yPoints, int points)`	Draws a sequence of connected lines. The *x*-coordinate of each point is specified in the `xPoints` array and the *y*-coordinate of each point in the `yPoints` array. The last argument specifies the number of `points`. If the last point is different from the first, the polyline is *not* closed.
`public void drawPolygon(Polygon p)`	Draws the specified polygon.

Fig. 27.26 | `Graphics` methods for polygons and class `Polygon` methods. (Part 1 of 2.)

Method	Description
`public void fillPolygon(int[] xPoints, int[] yPoints, int points)`	Draws a *filled* polygon. The *x*-coordinate of each point is specified in the `xPoints` array and the *y*-coordinate of each point in the `yPoints` array. The last argument specifies the number of `points`. This method draws a *closed polygon*. If the last point is different from the first, the polygon is *closed* by a line that connects the last point to the first.
`public void fillPolygon(Polygon p)`	Draws the specified *filled* polygon. The polygon is *closed*.
`Polygon` constructors and methods	
`public Polygon()`	Constructs a new polygon object. The polygon does not contain any points.
`public Polygon(int[] xValues, int[] yValues, int numberOfPoints)`	Constructs a new polygon object. The polygon has `numberOfPoints` sides, with each point consisting of an *x*-coordinate from `xValues` and a *y*-coordinate from `yValues`.
`public void addPoint(int x, int y)`	Adds pairs of *x*- and *y*-coordinates to the `Polygon`.

Fig. 27.26 | `Graphics` methods for polygons and class `Polygon` methods. (Part 2 of 2.)

```
1  // Fig. 13.27: PolygonsJPanel.java
2  // Drawing polygons.
3  import java.awt.Graphics;
4  import java.awt.Polygon;
5  import javax.swing.JPanel;
6
7  public class PolygonsJPanel extends JPanel
8  {
9     // draw polygons and polylines
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14
15       // draw polygon with Polygon object
16       int[] xValues = {20, 40, 50, 30, 20, 15};
17       int[] yValues = {50, 50, 60, 80, 80, 60};
18       Polygon polygon1 = new Polygon(xValues, yValues, 6);
19       g.drawPolygon(polygon1);
20
21       // draw polylines with two arrays
22       int[] xValues2 = {70, 90, 100, 80, 70, 65, 60};
23       int[] yValues2 = {100, 100, 110, 110, 130, 110, 90};
24       g.drawPolyline(xValues2, yValues2, 7);
```

Fig. 27.27 | Polygons displayed with `drawPolygon` and `fillPolygon`. (Part 1 of 2.)

```
25
26          // fill polygon with two arrays
27          int[] xValues3 = {120, 140, 150, 190};
28          int[] yValues3 = {40, 70, 80, 60};
29          g.fillPolygon(xValues3, yValues3, 4);
30
31          // draw filled polygon with Polygon object
32          Polygon polygon2 = new Polygon();
33          polygon2.addPoint(165, 135);
34          polygon2.addPoint(175, 150);
35          polygon2.addPoint(270, 200);
36          polygon2.addPoint(200, 220);
37          polygon2.addPoint(130, 180);
38          g.fillPolygon(polygon2);
39       }
40    } // end class PolygonsJPanel
```

Fig. 27.27 | Polygons displayed with `drawPolygon` and `fillPolygon`. (Part 2 of 2.)

Lines 16–17 of Fig. 27.27 create two `int` arrays and use them to specify the points for `Polygon polygon1`. The `Polygon` constructor call in line 18 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 19 displays `polygon1` by passing it as an argument to `Graphics` method **`drawPolygon`**.

Lines 22–23 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` the *y*-coordinate of each point. Line 24 uses `Graphics` method **`drawPolyline`** to display the series of connected lines specified with the arguments `xValues2`, `yValues2` and `7` (the number of points).

Lines 27–28 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` the *y*-coordinate of each point. Line 29 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 27.1

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 specify the polygon to display.

Line 32 creates `Polygon polygon2` with no points. Lines 33–37 use `Polygon` method **`addPoint`** to add pairs of *x*- and *y*-coordinates to the `Polygon`. Line 38 displays `Polygon polygon2` by passing it to `Graphics` method `fillPolygon`.

```
1  // Fig. 13.28: DrawPolygons.java
2  // Drawing polygons.
3  import javax.swing.JFrame;
4
```

Fig. 27.28 | Drawing polygons. (Part 1 of 2.)

```
5  public class DrawPolygons
6  {
7     // execute application
8     public static void main(String[] args)
9     {
10       // create frame for PolygonsJPanel
11       JFrame frame = new JFrame("Drawing Polygons");
12       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       PolygonsJPanel polygonsJPanel = new PolygonsJPanel();
15       frame.add(polygonsJPanel);
16       frame.setSize(280, 270);
17       frame.setVisible(true);
18    }
19 } // end class DrawPolygons
```

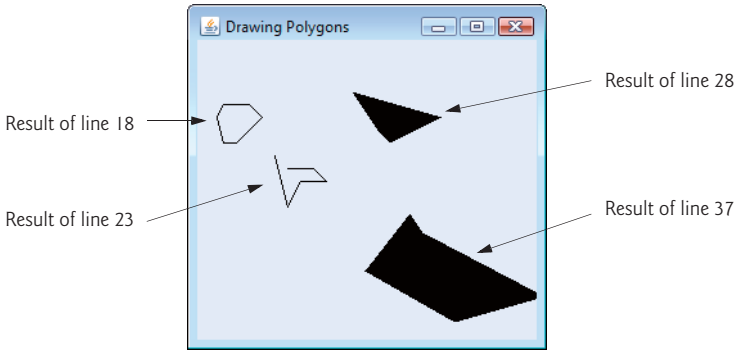


Fig. 27.28 | Drawing polygons. (Part 2 of 2.)

27.8 Java 2D API

The **Java 2D 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, visit `http://docs.oracle.com/javase/8/docs/technotes/guides/2d/`. In this section, we overview several Java 2D capabilities.

Drawing with the Java 2D API is accomplished with a **`Graphics2D`** reference (package `java.awt`). `Graphics2D` is an *abstract subclass* of class `Graphics`, so it has all the graphics capabilities demonstrated earlier in this chapter. In fact, the actual object used to draw in every `paintComponent` method is an instance of a *subclass* of `Graphics2D` that is passed to method `paintComponent` and accessed via the *superclass* `Graphics`. To access `Graphics2D` capabilities, we must cast the `Graphics` reference (`g`) passed to `paintComponent` into a `Graphics2D` reference with a statement such as

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

The next two examples use this technique.

Lines, Rectangles, Round Rectangles, Arcs and Ellipses

This example demonstrates several Java 2D shapes from package `java.awt.geom`, including **`Line2D.Double`**, **`Rectangle2D.Double`**, **`RoundRectangle2D.Double`**, **`Arc2D.Double`** and **`Ellipse2D.Double`**. Note the syntax of each class name. Each class represents a shape with dimensions specified as `double` values. There's a *separate* version of each represented with `float` values (e.g., **`Ellipse2D.Float`**). In each case, `Double` is a `public static` nested class of the class specified to the left of the dot (e.g., `Ellipse2D`). To use the `static` nested class, we simply qualify its name with the outer-class name.

In Figs. 27.29–27.30, we draw Java 2D shapes and modify their drawing characteristics, such as changing line thickness, filling shapes with patterns and drawing dashed lines. These are just a few of the many capabilities provided by Java 2D. Line 25 of Fig. 27.29 casts the `Graphics` reference received by `paintComponent` to a `Graphics2D` reference and assigns it to `g2d` to allow access to the Java 2D features.

```
1  // Fig. 13.29: ShapesJPanel.java
2  // Demonstrating some Java 2D shapes.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.BasicStroke;
6  import java.awt.GradientPaint;
7  import java.awt.TexturePaint;
8  import java.awt.Rectangle;
9  import java.awt.Graphics2D;
10 import java.awt.geom.Ellipse2D;
11 import java.awt.geom.Rectangle2D;
12 import java.awt.geom.RoundRectangle2D;
13 import java.awt.geom.Arc2D;
14 import java.awt.geom.Line2D;
15 import java.awt.image.BufferedImage;
16 import javax.swing.JPanel;
17
18 public class ShapesJPanel extends JPanel
19 {
20    // draw shapes with Java 2D API
21    @Override
22    public void paintComponent(Graphics g)
23    {
24       super.paintComponent(g);
25       Graphics2D g2d = (Graphics2D) g; // cast g to Graphics2D
26
27       // draw 2D ellipse filled with a blue-yellow gradient
28       g2d.setPaint(new GradientPaint(5, 30, Color.BLUE, 35, 100,
29          Color.YELLOW, true));
30       g2d.fill(new Ellipse2D.Double(5, 30, 65, 100));
31
32       // draw 2D rectangle in red
33       g2d.setPaint(Color.RED);
34       g2d.setStroke(new BasicStroke(10.0f));
35       g2d.draw(new Rectangle2D.Double(80, 30, 65, 100));
36
```

Fig. 27.29 | Demonstrating some Java 2D shapes. (Part 1 of 2.)

```
37          // draw 2D rounded rectangle with a buffered background
38          BufferedImage buffImage = new BufferedImage(10, 10,
39             BufferedImage.TYPE_INT_RGB);
40
41          // obtain Graphics2D from buffImage and draw on it
42          Graphics2D gg = buffImage.createGraphics();
43          gg.setColor(Color.YELLOW);
44          gg.fillRect(0, 0, 10, 10);
45          gg.setColor(Color.BLACK);
46          gg.drawRect(1, 1, 6, 6);
47          gg.setColor(Color.BLUE);
48          gg.fillRect(1, 1, 3, 3);
49          gg.setColor(Color.RED);
50          gg.fillRect(4, 4, 3, 3); // draw a filled rectangle
51
52          // paint buffImage onto the JFrame
53          g2d.setPaint(new TexturePaint(buffImage,
54             new Rectangle(10, 10)));
55          g2d.fill(
56             new RoundRectangle2D.Double(155, 30, 75, 100, 50, 50));
57
58          // draw 2D pie-shaped arc in white
59          g2d.setPaint(Color.WHITE);
60          g2d.setStroke(new BasicStroke(6.0f));
61          g2d.draw(
62             new Arc2D.Double(240, 30, 75, 100, 0, 270, Arc2D.PIE));
63
64          // draw 2D lines in green and yellow
65          g2d.setPaint(Color.GREEN);
66          g2d.draw(new Line2D.Double(395, 30, 320, 150));
67
68          // draw 2D line using stroke
69          float[] dashes = {10}; // specify dash pattern
70          g2d.setPaint(Color.YELLOW);
71          g2d.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND,
72             BasicStroke.JOIN_ROUND, 10, dashes, 0));
73          g2d.draw(new Line2D.Double(320, 30, 395, 150));
74       }
75    } // end class ShapesJPanel
```

Fig. 27.29 | Demonstrating some Java 2D shapes. (Part 2 of 2.)

```
1  // Fig. 13.30: Shapes.java
2  // Testing ShapesJPanel.
3  import javax.swing.JFrame;
4
5  public class Shapes
6  {
7     // execute application
8     public static void main(String[] args)
9     {
```

Fig. 27.30 | Testing `ShapesJPanel`. (Part 1 of 2.)

```
10         // create frame for ShapesJPanel
11         JFrame frame = new JFrame("Drawing 2D shapes");
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14         // create ShapesJPanel
15         ShapesJPanel shapesJPanel = new ShapesJPanel();
16
17         frame.add(shapesJPanel);
18         frame.setSize(425, 200);
19         frame.setVisible(true);
20      }
21   } // end class Shapes
```

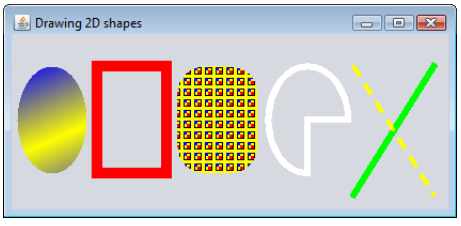


Fig. 27.30 | Testing `ShapesJPanel`. (Part 2 of 2.)

Ovals, Gradient Fills and `Paint` *Objects*

The first shape we draw is an *oval filled with gradually changing colors*. Lines 28–29 invoke `Graphics2D` method **`setPaint`** to set the **`Paint`** object that determines the color for the shape to display. A `Paint` object implements interface `java.awt.Paint`. It can be something as simple as one of the predeclared `Color` objects introduced in Section 27.3 (class `Color` implements `Paint`), or it can be an instance of the Java 2D API's `GradientPaint`, **`SystemColor`**, `TexturePaint`, `LinearGradientPaint` or `RadialGradientPaint` classes. In this case, we use a `GradientPaint` object.

Class `GradientPaint` helps draw a shape in *gradually changing colors*—called a **gradient**. The `GradientPaint` constructor used here requires seven arguments. The first two specify the starting coordinates for the gradient. The third specifies the starting `Color` for the gradient. The fourth and fifth specify the ending coordinates for the gradient. The sixth specifies the ending `Color` for the gradient. The last argument specifies whether the gradient is **cyclic** (`true`) or **acyclic** (`false`). The two sets of 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 30 uses `Graphics2D` method **`fill`** to draw a filled **`Shape`** object—an object that implements interface `Shape` (package `java.awt`). In this case, we display an `Ellipse2D.Double` object. The `Ellipse2D.Double` constructor receives four arguments specifying the *bounding rectangle* for the ellipse to display.

Rectangles, `Stroke`s

Next we draw a red rectangle with a thick border. Line 33 invokes `setPaint` to set the `Paint` object to `Color.RED`. Line 34 uses `Graphics2D` method **`setStroke`** to set the characteristics of the rectangle's border (or the lines for any other shape). Method `setStroke` requires as its argument an object that implements interface **`Stroke`** (package `java.awt`). In this case, we use an instance of class `BasicStroke`. Class `BasicStroke` provides several 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's a dashed line). The constructor here specifies that the line should be 10 pixels wide.

Line 35 uses `Graphics2D` method **`draw`** to draw a `Shape` object—in this case, a `Rectangle2D.Double`. The `Rectangle2D.Double` constructor receives arguments specifying the rectangle's *upper-left x*-coordinate, upper-left *y*-coordinate, width and height.

Rounded Rectangles, `BufferedImage`s and `TexturePaint` Objects

Next we draw a rounded rectangle filled with a pattern created in a **`BufferedImage`** (package `java.awt.image`) object. Lines 38–39 create the `BufferedImage` object. Class `BufferedImage` can be used to produce images in color and grayscale. This particular `BufferedImage` is 10 pixels wide and 10 pixels tall (as specified by the first two arguments of the constructor). The third argument **`BufferedImage.TYPE_INT_RGB`** indicates that the image is stored in color using the RGB color scheme.

To create the rounded rectangle's fill pattern, we must first draw into the `BufferedImage`. Line 42 creates a `Graphics2D` object (by calling `BufferedImage` method **`createGraphics`**) that can be used to draw into the `BufferedImage`. Lines 43–50 use methods `setColor`, `fillRect` and `drawRect` to create the pattern.

Lines 53–54 set the `Paint` object to a new `TexturePaint` (package `java.awt`) object. A `TexturePaint` object uses the image stored in its associated `BufferedImage` (the first constructor argument) 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 55–56 use `Graphics2D` method `fill` to draw a filled `Shape` object—in this case, a `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.

Arcs

Next we draw a pie-shaped arc with a thick white line. Line 59 sets the `Paint` object to `Color.WHITE`. Line 60 sets the `Stroke` object to a new `BasicStroke` for a line 6 pixels wide. Lines 61–62 use `Graphics2D` method `draw` to draw a `Shape` object—in this case, an `Arc2D.Double`. 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 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 should *not* be *closed*.

Lines

Finally, we draw two lines using **`Line2D`** objects—one solid and one dashed. Line 65 sets the `Paint` object to `Color.GREEN`. Line 66 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 the starting coordinates and ending coordinates of the line.

Line 69 declares a one-element `float` array containing the value 10. This array describes 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 length of each dash as an element in the array. Line 70 sets the `Paint` object to `Color.YELLOW`. Lines 71–72 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), their joining will be rounded (**`BasicStroke.JOIN_ROUND`**). The `dashes` argument specifies the dash lengths for the line. The last argument indicates the starting index in the `dashes` array for the first dash in the pattern. Line 73 then draws a line with the current `Stroke`.

Creating Your Own Shapes with General Paths

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 application of Figs. 27.31 and 27.32 demonstrates drawing a general path in the shape of a five-pointed star.

```
1  // Fig. 13.31: Shapes2JPanel.java
2  // Demonstrating a general path.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.geom.GeneralPath;
7  import java.security.SecureRandom;
8  import javax.swing.JPanel;
9
10 public class Shapes2JPanel extends JPanel
11 {
12    // draw general paths
13    @Override
14    public void paintComponent(Graphics g)
15    {
16       super.paintComponent(g);
17       SecureRandom random = new SecureRandom();
18
19       int[] xPoints = {55, 67, 109, 73, 83, 55, 27, 37, 1, 43};
20       int[] yPoints = {0, 36, 36, 54, 96, 72, 96, 54, 36, 36};
21
22       Graphics2D g2d = (Graphics2D) g;
23       GeneralPath star = new GeneralPath();
24
```

Fig. 27.31 | Java 2D general paths. (Part 1 of 2.)

```
25          // set the initial coordinate of the General Path
26          star.moveTo(xPoints[0], yPoints[0]);
27
28          // create the star--this does not draw the star
29          for (int count = 1; count < xPoints.length; count++)
30             star.lineTo(xPoints[count], yPoints[count]);
31
32          star.closePath(); // close the shape
33
34          g2d.translate(150, 150); // translate the origin to (150, 150)
35
36          // rotate around origin and draw stars in random colors
37          for (int count = 1; count <= 20; count++)
38          {
39             g2d.rotate(Math.PI / 10.0); // rotate coordinate system
40
41             // set random drawing color
42             g2d.setColor(new Color(random.nextInt(256),
43                random.nextInt(256), random.nextInt(256)));
44
45             g2d.fill(star); // draw filled star
46          }
47       }
48    } // end class Shapes2JPanel
```

Fig. 27.31 | Java 2D general paths. (Part 2 of 2.)

```
1  // Fig. 13.32: Shapes2.java
2  // Demonstrating a general path.
3  import java.awt.Color;
4  import javax.swing.JFrame;
5
6  public class Shapes2
7  {
8     // execute application
9     public static void main(String[] args)
10    {
11       // create frame for Shapes2JPanel
12       JFrame frame = new JFrame("Drawing 2D Shapes");
13       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14
15       Shapes2JPanel shapes2JPanel = new Shapes2JPanel();
16       frame.add(shapes2JPanel);
17       frame.setBackground(Color.WHITE);
18       frame.setSize(315, 330);
19       frame.setVisible(true);
20    }
21 } // end class Shapes2
```

Fig. 27.32 | Demonstrating a general path. (Part 1 of 2.)

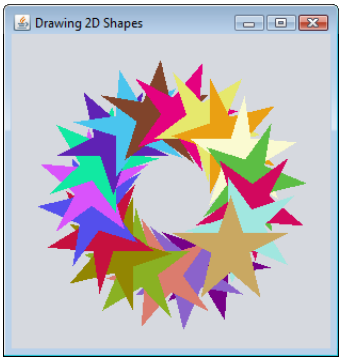


Fig. 27.32 | Demonstrating a general path. (Part 2 of 2.)

Lines 19–20 (Fig. 27.31) declare two `int` arrays representing the *x*- and *y*-coordinates of the points in the star. Line 23 creates `GeneralPath` object `star`. Line 26 uses `GeneralPath` method **`moveTo`** to specify the first point in the `star`. The `for` statement in lines 29–30 uses `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 32 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 34 uses `Graphics2D` method **`translate`** to move the drawing origin to location (150, 150). All drawing operations now use location (150, 150) as (0, 0).

The `for` statement in lines 37–46 draws the `star` 20 times by rotating it around the new origin point. Line 39 uses `Graphics2D` method **`rotate`** to rotate the next displayed shape. The argument specifies the rotation angle in radians (with 360° = 2π radians). Line 45 uses `Graphics2D` method `fill` to draw a filled version of the `star`.

27.9 Wrap-Up

In this chapter, you learned how to use Java's graphics capabilities to produce colorful drawings. You learned how to specify the location of an object using Java's coordinate system, and how to draw on a window using the `paintComponent` method. You were introduced to class `Color`, and you learned how to use this class to specify different colors using their RGB components. You used the `JColorChooser` dialog to allow users to select colors in a program. You then learned how to work with fonts when drawing text on a window. You learned how to create a `Font` object from a font name, style and size, as well as how to access the metrics of a font. From there, you learned how to draw various shapes on a window, such as rectangles (regular, rounded and 3D), ovals and polygons, as well as lines and arcs. You then used the Java 2D API to create more complex shapes and to fill them with gradients or patterns. The chapter concluded with a discussion of general paths, used to construct shapes from straight lines and complex curves.

Summary

Section 27.1 Introduction

- Java's coordinate system (p. 2) is a scheme for identifying every point (p. 13) on the screen.
- A coordinate pair (p. 2) has an *x*-coordinate (horizontal) and a *y*-coordinate (vertical).
- Coordinates are used to indicate where graphics should be displayed on a screen.
- Coordinate units are measured in pixels (p. 2). A pixel is a display monitor's smallest unit of resolution.

Section 27.2 Graphics Contexts and Graphics Objects

- A Java graphics context (p. 4) enables drawing on the screen.
- Class `Graphics` (p. 4) contains methods for drawing strings, lines, rectangles and other shapes. Methods are also included for font manipulation and color manipulation.
- A `Graphics` object manages a graphics context and draws pixels on the screen that represent text and other graphical objects, e.g., lines, ellipses, rectangles and other polygons (p. 4).
- Class `Graphics` is an `abstract` class. Each Java implementation has a `Graphics` subclass that provides drawing capabilities. This implementation is hidden from us by class `Graphics`, which supplies the interface that enables us to use graphics in a platform-independent manner.
- Method `paintComponent` can be used to draw graphics in any `JComponent` component.
- Method `paintComponent` receives a `Graphics` object that is passed to the method by the system when a lightweight Swing component needs to be repainted.
- When an application executes, the application container calls method `paintComponent`. For `paintComponent` to be called again, an event must occur.
- When a `JComponent` is displayed, its `paintComponent` method is called.
- Calling method `repaint` (p. 5) on a component updates the graphics drawn on that component.

Section 27.3 Color Control

- Class `Color` (p. 5) declares methods and constants for manipulating colors in a Java program.
- Every color is created from a red, a green and a blue component. Together these components are called RGB values (p. 6). The RGB components specify the amount of red, green and blue in a color, respectively. The larger the value, the greater the amount of that particular color.
- `Color` methods `getRed`, `getGreen` and `getBlue` (p. 6) return `int` values from 0 to 255 representing the amount of red, green and blue, respectively.
- `Graphics` method `getColor` (p. 6) returns a `Color` object with the current drawing color.
- `Graphics` method `setColor` (p. 6) sets the current drawing color.
- `Graphics` method `fillRect` (p. 6) draws a rectangle filled by the `Graphics` object's current color.
- `Graphics` method `drawString` (p. 6) draws a `String` in the current color.
- The `JColorChooser` GUI component (p. 8) enables application users to select colors.
- `JColorChooser` `static` method `showDialog` (p. 10) displays a modal `JColorChooser` dialog.

Section 27.4 Manipulating Fonts

- Class `Font` (p. 12) contains methods and constants for manipulating fonts.
- Class `Font`'s constructor takes three arguments—the font name (p. 13), font style and font size.
- A `Font`'s font style can be `Font.PLAIN`, `Font.ITALIC` or `Font.BOLD` (each is a `static` field 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` (p. 13) sets the drawing font in which text will be displayed.
- `Font` method `getSize` (p. 13) returns the font size in points.
- `Font` method `getName` (p. 13) returns the current font name as a string.
- `Font` method `getStyle` (p. 15) returns an integer value representing the current `Font`'s style.
- `Font` method `getFamily` (p. 15) returns the name of the font family to which the current font belongs. The name of the font family is platform specific.
- Class `FontMetrics` (p. 2) contains methods for obtaining font information.
- Font metrics (p. 15) include height, descent and leading.

Section 27.5 Drawing Lines, Rectangles and Ovals

- `Graphics` methods `fillRoundRect` (p. 19) and `drawRoundRect` (p. 19) draw rectangles with rounded corners.
- `Graphics` methods `draw3DRect` (p. 20) and `fill3DRect` (p. 20) draw three-dimensional rectangles.
- `Graphics` methods `drawOval` (p. 21) and `fillOval` (p. 21) draw ovals.

Section 27.6 Drawing Arcs

- An arc (p. 21) is drawn as a portion of an oval.
- Arcs sweep from a starting angle by the number of degrees specified by their arc angle (p. 21).
- `Graphics` methods `drawArc` (p. 22) and `fillArc` (p. 22) are used for drawing arcs.

Section 27.7 Drawing Polygons and Polylines

- Class `Polygon` contains methods for creating polygons.
- Polygons are closed multisided shapes composed of straight-line segments.
- Polylines (p. 24) are sequences of connected points.
- `Graphics` method `drawPolyline` (p. 26) displays a series of connected lines.
- `Graphics` methods `drawPolygon` (p. 26) and `fillPolygon` (p. 26) are used to draw polygons.
- `Polygon` method `addPoint` (p. 26) adds pairs of *x*- and *y*-coordinates to the `Polygon`.

Section 27.8 Java 2D API

- The Java 2D API (p. 27) provides advanced two-dimensional graphics capabilities.
- Class `Graphics2D` (p. 27)—a subclass of `Graphics`—is used for drawing with the Java 2D API.
- The Java 2D API's classes for drawing shapes include `Line2D.Double`, `Rectangle2D.Double`, `RoundRectangle2D.Double`, `Arc2D.Double` and `Ellipse2D.Double` (p. 28).
- Class `GradientPaint` (p. 30) helps draw a shape in gradually changing colors—called a gradient (p. 30).
- `Graphics2D` method `fill` (p. 30) draws a filled object of any type that implements interface `Shape` (p. 30).
- Class `BasicStroke` (p. 30) helps specify the drawing characteristics of lines.
- `Graphics2D` method `draw` (p. 31) is used to draw a `Shape` object.
- Classes `GradientPaint` (p. 31) and `TexturePaint` (p. 31) help specify the characteristics for filling shapes with colors or patterns.
- A general path (p. 32) is a shape constructed from straight lines and complex curves and is represented with an object of class `GeneralPath` (p. 32).

- `GeneralPath` method `moveTo` (p. 34) specifies the first point in a general path.
- `GeneralPath` method `lineTo` (p. 34) draws a line to the next point in the path. Each new call to `lineTo` draws a line from the previous point to the current point.
- `GeneralPath` method `closePath` (p. 34) draws a line from the last point to the point specified in the last call to `moveTo`. This completes the general path.
- `Graphics2D` method `translate` (p. 34) is used to move the drawing origin to a new location.
- `Graphics2D` method `rotate` (p. 34) is used to rotate the next displayed shape.

Self-Review Exercises

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

a) In Java 2D, method _____ of class _____ sets the characteristics of a stroke used to draw a shape.
b) Class _____ helps specify the fill for a shape such that the fill gradually changes from one color to another.
c) The _____ method of class `Graphics` draws a line between two points.
d) RGB is short for _____, _____ and _____.
e) Font sizes are measured in units called _____.
f) Class _____ helps specify the fill for a shape using a pattern drawn in a `BufferedImage`.

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

a) The first two arguments of `Graphics` method `drawOval` specify the center coordinate of the oval.
b) In the Java coordinate system, *x*-coordinates increase from left to right and *y*-coordinates from top to bottom.
c) `Graphics` method `fillPolygon` draws a filled polygon in the current color.
d) `Graphics` method `drawArc` allows negative angles.
e) `Graphics` method `getSize` returns the size of the current font in centimeters.
f) Pixel coordinate (0, 0) is located at the exact center of the monitor.

27.3 Find the error(s) in each of the following and explain how to correct them. 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(255, 255, 0);  // change color to yellow
```

Answers to Self-Review Exercises

27.1 a) `setStroke`, `Graphics2D`. b) `GradientPaint`. c) `drawLine`. d) red, green, blue. e) points. f) `TexturePaint`.

27.2 Answers for a) through f):

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.

27.3 Answers for a) through d):
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) Method `setColor` takes a `Color` object as an argument, not three integers.

Exercises

27.4 Fill in the blanks in each of the following statements:
a) Class _____ of the Java 2D API is used to draw 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 Java 2D shapes.

27.5 State whether each of the following is *true* or *false*. If *false*, explain why.
a) `Graphics` method `drawPolygon` automatically connects the endpoints of the polygon.
b) `Graphics` method `drawLine` draws a line between two points.
c) `Graphics` method `fillArc` uses degrees to specify the angle.
d) In the Java coordinate system, values on the *y*-axis increase from left to right.
e) `Graphics` inherits directly from class `Object`.
f) `Graphics` is an `abstract` class.
g) The `Font` class inherits directly from class `Graphics`.

27.6 *(Concentric Circles Using Method `drawArc`)* Write an application that draws a series of eight concentric circles. The circles should be separated by 10 pixels. Use `Graphics` method `drawArc`.

27.7 *(Concentric Circles Using Class `Ellipse2D.Double`)* Modify your solution to Exercise 27.6 to draw the ovals by using class `Ellipse2D.Double` and method `draw` of class `Graphics2D`.

27.8 *(Random Lines Using Class `Line2D.Double`)* Modify your solution to Exercise 27.7 to draw random lines in random colors and random thicknesses. Use class `Line2D.Double` and method `draw` of class `Graphics2D` to draw the lines.

27.9 *(Random Triangles)* Write an application 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.

27.10 *(Random Characters)* Write an application that randomly draws characters in different fonts, sizes and colors.

27.11 *(Grid Using Method `drawLine`)* Write an application that draws an 8-by-8 grid. Use `Graphics` method `drawLine`.

27.12 *(Grid Using Class `Line2D.Double`)* Modify your solution to Exercise 27.11 to draw the grid using instances of class `Line2D.Double` and method `draw` of class `Graphics2D`.

27.13 *(Grid Using Method `drawRect`)* Write an application that draws a 10-by-10 grid. Use the `Graphics` method `drawRect`.

27.14 *(Grid Using Class `Rectangle2D.Double`)* Modify your solution to Exercise 27.13 to draw the grid by using class `Rectangle2D.Double` and method `draw` of class `Graphics2D`.

27.15 *(Drawing Tetrahedrons)* Write an application that draws a tetrahedron (a three-dimensional shape with four triangular faces). Use class `GeneralPath` and method `draw` of class `Graphics2D`.

27.16 *(Drawing Cubes)* Write an application that draws a cube. Use class `GeneralPath` and method `draw` of class `Graphics2D`.

27.17 *(Circles Using Class `Ellipse2D.Double`)* Write an application that asks the user to input the radius of a circle as a floating-point number and draws the circle, as well as the values of the circle's diameter, circumference and area. Use the value 3.14159 for π. [*Note:* You may also use the predefined constant `Math.PI` for the value of π. This constant is more precise than the value 3.14159. Class `Math` is declared in the `java.lang` package, so you need not `import` it.] Use the following formulas (r is the radius):

$$diameter = 2r$$
$$circumference = 2\pi r$$
$$area = \pi r^2$$

The user should also be prompted for a set of coordinates in addition to the radius. Then draw the circle and display its diameter, circumference and area, using an `Ellipse2D.Double` object to represent the circle and method `draw` of class `Graphics2D` to display it.

27.18 *(Screen Saver)* 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 `paintComponent`. Do you notice any problems with this on your system?

27.19 *(Screen Saver Using `Timer`)* 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 27.18 to remove the call to `repaint` from method `paintComponent`. Declare your class to implement `ActionListener`. (The `actionPerformed` method should simply call `repaint`.) Declare 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).

27.20 *(Screen Saver for a Random Number of Lines)* Modify your solution to Exercise 27.19 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 to perform event handling for the `JTextField`.

27.21 *(Screen Saver with Shapes)* Modify your solution to Exercise 27.20 such that it uses random-number generation to choose different shapes to display. Use methods of class `Graphics`.

27.22 *(Screen Saver Using the Java 2D API)* Modify your solution to Exercise 27.21 to use classes and drawing capabilities of the Java 2D API. Draw shapes like rectangles and ellipses, with randomly generated gradients. Use class `GradientPaint` to generate the gradient.

27.23 *(Turtle Graphics)* Modify your solution to Exercise 7.21—*Turtle Graphics*—to add a graphical user interface using `JTextFields` and `JButtons`. Draw lines rather than 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 Java 2D API features.

27.24 *(Knight's Tour)* Produce a graphical version of the Knight's Tour problem (Exercise 7.22, Exercise 7.23 and Exercise 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 like, use class `Timer` (see Exercise 27.19) to help animate the Knight's Tour.

27.25 ***(Tortoise and Hare)*** Produce a graphical version of the *Tortoise and Hare* simulation (Exercise 7.28). Simulate the mountain by drawing an arc that extends from the bottom-left corner of the window to the top-right corner. The tortoise and the hare should race up the mountain. Implement the graphical output to actually print the tortoise and the hare on the arc for every move. [*Hint:* Extend the length of the race from 70 to 300 to allow yourself a larger graphics area.]

27.26 ***(Drawing Spirals)*** Write an application that uses `Graphics` method `drawPolyline` to draw a spiral similar to the one shown in Fig. 27.33.

27.27 ***(Pie Chart)*** Write a program that inputs four numbers and graphs them 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.

27.28 ***(Selecting Shapes)*** Write an application that allows the user to select a shape from a `JComboBox` and draws it 20 times with random locations and dimensions in method `paintComponent`. The first item in the `JComboBox` should be the default shape that is displayed the first time `paintComponent` is called.

27.29 ***(Random Colors)*** Modify Exercise 27.28 to draw each of the 20 randomly sized shapes in a randomly selected color. Use all 13 predefined `Color` objects in an array of `Colors`.

27.30 ***(`JColorChooser` Dialog)*** Modify Exercise 27.28 to allow the user to select the color in which shapes should be drawn from a `JColorChooser` dialog.

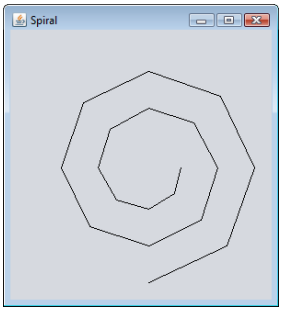


Fig. 27.33 | Spiral drawn using method `drawPolyline`.

(Optional) GUI and Graphics Case Study Exercise: Adding Java 2D

27.31 Java 2D introduces many new capabilities for creating unique and impressive graphics. We'll add a small subset of these features to the drawing application you created in Exercise 26.17. In this version, you'll enable the user to specify gradients for filling shapes and to change stroke characteristics for drawing lines and outlines of shapes. The user will be able to choose which colors compose the gradient and set the width and dash length of the stroke.

First, you must update the `MyShape` hierarchy to support Java 2D functionality. Make the following changes in class `MyShape`:

a) Change `abstract` method `draw`'s parameter type from `Graphics` to `Graphics2D`.
b) Change all variables of type `Color` to type `Paint` to enable support for gradients. [*Note:* Recall that class `Color` implements interface `Paint`.]

c) Add an instance variable of type `Stroke` in class `MyShape` and a `Stroke` parameter in the constructor to initialize the new instance variable. The default stroke should be an instance of class `BasicStroke`.

Classes `MyLine`, `MyBoundedShape`, `MyOval` and `MyRectangle` should each add a `Stroke` parameter to their constructors. In the `draw` methods, each shape should set the `Paint` and the `Stroke` before drawing or filling a shape. Since `Graphics2D` is a subclass of `Graphics`, we can continue to use `Graphics` methods `drawLine`, `drawOval`, `fillOval`, and so on to draw the shapes. When these methods are called, they'll draw the appropriate shape using the specified `Paint` and `Stroke` settings.

Next, you'll update the `DrawPanel` to handle the Java 2D features. Change all `Color` variables to `Paint` variables. Declare an instance variable `currentStroke` of type `Stroke` and provide a *set* method for it. Update the calls to the individual shape constructors to include the `Paint` and `Stroke` arguments. In method `paintComponent`, cast the `Graphics` reference to type `Graphics2D` and use the `Graphics2D` reference in each call to `MyShape` method `draw`.

Next, make the Java 2D features accessible from the GUI. Create a `JPanel` of GUI components for setting the Java 2D options. Add these components at the top of the `DrawFrame` below the panel that currently contains the standard shape controls (see Fig. 27.34). These GUI components should include:

a) A checkbox to specify whether to paint using a gradient.
b) Two `JButtons` that each show a `JColorChooser` dialog to allow the user to choose the first and second color in the gradient. (These will replace the `JComboBox` used for choosing the color in Exercise 26.17.)
c) A text field for entering the `Stroke` width.
d) A text field for entering the `Stroke` dash length.
e) A checkbox for selecting whether to draw a dashed or solid line.

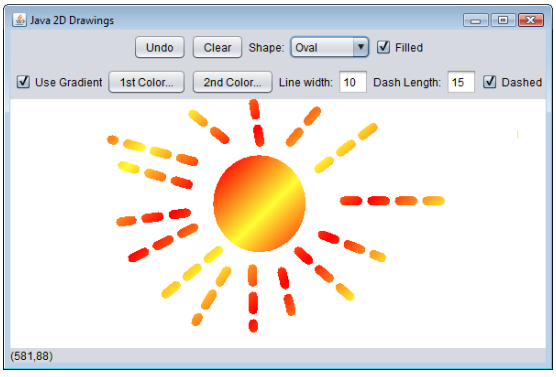


Fig. 27.34 | Drawing with Java 2D.

If the user selects to draw with a gradient, set the `Paint` on the `DrawPanel` to be a gradient of the two colors chosen by the user. The expression

```
new GradientPaint(0, 0, color1, 50, 50, color2, true))
```

creates a `GradientPaint` that cycles diagonally from the upper-left to the bottom-right every 50 pixels. Variables `color1` and `color2` represent the colors chosen by the user. If the user does not select to use a gradient, then simply set the `Paint` on the `DrawPanel` to be the first `Color` chosen by the user.

For strokes, if the user chooses a solid line, then create the `Stroke` with the expression

```
new BasicStroke(width, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND)
```

where variable `width` is the width specified by the user in the line-width text field. If the user chooses a dashed line, then create the `Stroke` with the expression

```
new BasicStroke(width, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND,
   10, dashes, 0)
```

where `width` again is the width in the line-width field, and `dashes` is an array with one element whose value is the length specified in the dash-length field. The `Panel` and `Stroke` objects should be passed to the shape object's constructor when the shape is created in `DrawPanel`.

Making a Difference

27.32 ***(Large-Type Displays for People with Low Vision)*** The accessibility of computers and the Internet to all people, regardless of disabilities, is becoming more important as these tools play increasing roles in our personal and business lives. According to a recent estimate by the World Health Organization (`http://www.who.int/mediacentre/factsheets/fs282/en/`), 246 million people worldwide have low vision. To learn more about low vision, check out the GUI-based low-vision simulation at `http://webaim.org/simulations/lowvision`. People with low vision might prefer to choose a font and/or a larger font size when reading electronic documents and web pages. Java has five built-in "logical" fonts that are guaranteed to be available in any Java implementation, including `Serif`, `Sans-serif` and `Monospaced`. Write a GUI application that provides a `JTextArea` in which the user can type text. Allow the user to select `Serif`, `Sans-serif` or `Monospaced` from a `JComboBox`. Provide a **Bold** `JCheckBox`, which, if checked, makes the text bold. Include **Increase Font Size** and **Decrease Font Size** `JButton`s that allow the user to scale the size of the font up or down, respectively, by one point at a time. Start with a font size of 18 points. For the purposes of this exercise, set the font size on the `JComboBox`, `JButton`s and `JCheckBox` to 20 points so that a person with low vision will be able to read the text on them.