Chapter 13
Graphics and Java 2D

Java How to Program, 10/e

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`.

13.1 Introduction

- Overview capabilities for drawing two-dimensional shapes, controlling colors and controlling fonts.
- One of Java's initial appeals 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 25 and two online chapters).
- Figure 13.1 shows a portion of the class hierarchy that includes various graphics classes and Java 2D API classes and interfaces covered in this chapter.

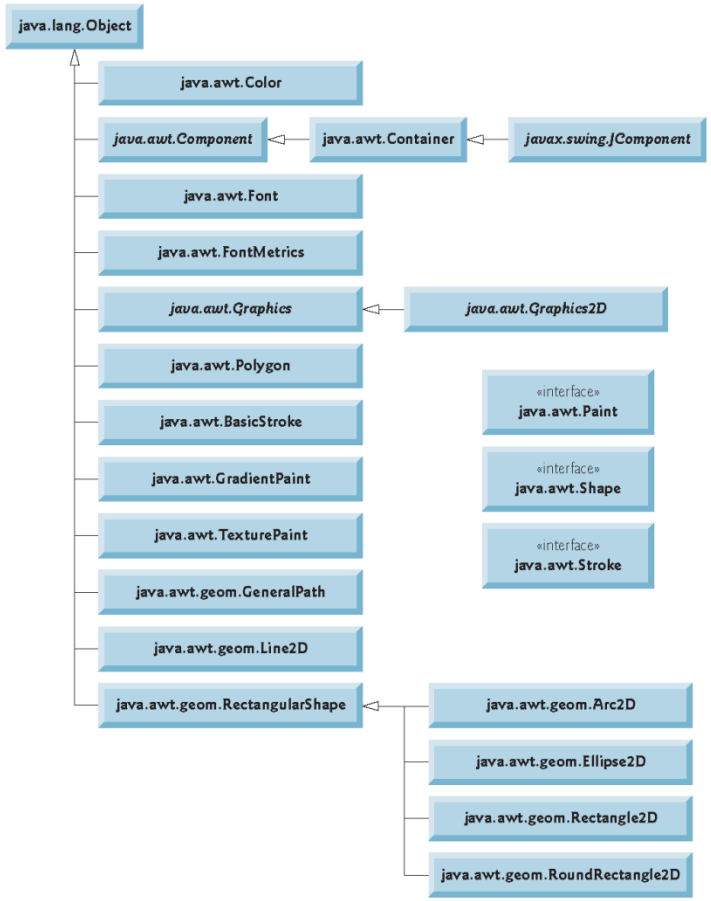

Fig. 13.1 | Classes and interfaces used in this chapter from Java's original graphics capabilities and from the Java 2D API.

Portability Tip 13.1

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.

13.1 Introduction (cont.)

- 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.

13.1 Introduction (cont.)

- 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.

13.1 Introduction (cont.)

- Coordinate system (Fig. 13.2)
  - a scheme for identifying every *point* on the screen.
- 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).
  - *x*-coordinates from left to right.
  - *y*-coordinates from top to bottom.
- The *x*-axis describes every horizontal coordinate, and the *y*-axis every vertical coordinate.
- Coordinate units are measured in pixels.
  - A pixel is a display monitor's smallest unit of resolution.

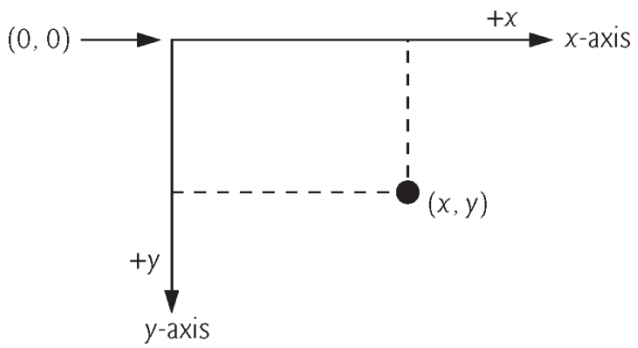

Fig. 13.2 | Java coordinate system. Units are measured in pixels.

13.2 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.
- `Graphics` objects contain methods for *drawing*, *font manipulation*, *color manipulation* and the like.
- Class `JComponent` (package `javax.swing`) contains a `paintComponent` for drawing graphics.
  - Takes a `Graphics` object as an argument.
  - Passed to the `paintComponent` method by the system when a lightweight Swing component needs to be repainted.

13.2 Graphics Contexts and Graphics Objects (cont.)

- When you create a GUI-based application, one of those threads is known as the event-dispatch thread (EDT) and it is used to process all GUI events.
- All manipulation of the on-screen GUI components must be performed in that thread.
- The application container calls method `paintComponent` (in the EDT) for each lightweight component as the GUI is displayed.
- If you need `paintComponent` to execute, you can call method repaint, which returns `void`, takes no arguments and is inherited by all `JComponent`s indirectly from class `Component` (package `java.awt`).

13.3 Color Control

- Class `Color` declares methods and constants for manipulating colors in a Java program.
- The predeclared color constants are summarized in Fig. 13.3, and several color methods and constructors are summarized in Fig. 13.4.
- Two of the methods in Fig. 13.4 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. 13.3 | `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.
`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()`	

Fig. 13.4 | `Color` methods and color-related `Graphics` methods. (Part 1 of 2.)

Method	Description
	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. 13.4 | `Color` methods and color-related `Graphics` methods. (Part 2 of 2.)

13.3 Color Control (cont.)

- Every color is created from a red, a green and a blue value.
  - RGB values: Integers in the range from 0 to 255, or floating-point values in the range 0.0 to 1.0.
  - Specifies the amount of red, the second the amount of green and the third the amount of blue.
  - Larger values == more of that particular color.
  - Approximately 16.7 million colors.
- `Graphics` method `getColor` returns a `Color` object representing the current drawing color.
- `Graphics` method `setColor` sets the current drawing color.

13.3 Color Control (cont.)

- `Graphics` method fillRect draws a *filled rectangle* in the current color.
- 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.
- `Graphics` method drawString draws a `String` in the current color.

```
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
```

Fig. 13.5 | Changing drawing colors. (Part 1 of 2.)

```
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. 13.5 | Changing drawing colors. (Part 2 of 2.)

```
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");
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. 13.6 | Demonstrating Colors. (Part 1 of 2.)

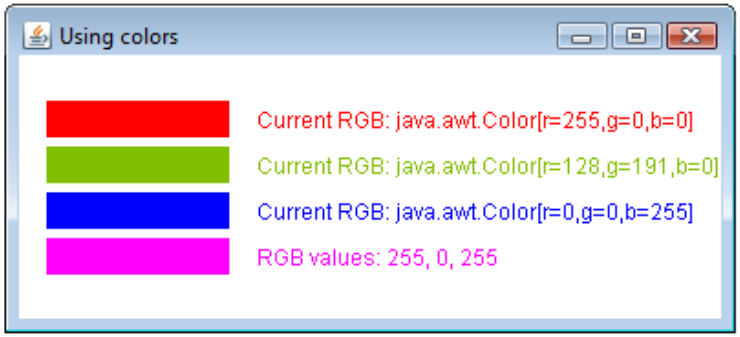

Fig. 13.6 | Demonstrating `Color`s. (Part 2 of 2.)

Look-and-Feel Observation 13.1

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.

Software Engineering Observation 13.1

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).*

13.3 Color Control (cont.)

- The JColorChooser component (package `javax.swing`) enables application users to select colors.
- `JColorChooser` `static` method showDialog creates a `JColorChooser` object, attaches it to a dialog box and displays the dialog.
  - Returns the selected `Color` object, or `null` if the user presses Cancel or closes the dialog without pressing OK.
  - 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.
- Method `setBackground` changes the background color of a `Component`.

```
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);
```

Fig. 13.7 | Choosing colors with `JColorChooser`. (Part 1 of 3.)

```
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
```

Fig. 13.7 | Choosing colors with `JColorChooser`. (Part 2 of 3.)

```
48
49          add(colorJPanel, BorderLayout.CENTER);
50          add(changeColorJButton, BorderLayout.SOUTH);
51
52          setSize(400, 130);
53          setVisible(true);
54       } // end ShowColor2JFrame constructor
55    } // end class ShowColors2JFrame
```

Fig. 13.7 | Choosing colors with `JColorChooser`. (Part 3 of 3.)

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

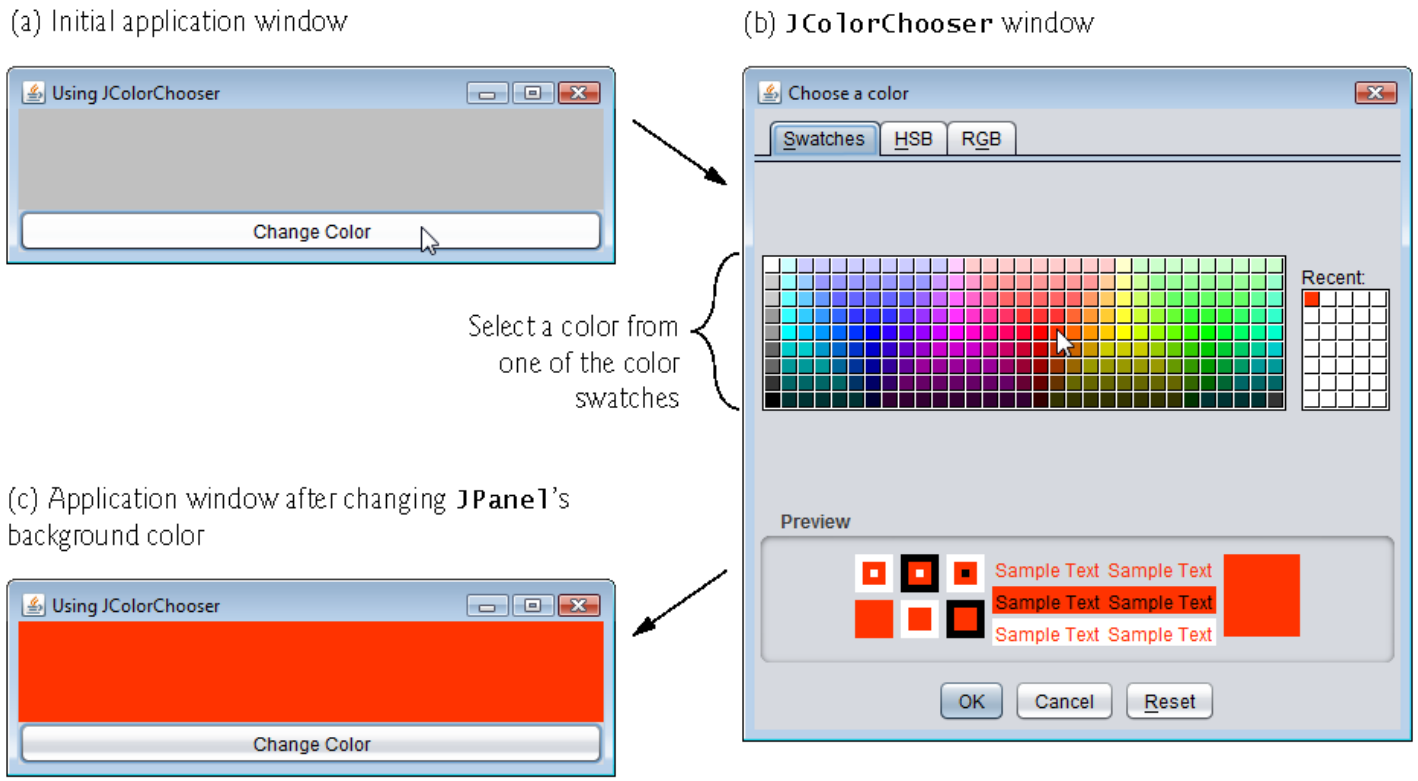

Fig. 13.8 | Choosing colors with `JColorChooser`. (Part 2 of 2.)

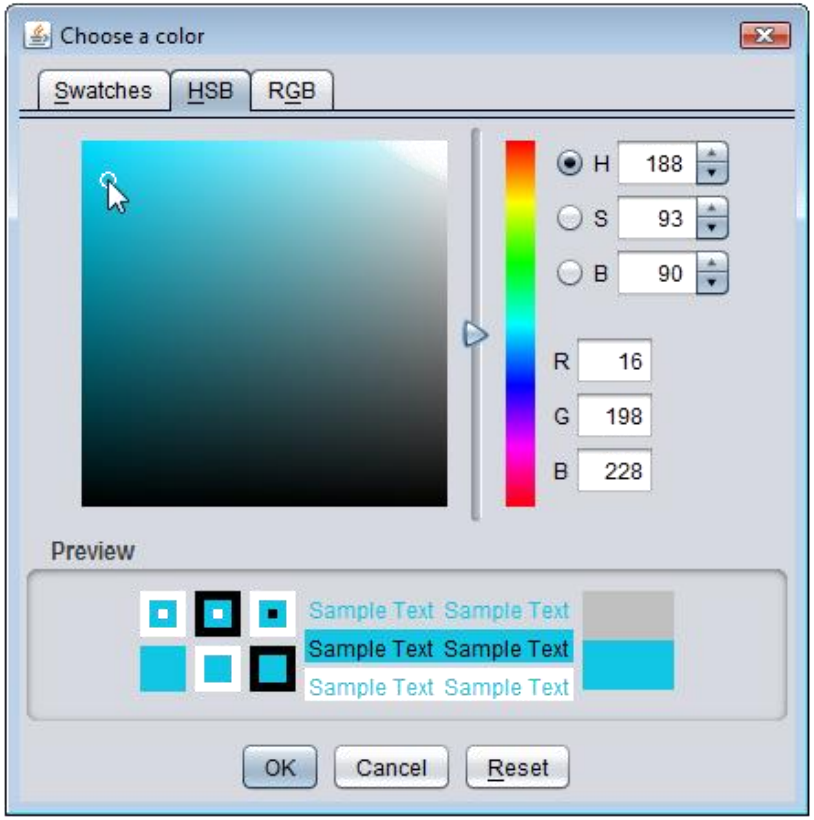

Fig. 13.9 | **HSB** and **RGB** tabs of the `JColorChooser` dialog. (Part 1 of 2.)

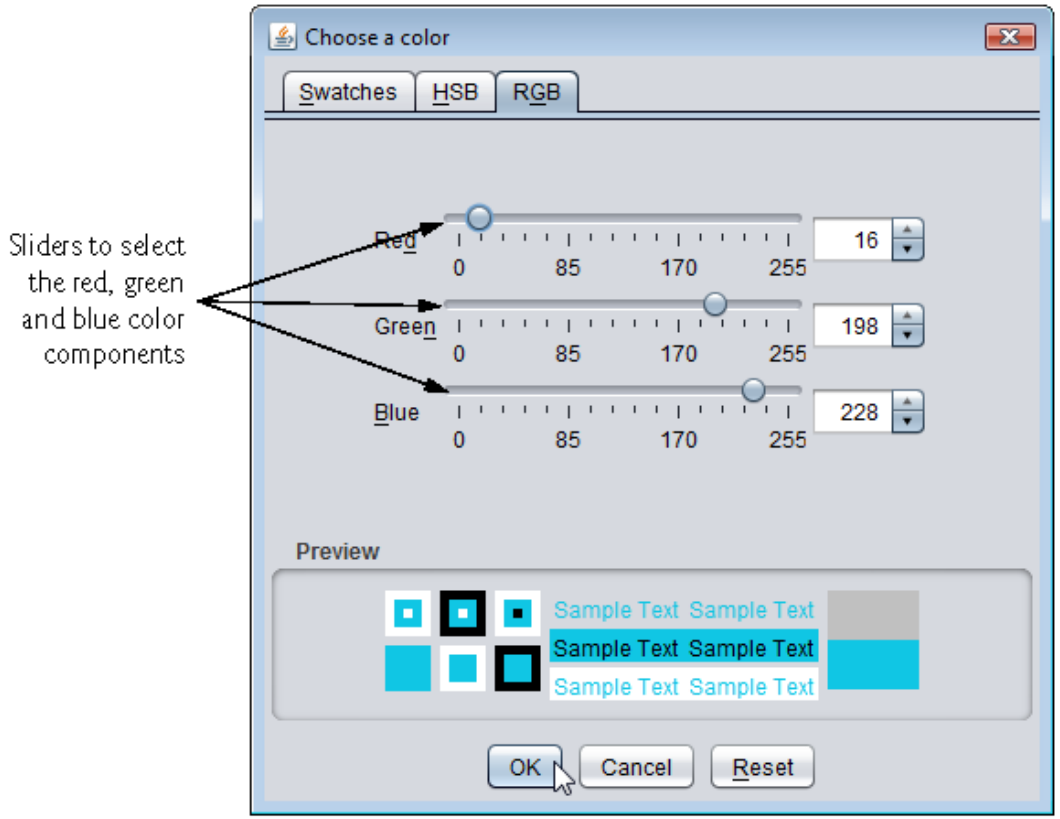

Fig. 13.9 | **HSB** and **RGB** tabs of the `JColorChooser` dialog. (Part 2 of 2.)

13.4 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. 13.10.
- Class `Font`'s constructor takes three arguments—the font name, font style and font size.
  - 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.
  - Font styles can be used in combination.
- 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.

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. 13.10 | `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. 13.10 | `Font`-related methods and constants. (Part 2 of 2.)

Portability Tip 13.2

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.*

Software Engineering Observation 13.2

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;
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
```

Fig. 13.11 | Display strings in different fonts and colors. (Part 1 of 2.)

```
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. 13.11 | Display strings in different fonts and colors. (Part 2 of 2.)

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

Fig. 13.12 | Using fonts. (Part 1 of 2.)

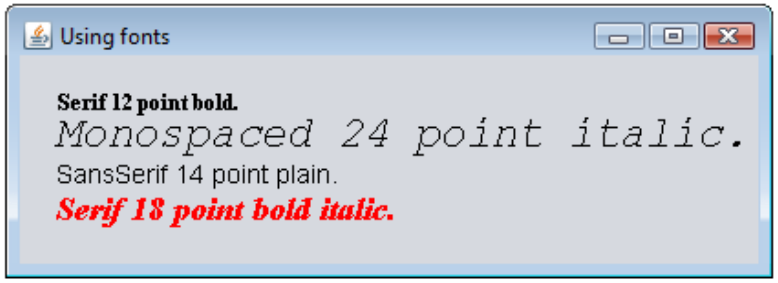

Fig. 13.12 | Using fonts. (Part 2 of 2.)

13.4 Manipulating Fonts (cont.)

- Figure 13.13 illustrates some of the common font metrics, which provide precise information about a font
  - Height
  - descent (the amount a character dips below the baseline)
  - ascent (the amount a character rises above the baseline)
  - 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).
- Class FontMetrics declares several methods for obtaining font metrics.

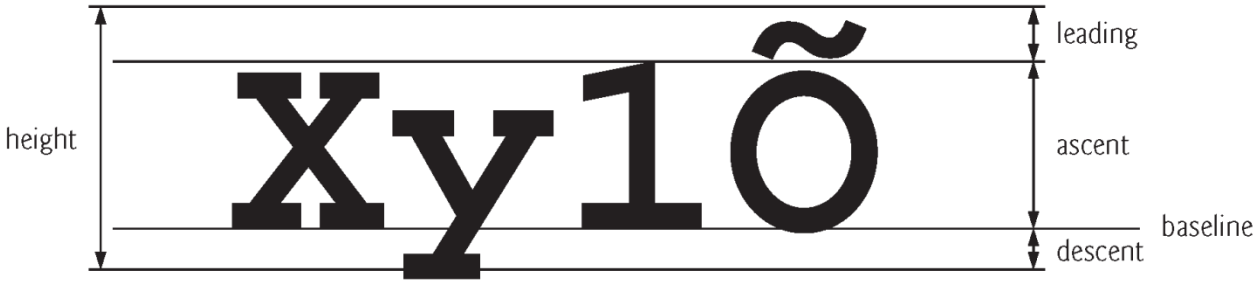

Fig. 13.13 | Font metrics.

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. 13.14 | `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
```

Fig. 13.15 | `FontMetrics` and `Graphics` methods useful for obtaining font metrics. (Part 1 of 2.)

```
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. 13.15 | `FontMetrics` and `Graphics` methods useful for obtaining font metrics. (Part 2 of 2.)

```
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. 13.16 | Displaying font metrics.

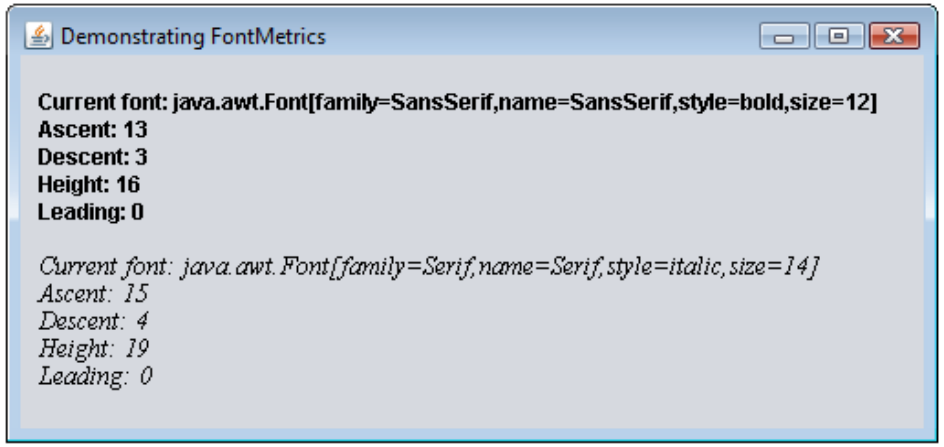

Fig. 13.16 | Displaying font metrics.

13.5 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. 13.17.

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.

Fig. 13.17 | `Graphics` methods that draw lines, rectangles and ovals. (Part 1 of 3.)

Method	Description
`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. 13.20). 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. 13.20).
`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.

Fig. 13.17 | `Graphics` methods that draw lines, rectangles and ovals. (Part 2 of 3.)

Method	Description
`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. 13.21). 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. 13.21).

Fig. 13.17 | `Graphics` methods that draw lines, rectangles and ovals. (Part 3 of 3.)

```
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
```

Fig. 13.18 | Drawing lines, rectangles and ovals. (Part 1 of 2.)

```
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. 13.18 | Drawing lines, rectangles and ovals. (Part 2 of 2.)

```
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. 13.19 | Testing `LinesRectsOvalsJPanel`. (Part 1 of 2.)

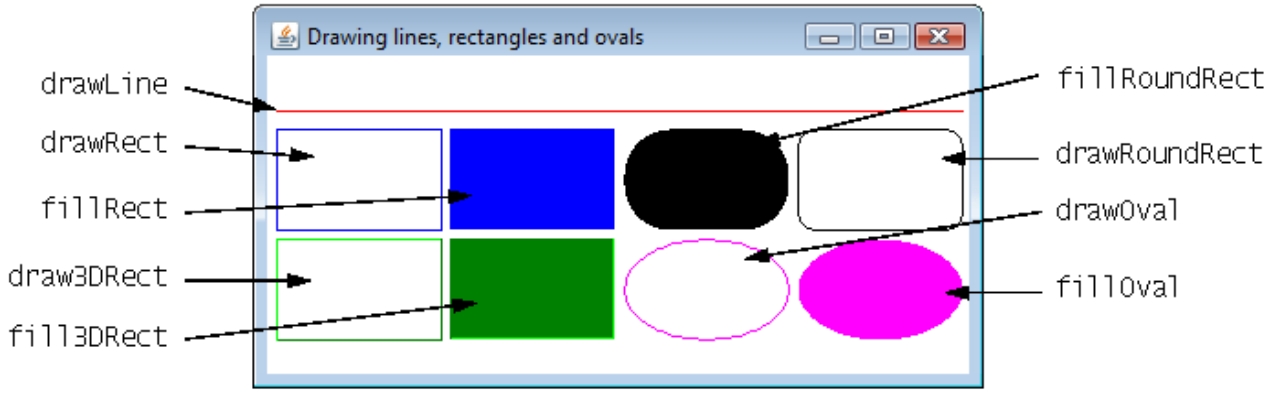

Fig. 13.19 | Testing `LinesRectsOvalsJPanel`. (Part 2 of 2.)

13.5 Drawing Lines, Rectangles and Ovals (cont.)

- Figure 13.20 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.
- Figure 13.21 shows an oval bounded by a rectangle.

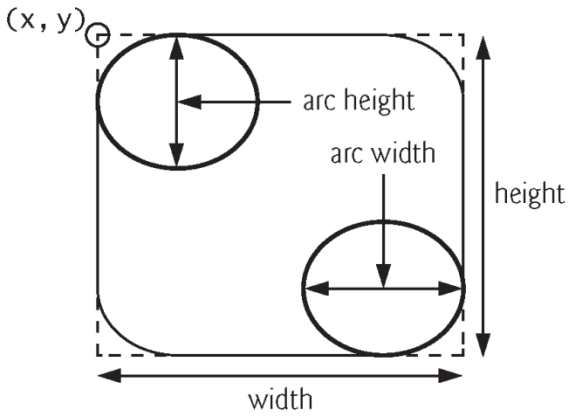

Fig. 13.20 | Arc width and arc height for rounded rectangles.

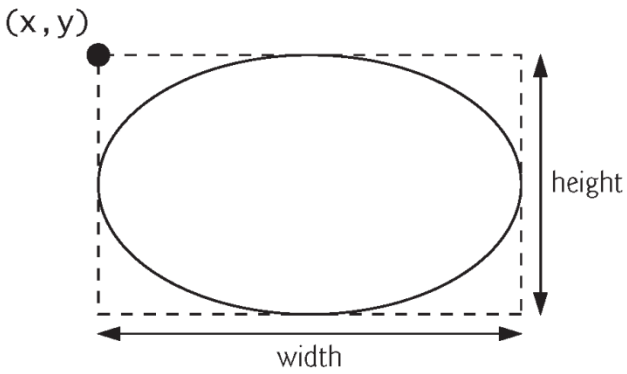

Fig. 13.21 | Oval bounded by a rectangle.

13.6 Drawing Arcs

- An arc is drawn as a portion of an oval.
  - Arc angles are measured in degrees.
  - Arcs sweep from a starting angle by the number of degrees specified by their arc angle.
- Arcs that sweep in a *counterclockwise* direction are measured in positive degrees.
- Arcs that sweep in a *clockwise* direction are measured in negative degrees.
- When drawing an arc, we specify a bounding rectangle for an oval.
- The arc will sweep along part of the oval.

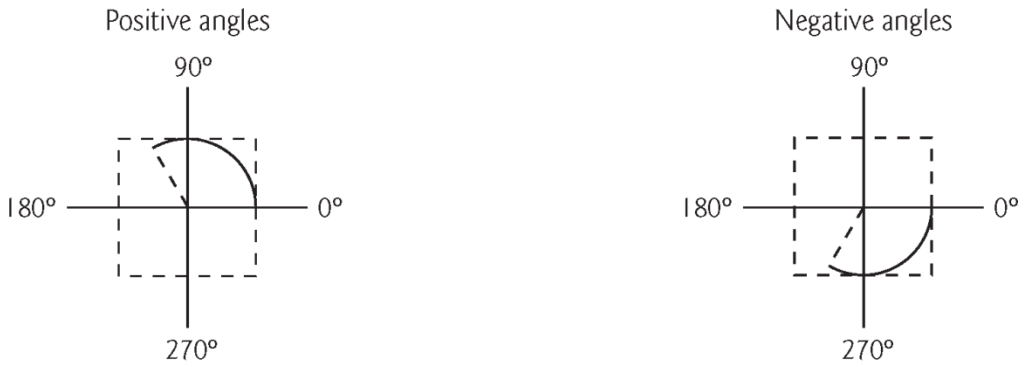

Fig. 13.22 | Positive and negative arc angles.

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. 13.23 | `Graphics` methods for drawing 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  {
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);
```

Fig. 13.24 | Arcs displayed with `drawArc` and `fillArc`. (Part 1 of 2.)

```
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. 13.24 | 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
14       ArcsJPanel arcsJPanel = new ArcsJPanel();
15       frame.add(arcsJPanel);
16       frame.setSize(300, 210);
17       frame.setVisible(true);
18    }
19 } // end class DrawArcs
```

Fig. 13.25 | Drawing arcs. (Part 1 of 2.)

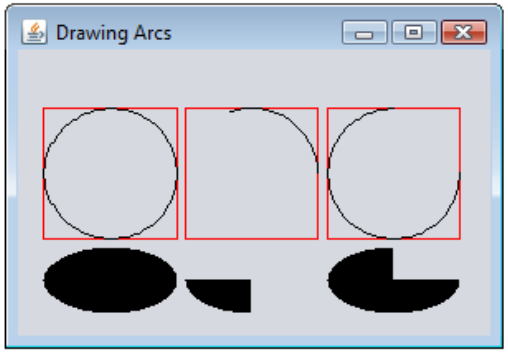

Fig. 13.25 | Drawing arcs. (Part 2 of 2.)

13.7 Drawing Polygons and Polylines

- Polygons are *closed multisided* shapes composed of straight-line segments.
- Polylines are *sequences of connected points*.
- Some methods require a Polygon object (package `java.awt`).

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. 13.26 | `Graphics` methods for polygons and class `Polygon` methods. (Part 1 of 3.)

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*.

Fig. 13.26 | `Graphics` methods for polygons and class `Polygon` methods. (Part 2 of 3.)

Method	Description
`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. 13.26 | `Graphics` methods for polygons and class `Polygon` methods. (Part 3 of 3.)

```
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
```

Fig. 13.27 | Polygons displayed with `drawPolygon` and `fillPolygon`. (Part 1 of 2.)

```
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);
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. 13.27 | Polygons displayed with `drawPolygon` and `fillPolygon`. (Part 2 of 2.)

```
1  // Fig. 13.28: DrawPolygons.java
2  // Drawing polygons.
3  import javax.swing.JFrame;
4
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. 13.28 | Drawing polygons. (Part 1 of 2.)

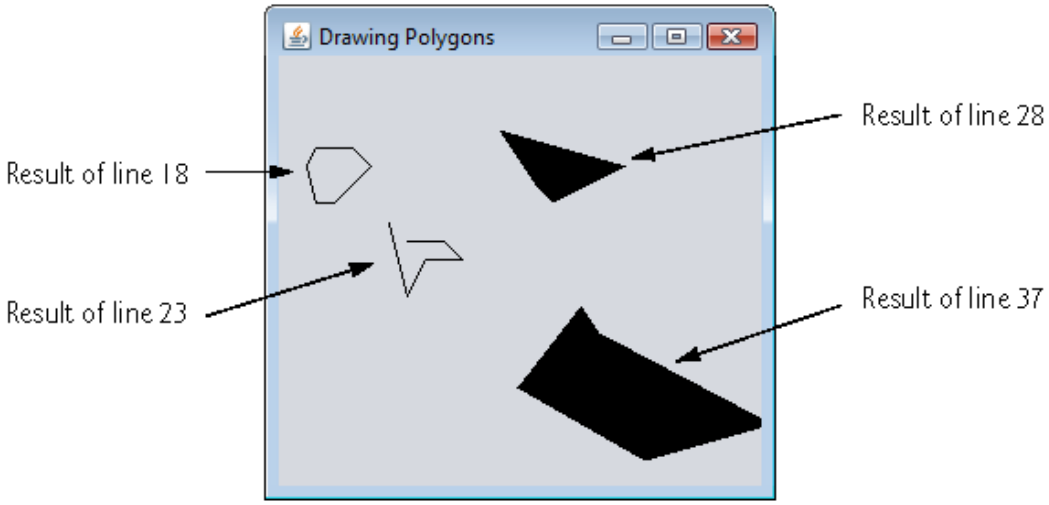

Fig. 13.28 | Drawing polygons. (Part 2 of 2.)

Common Programming Error 13.1

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.*

13.8 Java 2D API

- The Java 2D API provides advanced two-dimensional graphics capabilities for programmers who require detailed and complex graphical manipulations.
- For an overview, visit
  - http://docs.oracle.com/javase/7/docs/technotes/guides/2d/
- Drawing with the Java 2D API is accomplished with a Graphics2D reference (package `java.awt`).
- 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;`

13.8 Java 2D API (cont.)

- Example demonstrates several Java 2D shapes from package `java.awt.geom`, including Line2D.Double, Rectangle2D.Double, RoundRectangle2D.Double, Arc2D.Double and Ellipse2D.Double.

```
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
```

Fig. 13.29 | Demonstrating some Java 2D shapes. (Part 1 of 4.)

```
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
37          // draw 2D rounded rectangle with a buffered background
38          BufferedImage buffImage = new BufferedImage(10, 10,
39             BufferedImage.TYPE_INT_RGB);
```

Fig. 13.29 | Demonstrating some Java 2D shapes. (Part 2 of 4.)

```
2          // obtain Graphics2D from buffImage and draw on it
3          Graphics2D gg = buffImage.createGraphics();
4          gg.setColor(Color.YELLOW);
5          gg.fillRect(0, 0, 10, 10);
6          gg.setColor(Color.BLACK);
7          gg.drawRect(1, 1, 6, 6);
8          gg.setColor(Color.BLUE);
9          gg.fillRect(1, 1, 3, 3);
10         gg.setColor(Color.RED);
11         gg.fillRect(4, 4, 3, 3); // draw a filled rectangle
12
13         // paint buffImage onto the JFrame
14         g2d.setPaint(new TexturePaint(buffImage,
15            new Rectangle(10, 10)));
16         g2d.fill(
17            new RoundRectangle2D.Double(155, 30, 75, 100, 50, 50));
18
19         // draw 2D pie-shaped arc in white
20         g2d.setPaint(Color.WHITE);
21         g2d.setStroke(new BasicStroke(6.0f));
22         g2d.draw(
23            new Arc2D.Double(240, 30, 75, 100, 0, 270, Arc2D.PIE));
24
```

Fig. 13.29 | Demonstrating some Java 2D shapes. (Part 3 of 4.)

```
25          // draw 2D lines in green and yellow
26          g2d.setPaint(Color.GREEN);
27          g2d.draw(new Line2D.Double(395, 30, 320, 150));
28
29          // draw 2D line using stroke
30          float[] dashes = {10}; // specify dash pattern
31          g2d.setPaint(Color.YELLOW);
32          g2d.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND,
33             BasicStroke.JOIN_ROUND, 10, dashes, 0));
34          g2d.draw(new Line2D.Double(320, 30, 395, 150));
35       }
36    } // end class ShapesJPanel
```

Fig. 13.29 | Demonstrating some Java 2D shapes. (Part 4 of 4.)

```
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     {
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. 13.30 | Testing ShapesJPanel. (Part 1 of 2.)

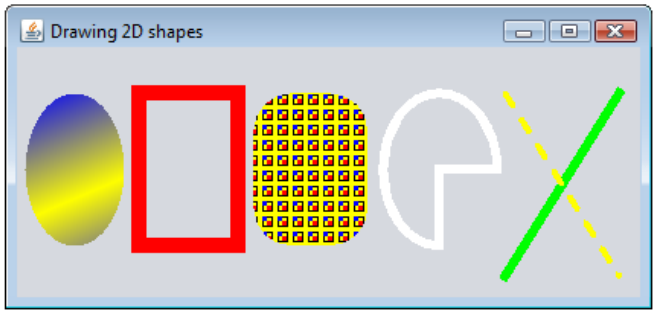

Fig. 13.30 | Testing `ShapesJPanel`. (Part 2 of 2.)

13.8 Java 2D API (cont.)

- `Graphics2D` method setPaint sets the Paint object that determines the color for the shape to display.
- A `Paint` object implements interface `java.awt.Paint`.
  - Can be something one of the predeclared `Color`, or it can be an instance of the Java 2D API's `GradientPaint`, SystemColor, `TexturePaint`, `LinearGradientPaint` or `RadialGradientPaint` classes.
- Class `GradientPaint` helps draw a shape in *gradually changing colors*—called a gradient.
- `Graphics2D` method fill draws a filled Shape object—an object that implements interface `Shape` (package `java.awt`).

13.8 Java 2D API (cont.)

- `Graphics2D` method setStroke sets the characteristics of the shape's border (or the lines for any other shape).
  - Requires as its argument an object that implements interface Stroke (package `java.awt`).
- 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).
- `Graphics2D` method draw draws a `Shape` object.

13.8 Java 2D API (cont.)

- Class BufferedImage (package `java.awt.image`) can be used to produce images in color and grayscale.
- The third argument BufferedImage.TYPE_INT_RGB indicates that the image is stored in color using the RGB color scheme.
- `BufferedImage` method create-Graphics creates a `Graphics2D` object for drawing into the `BufferedImage`.
- A `TexturePaint` object uses the image stored in its associated `BufferedImage` (the first constructor argument) as the fill texture for a filled-in shape.

13.8 Java 2D API (cont.)

- 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.
- 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.

13.8 Java 2D API (cont.)

- BasicStroke.CAP_ROUND causes a line to have rounded ends.
- If lines join together (as in a rectangle at the corners), use BasicStroke.JOIN_ROUND to indicate a rounded join.

13.8 Java 2D API (cont.)

- General path—constructed from straight lines and complex curves.
- Represented with an object of class GeneralPath (package `java.awt.geom`).
- `GeneralPath` method moveTo moves to the specified point.
- `GeneralPath` method lineTo draws a line from the current point to the specified point.
- `GeneralPath` method closePath draws a line from the last point to the point specified in the last call to `moveTo`.
- `Graphics2D` method translate moves the drawing origin to the specified location.
- `Graphics2D` method rotate rotates the next displayed shape.
  - The argument specifies the rotation angle in radians (with 360° = 2π radians).

```
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. 13.31 | 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. 13.31 | 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. 13.32 | Demonstrating a general path. (Part 1 of 2.)

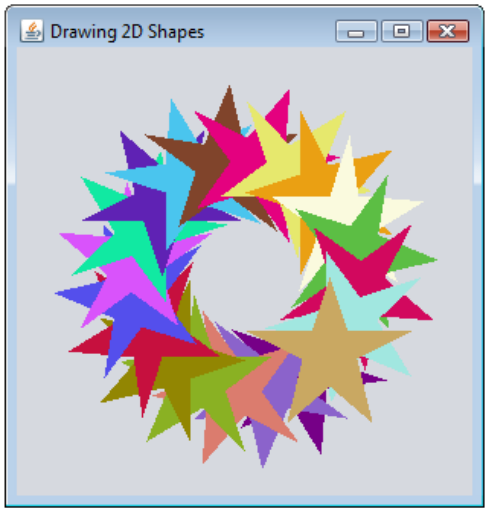

Fig. 13.32 | Demonstrating a general path. (Part 2 of 2.)