CHAPTER 16

# Graphics

In this chapter, you will:

- Learn about the `paint()` and `repaint()` methods
- Use the `drawString()` method to draw `String`s using various fonts and colors
- Draw lines and shapes
- Learn more about fonts
- Draw with Java 2D graphics

## Learning About the `paint()` and `repaint()` Methods

When you run a Java program that contains graphics, such as the `JFrame` applications in the previous chapters, the display surface frequently must be redisplayed, or **rerendered**. Redisplaying a surface also is called **painting**. Painting operations fall into two broad categories based on what causes them:

- **System-triggered painting** operations occur when the system asks a component to render its contents. This happens when the component is first made visible, if it is resized, or if it is damaged. For example, a component becomes damaged when another component that previously covered part of it has been moved, revealing a portion that was not visible.
- **Application-triggered painting** operations occur when the internal state of a component has changed. For example, when a user clicks a button, a "pressed" version of the button must be rendered.

Whether a paint request is triggered by the system or by an application, a `Component`'s **`paint()` method** is invoked. The header for the `paint()` method is:

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

The parameter to the method is a `Graphics` object. The **`Graphics` class** is an abstract class that descends directly from `Object` and holds data about graphics operations and methods for drawing shapes, text, and images. When AWT invokes the `paint()` method, the `Graphics` object parameter is preconfigured with the appropriate values for drawing on the component:

- The `Graphics` object's *color* is set to the component's `foreground` property.
- The `Graphics` object's *font* is set to the component's `font` property.
- The `Graphics` object's *translation* is set such that the coordinates 0, 0 represent the upper-left corner of the component.
- The `Graphics` object's *clip rectangle* is set to the area of the component that needs repainting.

Programs must use this `Graphics` object (or one derived from it) to render graphic output. They can change the values of the `Graphics` object as necessary.

You override the `paint()` method in your programs when you want specific actions to take place when components must be rendered. You don't usually call the `paint()` method directly. Instead, you call the **`repaint()` method**, which you can use when a window needs to be updated, such as when it contains new images or you have moved a new object onto the screen. The Java system calls the `repaint()` method when it needs to update a window, or you can call it yourself—in either case, `repaint()` creates a `Graphics` object for you that becomes the `paint()` method parameter. The `repaint()` method calls another method named `update()`, which clears its `Component`'s content pane and calls the `paint()` method. The series of events is best described with an example. Figure 16-1 shows a `JDemoPaint` class that extends `JFrame`. The frame contains a `JButton`. The constructor sets a title, layout manager, and default close operation, and it adds the button to the frame. The button is designated as a source for actions to which the frame can respond.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoPaint extends JFrame implements ActionListener
{
   JButton pressButton = new JButton("Press");
   public JDemoPaint()
   {
      setTitle("Paint Demo");
      setLayout(new FlowLayout());
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      add(pressButton);
      pressButton.addActionListener(this);
   }
   public void actionPerformed(ActionEvent e)
   {
      System.out.print("Button pressed. ");
      repaint();
   }
   public void paint(Graphics g)
   {
      super.paint(g);
      System.out.println("In paint method.");
   }
   public static void main(String[] args)
   {
      JDemoPaint frame = new JDemoPaint();
      frame.setSize(150, 100);
      frame.setVisible(true);
   }
}
```

**Figure 16-1** The JDemoPaint class

In Figure 16-1, the shaded first line of code in the paint() method is super.paint(g);. This statement is a call to the paint() method that is part of JDemoPaint's parent class (JFrame), and it passes the local Graphics object (named g) to this method. Although this program and others in this chapter will work without this statement, omitting it causes errors in more complicated applications. For now, get in the habit of including this method call as the first statement in any JFrame's paint() method, using whatever local name you have declared for your paint() method's Graphics argument.

In the JDemoPaint class in Figure 16-1, the actionPerformed() method executes when the user presses the JButton. The method contains a call to repaint(), which is unseen in the class and which automatically calls the paint() method.

The paint() method in the JDemoPaint class overrides the automatically supplied paint() method. The paint() method displays a line of output at the command line—it announces that the paint() method is executing. Figure 16-2 shows a typical execution of the program. The JFrame is first drawn when it is constructed, and the message "In paint method." appears

at the command line. When the user clicks the button on the frame, two messages are displayed: "Button pressed." from the `actionPerformed()` method, and "In paint method." from the `paint()` method that is called by `repaint()`. When the user minimizes and restores the frame, `paint()` is called automatically, and the "In paint method." message is displayed again.



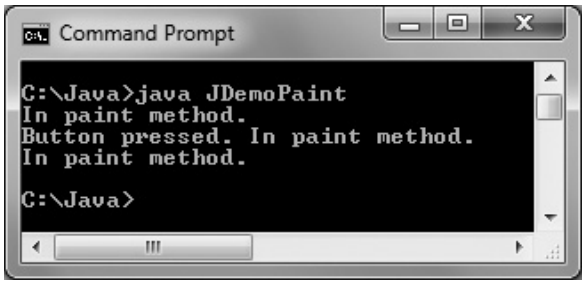


**Figure 16-2** Typical execution of the `JDemoPaint` program

If you call `repaint()` alone in a class that is a container, then the entire container is repainted. (The call to `repaint()` in Figure 16-1 is actually `this.repaint();`.) Repainting the entire container might be unnecessary and waste time if only part of the container has changed. If you call `repaint()` with a component, as in `pressButton.repaint()`, then only that component is repainted.

The `repaint()` method only requests that Java repaint the screen. If a second request to `repaint()` occurs before Java can carry out the first request, Java executes only the last `repaint()` method.

Before the built-in `paint()` method is called, the entire container is filled with its background color. Then the `paint()` method redraws the contents. The effect is that components are "erased" before being redrawn.

## Using the `setLocation()` Method

The **`setLocation()` method** allows you to place a component at a specific location within a `JFrame`'s content pane. In Chapter 15, you learned that a window or frame consists of a number of pixels on the screen, and that any component you place on the screen has a horizontal, or x-axis, position as well as a vertical, or y-axis, position. The horizontal position number increases from left to right across the screen, and the vertical position number increases from top to bottom.

When you allow a layout manager to position components, specific positions are selected automatically for each component. You can change the position of a component by using the `setLocation()` method and passing it x- and y-coordinate positions. For example, to position a `JLabel` object named `someLabel` at the upper-left corner of a `JFrame`, you write the following within the `JFrame` class:

```
someLabel.setLocation(0, 0);
```

If a window is 200 pixels wide by 100 pixels tall, you can place a `Button` named `pressMe` in the approximate center of the window with the following statement:

```
pressMe.setLocation(100, 50);
```

The coordinate arguments can be numeric constants or variables.

When you use `setLocation()`, the upper-left corner of the component is placed at the specified x- and y-coordinates. In other words, if a window is 100 by 100 pixels, `aButton.setLocation(100,100);` places the `JButton` outside the window, where you cannot see the component.

Figure 16-3 shows a `JDemoLocation` class that uses a call to the `setLocation()` method in the `actionPerformed()` method. The values of the x- and y-coordinates passed to `setLocation()` are initialized to 0, and then each is increased by 30 every time the user clicks the `JButton`. The `JButton` moves 30 pixels down and to the right every time it is clicked.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoLocation extends JFrame implements ActionListener
{
   JButton pressButton = new JButton("Press");
   int x = 0, y = 0;
   final int GAP = 30;
   public JDemoLocation()
   {
      setTitle("Location Demo");
      setLayout(new FlowLayout());
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      add(pressButton);
      pressButton.addActionListener(this);
   }
   public void actionPerformed(ActionEvent e)
   {
      pressButton.setLocation(x, y);
      x += GAP;
      y += GAP;
   }
   public static void main(String[] args)
   {
      JDemoLocation frame = new JDemoLocation();
      frame.setSize(150, 150);
      frame.setVisible(true);
   }
}
```

**Figure 16-3** The `JDemoLocation` class

Figure 16-4 shows the `JFrame` in four positions: when it starts with the `JButton` in its default position; after the user clicks the `JButton` once, moving it to position 0, 0; after the user clicks it a second time, moving it to position 30, 30; and after the user clicks it a third time, moving it to position 60, 60. If the user continues to click the `JButton`, it moves off the frame surface. You could add a decision to prevent continued progression of the `setLocation()` coordinates.


**Figure 16-4** Execution of the `JDemoLocation` program

The `setLocation()` method works correctly only when it is used after the layout manager has finished positioning all the application's components (or in cases where no layout manager is functioning). If you try to use `setLocation()` on a component within its container's constructor, the component will not be repositioned because the layout manager will not be finished placing components.

## Creating `Graphics` Objects

When you call the `paint()` method from an application, you can use the automatically created `Graphics` object that is passed to it, but you can also instantiate your own `Graphics` objects. For example, you might want to use a `Graphics` object when some action occurs, such as a mouse event. Because the `actionPerformed()` method does not supply you with a `Graphics` object automatically, you can create your own.

To display a string when the user clicks a `JButton`, you can code an `actionPerformed()` method such as the following:

```
public void actionPerformed(ActionEvent e)
{
   Graphics draw = getGraphics();
   draw.drawString("You clicked the button!", 50, 100);
}
```

This method instantiates a `Graphics` object named `draw`. (You can use any legal Java identifier.) The `getGraphics()` method provides the `draw` object with `Graphics` capabilities. Then the `draw` object can employ `Graphics` methods such as `setFont()`, `setColor()`, and `drawString()`.

Notice that when you create the `draw` object, you are not calling the `Graphics` constructor directly. (The name of the `Graphics` constructor is `Graphics()`, not `getGraphics()`.)

This operation is similar to the way you call `getContentPane()`. You are not allowed to call the `Graphics` or `ContentPane` constructors because those classes are `abstract` classes.

If you call `getGraphics()` in a frame that is not visible, you receive a `NullPointerException`, and the program will not execute.



Watch the video *Using `paint()` and `repaint()`.*

**TWO TRUTHS & A LIE**

**Learning About the `paint()` and `repaint()` Methods**

1. Painting can be system triggered (for example, when a component is resized) or application triggered (for example, when a user clicks a button).
2. When the `paint()` method is called, the `Graphics` object parameter is preconfigured with the appropriate state for drawing on the component, including the color and font.
3. You override the `repaint()` method in your programs when you want specific actions to take place when components must be rendered. You usually call the `paint()` method directly, and it calls `repaint()`.

The false statement is #3. You override the `paint()` method in your programs when you want specific actions to take place when components must be rendered. You don't usually call the `paint()` method directly—you call `repaint()`.

## Using the `drawString()` Method

The **`drawString()` method** allows you to draw a `String` in a `JFrame` or other component. The `drawString()` method requires three arguments: a `String`, an x-axis coordinate, and a y-axis coordinate.

You are already familiar with x- and y-axis coordinates because you used them with the `setLocation()` method for components. However, there is a minor difference in how you place components using the `setLocation()` method and how you place `String`s using the `drawString()` method. When you use x- and y-coordinates with components, such as `JButton`s or `JLabel`s, the upper-left corner of the component is placed at the coordinate position. When you use x- and y-coordinates with `drawString()`, the lower-left corner of the

String appears at the coordinates. Figure 16-5 shows the positions of a JLabel placed at the coordinates 30, 10 and a String placed at the coordinates 10, 30.

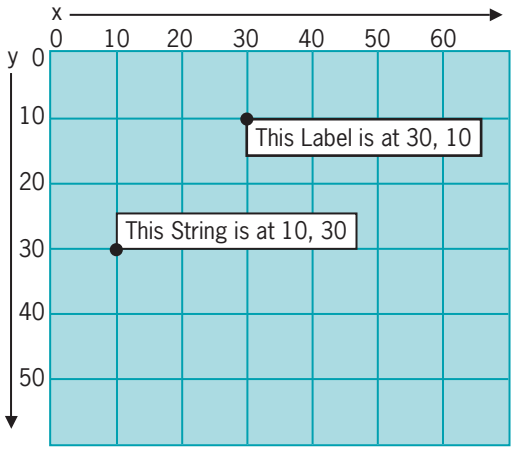


**Figure 16-5** Placement of String and JLabel objects on a frame

The drawString() method is a member of the Graphics class, so you need to use a Graphics object to call it. Recall that the paint() method header shows that the method receives a Graphics object from the update() method. If you use drawString() within paint(), the Graphics object you name in the header is available to you. For example, if you write a paint() method with the header public void paint(Graphics brush), you can draw a String within your paint() method by using a statement such as:

```
brush.drawString("Hi", 50, 80);
```

Interestingly, when you use the drawString() method with a negative font size, the string appears upside down. The coordinates then indicate the lower-right corner of the string.

## Using the setFont() and setColor() Methods

You can improve the appearance of strings drawn using Graphics objects by using the setFont() method. The setFont() method requires a Font object, which, as you may recall from Chapter 14, you can create with a statement such as:

```
Font someFont = new Font("Arial", Font.BOLD, 16);
```

Then you can instruct a Graphics object to use the font by inserting the font as the argument in a setFont() method. For example, if a Graphics object is named artist and a Font object is named smallFont, the font is set to smallFont with the following:

```
artist.setFont(smallFont);
```

Figure 16-6 shows an application that uses the setFont() method with a Graphics object named brush.

```
import javax.swing.*;
import java.awt.*;
public class JDemoFont extends JFrame
{
   Font bigFont = new Font("Serif", Font.ITALIC, 48);
   String hello = "Hello";
   public void paint(Graphics brush)
   {
      super.paint(brush);
      brush.setFont(bigFont);
      brush.drawString(hello, 10, 100);
   }
   public static void main(String[] args)
   {
      JDemoFont frame = new JDemoFont();
      frame.setSize(180, 150);
      frame.setVisible(true);
   }
}
```

**Figure 16-6** The JDemoFont class

When the paint() method executes in the JDemoFont example, bigFont is assigned to the automatically created brush object. Then the brush object is used to draw the hello string at position 10, 100. Figure 16-7 shows the output.


**Figure 16-7** Output of the JDemoFont program

## Using Color

You can designate a Graphics color with the setColor() method. As you learned in Chapter 15, the Color class contains 13 constants; you can use any of these constants as an argument to the setColor() method. For example, you can instruct a Graphics object named brush to apply green paint by using the following statement:

```
brush.setColor(Color.GREEN);
```

Until you change the color, subsequent graphics output appears as green.

## TWO TRUTHS & A LIE

### Using the drawString() Method

1. The `drawString()` method requires three arguments: a `String`, an x-axis coordinate, and a y-axis coordinate.
2. When you use x- and y-coordinates with components such as `JButton`s or `JLabel`s, the lower-left corner of the component is placed at the coordinate position, but when you use x- and y-coordinates with `drawString()`, the upper-left corner of the `String` appears at the coordinates.
3. The `drawString()` method is a member of the `Graphics` class, so you need to use a `Graphics` object to call it.

The false statement is #2. When you use x- and y-coordinates with components, such as `JButton`s or `JLabel`s, the upper-left corner of the component is placed at the coordinate position, but when you use x- and y-coordinates with `drawString()`, the lower-left corner of the `String` appears at the coordinates.

## You Do It

*Using the drawString() Method*

In the next steps, you write a class that extends `JFrame` and uses the `drawString()` method.

1. Open a new text file, and begin a class definition for a `JDemoGraphics` class by typing the following:

```
import javax.swing.*;
import java.awt.*;
public class JDemoGraphics extends JFrame
{
```

2. Declare a `String` by typing the following:

```
String movieQuote = new String("You talkin' to me?");
```

3. Add a constructor to set the default close operation:

```
public JDemoGraphics()
{
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

*(continues)*

*(continued)*

4. Type the following `paint()` method that calls the `super()` method and uses a `Graphics` object to draw the `movieQuote String`.

```
public void paint(Graphics gr)
{
   super.paint(gr);
   gr.drawString(movieQuote, 30, 100);
}
```

5. Add a `main()` method that instantiates a `JDemoGraphics` object and sets its size and visibility. Then add the closing curly brace for the class:

```
   public static void main(String[] args)
   {
      JDemoGraphics frame = new JDemoGraphics();
      frame.setSize(280, 200);
      frame.setVisible(true);
   }
}
```

6. Save the file as **JDemoGraphics.java**, and then compile and execute it. The program's output appears in Figure 16-8.


**Figure 16-8** Output of the `JDemoGraphics` program

7. Close the `JFrame` to end the application.

### *Using Fonts and Colors*

Next, you use your knowledge of fonts and colors to set the color and font style of a drawn `String`.

*(continues)*

*(continued)*

1. Open the **JDemoGraphics.java** text file in your text editor, and immediately save it as **JDemoGraphics2.java**. Change the class name, the constructor name, and the two references in the `main()` method to match.
2. Add a new `import` statement to the current list so that the application can use color:

```
import java.awt.Color;
```

3. Just after the `movieQuote` declaration, add a `Font` object by typing the following:

```
Font bigFont = new Font("Boopee", Font.ITALIC, 30);
```

4. Within the `paint()` method after the call to `super()`, type the following statements so the `gr` object uses the `bigFont` object and the color magenta:

```
gr.setFont(bigFont);
gr.setColor(Color.MAGENTA);
```

5. Following the existing `drawString()` method call, type the following lines to change the color and add another call to the `drawString()` method:

```
gr.setColor(Color.BLUE);
gr.drawString(movieQuote, 60, 140);
```

6. Save the file, compile it, and execute it. The program's output appears in Figure 16-9. Although the figure is shown in black and white in this book, notice that the `String`s on your screen are displayed as magenta and blue text. The font that appears in your program might be different from the one shown in the figure, depending on your computer's installed fonts. (Later in this chapter, you will learn how to view a list of all the available fonts on your computer.)

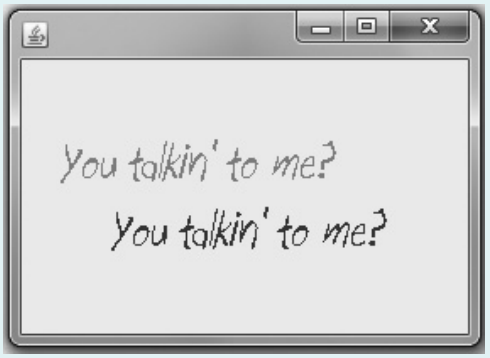


**Figure 16-9** Output of the `JDemoGraphics2` program

*(continues)*

*(continued)*

7. Close the JFrame to end the application.

*Creating Your Own Graphics Object*

Next, you create a Graphics object named pen and use the object to draw a String on the screen. The text of the String will appear to move each time a JButton is clicked.

1. Open a new text file in your text editor, and type the following import statements for the program:

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

2. Start typing the following class that extends JFrame and uses the mouse. The class defines a String, a JButton, a Font, and four integers: two to hold x- and y-coordinates, one to act as a constant size to measure the gap between lines displayed on the screen, and one to hold the size of the JFrame:

```
public class JDemoCreateGraphicsObject extends JFrame
    implements ActionListener
{
    String movieQuote = new String("Here's looking at you, kid");
    JButton moveButton = new JButton("Move It");
    Font broadwayFont = new Font("Broadway", Font.ITALIC, 12);
    int x = 10, y = 50;
    final int GAP = 20;
    final int SIZE = 400;
```

3. Type the following constructor; it changes the background color and sets the layout of the Container, adds the JButton, prepares the JFrame to listen for JButton events, sets the close operation, and sets the size of the frame:

```
public JDemoCreateGraphicsObject()
{
    Container con = getContentPane();
    con.setBackground(Color.YELLOW);
    con.setLayout(new FlowLayout() );
    con.add(moveButton);
    moveButton.addActionListener(this);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(SIZE, SIZE);
}
```

*(continues)*

*(continued)*

4. Within the `actionPerformed()` method, you can create a `Graphics` object and use it to draw the `String` on the screen. Each time a user clicks the `JButton`, the x- and y-coordinates both increase, so a copy of the movie quote appears slightly below and to the right of the previous one. Type the following `actionPerformed()` method to accomplish this processing:

```
public void actionPerformed(ActionEvent e)
{
   Graphics pen = getGraphics();
   pen.setFont(broadwayFont);
   pen.setColor(Color.MAGENTA);
   pen.drawString(movieQuote, x += GAP, y += GAP);
}
```

5. Add a `main()` method to instantiate a `JDemoCreateGraphicsObject` object and give it visibility. Add a closing curly brace for the class.

```
   public static void main(String[] args)
   {
      JDemoCreateGraphicsObject frame = new
         JDemoCreateGraphicsObject();
      frame.setVisible(true);
   }
}
```

6. Save the file as **JDemoCreateGraphicsObject.java**, and then compile and run the program. Click the **Move It** button several times to see the `String` message appear and move on the screen.

7. When you finish clicking the button, close the `JFrame` to end the application.

*Examining Screen Coordinates*

If you run `JDemoCreateGraphicsObject` and click the `JButton` enough times, the movie quote `String` appears to march off the bottom of the `JFrame`. Every time you click the `JButton`, the x- and y-coordinates used by `drawString()` increase, and there is no limit to their value. You can prevent this error by checking the screen coordinates' values to see if they exceed the `JFrame`'s dimensions.

1. Open the **JDemoCreateGraphicsObject** file, and immediately save it as **JDemoCreateGraphicsObject2**. Change the class name, constructor name, and the two references to the class in the `main()` method to match.

*(continues)*

*(continued)*

2. Because the screen size is 400 by 400, you can ensure that at least part of the `String` appears in the frame by preventing the y-coordinate from exceeding a value that is slightly less than 400. Create a constant to hold this limit by adding the following just after the declaration of `SIZE`:

```
final int LIMIT = SIZE - 50;
```

3. In the `actionPerformed()` method, replace the stand-alone call to `drawString()` with one that depends on `LIMIT` as follows:

```
if(y < LIMIT)
   pen.drawString(movieQuote, x += GAP, y += GAP);
```

4. Add an `else` clause that disables the `JButton` after the x-coordinate becomes too large:

```
else
   moveButton.setEnabled(false);
```

5. Save the file, compile it, and execute it. Now when you click the **Move It** button, the movie quote moves until the y-coordinate reaches 350. At that point, the `JButton` is disabled, and the movie quote no longer violates the frame's size limits.
6. Close the frame to end the program.

# Drawing Lines and Shapes

Just as you can draw `String`s using a `Graphics` object and the `drawString()` method, Java provides you with several methods for drawing a variety of lines and geometric shapes. Any line or shape is drawn in the current color you set with the `setColor()` method. When you do not set a color, lines are drawn in black by default.

It is almost impossible to draw a picture of any complexity without sketching it first on a piece of graph paper to help you determine correct coordinates.

## Drawing Lines

You can use the **`drawLine()` method** to draw a straight line between any two points on the screen. The `drawLine()` method takes four arguments: the x- and y-coordinates of the line's starting point and the x- and y-coordinates of the line's ending point. For example, if you create

a `Graphics` object named `pen`, then the following statement draws a straight line that slants down and to the right, from position 50, 50 to position 100, 200, as shown in Figure 16-10.

```
pen.drawLine(50, 50, 100, 200);
```



Because you can start at either end when you draw a line, an identical line is created with the following:

```
pen.drawLine(100, 200, 50, 50);
```

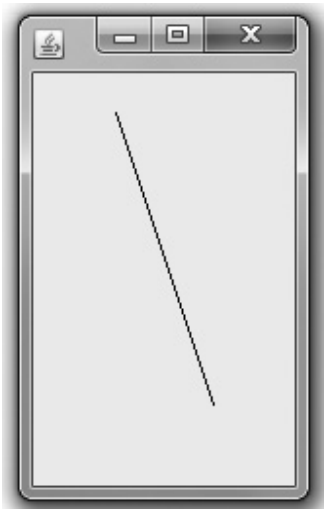

**Figure 16-10** A line created with `pen.drawLine(50, 50, 100, 200)`

Your downloadable student files contain a JDemoLine.java file with a working program that draws the line shown in Figure 16-10.

## Drawing Rectangles

You could draw a rectangle by drawing four lines. Alternatively, you can use the **`drawRect()` method** and **`fillRect()` method**, respectively, to draw the outline of a rectangle or to draw a solid, or filled, rectangle. Each of these methods requires four arguments. The first two arguments represent the x- and y-coordinates of the upper-left corner of the rectangle. The last two arguments represent the width and height of the rectangle. For example, the following statement draws a short, wide rectangle that begins at position 20, 100, and is 200 pixels wide by 10 pixels tall:

```
drawRect(20, 100, 200, 10);
```

The **`clearRect()` method** also requires four arguments and draws a rectangle. The difference between using the `drawRect()` and `fillRect()` methods and the `clearRect()` method is that

the first two methods use the current drawing color, whereas the `clearRect()` method draws what appears to be an empty or "clear" rectangle. A rectangle created with the `clearRect()` method is not really "clear"; in other words, it is not transparent. When you create a rectangle, you do not see objects that might be hidden behind it. Instead, the `clearRect()` method clears anything drawn from view, showing the original content pane.

For example, the constructor in the `JDemoRectangles` program shown in Figure 16-11 sets the background color of the content pane to blue and sets the layout manager.

```
import javax.swing.*;
import java.awt.*;
import java.awt.Color;
public class JDemoRectangles extends JFrame
{
   Container con = getContentPane();
   public JDemoRectangles()
   {
      con.setBackground(Color.BLUE);
      con.setLayout(new FlowLayout());
   }
   public void paint(Graphics gr)
   {
      super.paint(gr);
      gr.setColor(Color.RED);
      gr.fillRect(40, 40, 120, 120);
      gr.setColor(Color.YELLOW);
      gr.fillRect(80, 80, 160, 160);
      gr.clearRect(50, 60, 50, 50);
   }
   public static void main(String[] args)
   {
      JDemoRectangles frame = new JDemoRectangles();
      frame.setSize(200, 200);
      frame.setVisible(true);
   }
}
```

**Figure 16-11** The `JDemoRectangles` class

In the `paint()` method in Figure 16-11, the drawing color is set to red, and a filled rectangle is drawn. Then the drawing color is changed to yellow and a second filled rectangle is drawn to overlap the first. Finally, a smaller, "clear" rectangle is drawn that overlaps the other rectangles. As Figure 16-12 shows, you cannot see the boundaries of the original rectangles in the "clear" area—you simply see that portions of the filled rectangles have been removed from the drawing.

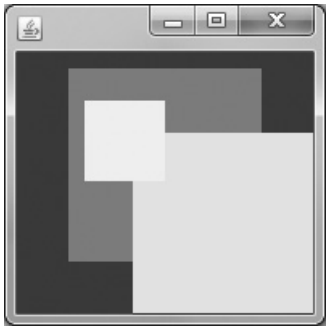

**Figure 16-12** Output of the `JDemoRectangles` program

You can create rectangles with rounded corners when you use the **`drawRoundRect()` method**. The `drawRoundRect()` method requires six arguments. The first four arguments match the four arguments required to draw a rectangle: the x- and y-coordinates of the upper-left corner, the width, and the height. The two additional arguments represent the arc width and height associated with the rounded corners (an **arc** is a portion of a circle). If you assign zeros to the arc coordinates, the rectangle is not rounded; instead, the corners are square. At the other extreme, if you assign values to the arc coordinates that are at least the width and height of the rectangle, the rectangle is so rounded that it is a circle. The `paint()` method in Figure 16-13 draws four rectangles with increasingly large corner arcs. The first rectangle is drawn at coordinates 20, 40, and the horizontal coordinate is increased by 100 for each subsequent rectangle. Each rectangle is the same width and height, but each set of arc values becomes larger, producing rectangles that are not rounded, slightly rounded, very rounded, and completely rounded in sequence. Figure 16-14 shows the program's output.

```
import javax.swing.*;
import java.awt.*;
public class JDemoRoundRectangles extends JFrame
{
   public void paint(Graphics gr)
   {
      super.paint(gr);
      int x = 20;
      int y = 40;
      final int WIDTH = 80, HEIGHT = 80;
      final int HORIZONTAL_GAP = 100;
      for(int arcSize = x; arcSize <= HEIGHT; arcSize += 20)
      {
         gr.drawRoundRect(x, y, WIDTH, HEIGHT, arcSize, arcSize);
         x += HORIZONTAL_GAP;
      }
   }
```

**Figure 16-13** The `JDemoRoundRectangles` class *(continues)*

(*continued*)

```
    public static void main(String[] args)
    {
       JDemoRoundRectangles frame = new JDemoRoundRectangles();
       frame.setSize(420, 140);
       frame.setVisible(true);
    }
}
```

**Figure 16-13** The `JDemoRoundRectangles` class

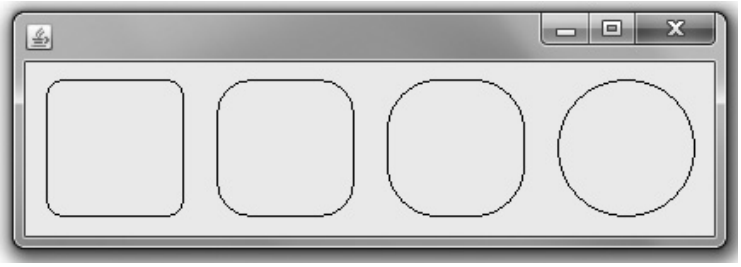

**Figure 16-14** Output of the `JDemoRoundRectangles` program

Java also contains a `fillRoundRect()` method that creates a filled rounded rectangle and a `clearRoundRect()` method that creates a clear rounded rectangle.

## Creating Shadowed Rectangles

The **`draw3DRect()` method** is a minor variation on the `drawRect()` method. You use the `draw3DRect()` method to draw a rectangle that appears to have "shadowing" on two of its edges—the effect is that of a rectangle that is lit from the upper-left corner and slightly raised or slightly lowered. The `draw3DRect()` method requires a fifth argument in addition to the x- and y-coordinates and width and height required by the `drawRect()` method. The fifth argument is a Boolean value, which is true if you want the raised rectangle effect (darker on the right and bottom) and false if you want the lowered rectangle effect (lighter on the right and bottom). There is also a **`fill3DRect()` method** for creating filled three-dimensional (3D) rectangles; this method is used in the program in Figure 16-15.

```
import javax.swing.*;
import java.awt.*;
import java.awt.Color;
public class JDemo3DRectangles extends JFrame
{
   public void paint(Graphics gr)
   {
      super.paint(gr);
      final int WIDTH = 60, HEIGHT = 80;
      gr.setColor(Color.PINK);
      gr.fill3DRect(20, 40, WIDTH, HEIGHT, true);
      gr.fill3DRect(100, 40, WIDTH, HEIGHT, false);
   }
   public static void main(String[] args)
   {
      JDemo3DRectangles frame = new JDemo3DRectangles();
      frame.setSize(180, 150);
      frame.setVisible(true);
   }
}
```

**Figure 16-15** The `JDemo3DRectangles` class

The program in Figure 16-15 creates two filled 3D rectangles in pink. (The 3D methods work best with lighter drawing colors.) You can see that the shadowing effect on the output in Figure 16-16 is very subtle; the shadowing is only one pixel wide.

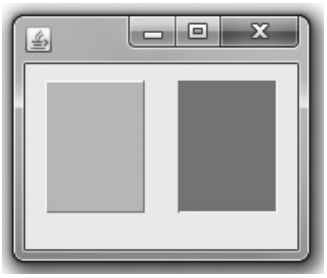

**Figure 16-16** Output of the `JDemo3DRectangles` program

## Drawing Ovals

You can draw an oval using the `drawRoundRect()` or `fillRoundRect()` method, but it is usually easier to use the **`drawOval()`** and **`fillOval()` methods**. The `drawOval()` and `fillOval()` methods both draw ovals using the same four arguments that rectangles use. When you supply `drawOval()` or `fillOval()` with x- and y-coordinates for the upper-left corner and width and height measurements, you can picture an imaginary rectangle that uses

the four arguments. The oval is then placed within the rectangle so it touches the rectangle at the center of each of the rectangle's sides. For example, suppose that you create a `Graphics` object named `tool` and draw a rectangle with the following statement:

```
tool.drawRect(50, 50, 100, 60);
```

Suppose that then you create an oval with the same coordinates as follows:

```
tool.drawOval(50, 50, 100, 60);
```

The output appears as shown in Figure 16-17, with the oval edges just skimming the rectangle's sides.

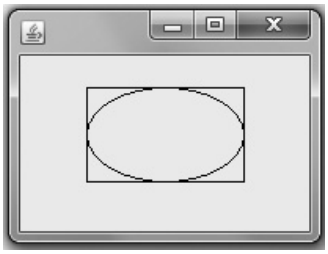

**Figure 16-17** Demonstration of the `drawOval()` method

Your downloadable student files contain a JDemoOval.java file that produces the frame in Figure 16-17.

If you draw a rectangle with identical height and width, you draw a square. If you draw an oval with identical height and width, you draw a circle.

## Drawing Arcs

In Java, you can draw an arc using the `Graphics` **`drawArc()` method**. To use the `drawArc()` method, you provide six arguments:

- The x- and y-coordinates of the upper-left corner of an imaginary rectangle that represents the bounds of the imaginary circle that contains the arc
- The width and height of the imaginary rectangle that represents the bounds of the imaginary circle that contains the arc
- The beginning arc position and the arc angle

Arc positions and angles are measured in degrees; there are 360 degrees in a circle. The 0° position for any arc is the three o'clock position, as shown in Figure 16-18. The other 359 degree positions increase as you move counterclockwise around an imaginary circle, so 90°

is at the top of the circle in the 12 o'clock position, 180° is opposite the starting position at nine o'clock, and 270° is at the bottom of the circle in the six o'clock position.



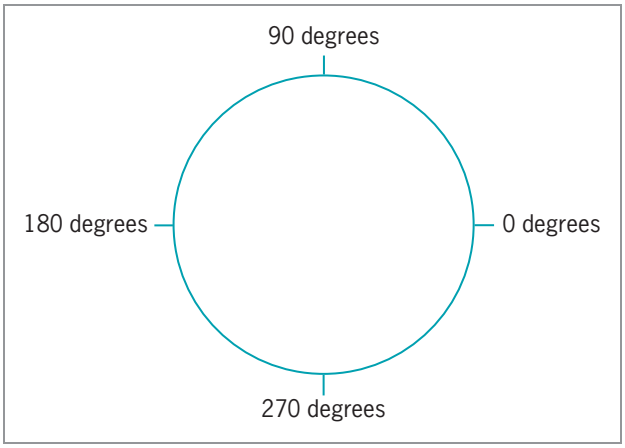


**Figure 16-18** Arc positions

The arc angle is the number of degrees over which you want to draw the arc, traveling counterclockwise from the starting position. For example, you can draw a half circle by indicating an arc angle of 180° or a quarter circle by indicating an arc angle of 90°. If you want to travel clockwise from the starting position, you express the degrees as a negative number. Just as when you draw a line, when drawing any arc you can take one of two approaches: either start at point A and travel to point B, or start at point B and travel to point A. For example, to create an arc object using a `Graphics` object named `g` that looks like the top half of a circle, the following statements produce identical results:

```
g.drawArc(x, y, w, h, 0, 180);
g.drawArc(x, y, w, h, 180, -180);
```

The first statement starts an arc at the three o'clock position and travels 180 degrees counterclockwise to the nine o'clock position. The second statement starts at nine o'clock and travels clockwise to three o'clock.

The **`fillArc()` method** creates a solid arc. The arc is drawn, and two straight lines are drawn from the arc endpoints to the center of the imaginary circle whose perimeter the arc occupies. For example, assuming you have declared a `Graphics` object named `g`, the following two statements together produce the output shown in Figure 16-19:

```
g.fillArc(20, 50, 100, 100, 20, 320);
g.fillArc(60, 50, 100, 100, 340, 40);
```

Each of the two arcs is in a circle that has a size of 100 by 100. The first arc almost completes a full circle, starting at position 20 (near two o'clock) and ending 320 degrees around the circle (at position 340, near four o'clock). The second filled arc more closely resembles a pie slice, starting at position 340 and extending 40 degrees to end at position 20.



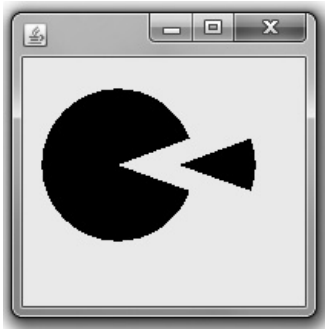

**Figure 16-19** Two filled arcs

Your downloadable student files contain a program named JDemoFillArc.java that produces Figure 16-19.

## Creating Polygons

When you want to create a shape that is more complex than a rectangle, you can use a sequence of calls to the `drawLine()` method, or you can use the **`drawPolygon()` method** to draw complex shapes. The `drawPolygon()` method requires three arguments: two integer arrays and a single integer.

The first integer array holds a series of x-coordinate positions, and the second array holds a series of corresponding y-coordinate positions. These positions represent points that are connected to form the polygon. The third integer argument is the number of pairs of points you want to connect. If you don't want to connect all the points represented by the array values, you can assign this third argument integer a value that is smaller than the number of elements in each array. However, an error occurs if the third argument is a value higher than the available number of coordinate pairs.

For example, examine the code shown in Figure 16-20, which is a `JFrame` application that has one task: to draw a star-shaped polygon.

```
import javax.swing.*;
import java.awt.*;
public class JStar extends JFrame
{
   public void paint(Graphics gr)
   {
      super.paint(gr);
      int xPoints[] = {42, 52, 72, 52,  60, 40, 15,  28, 9,  32, 42};
      int yPoints[] = {38, 62, 68, 80, 105, 85, 102, 75, 58, 60, 38};
      gr.drawPolygon(xPoints, yPoints, xPoints.length);
   }
   public static void main(String[] args)
   {
      JStar frame = new JStar();
      frame.setSize(80, 150);
      frame.setVisible(true);
   }
}
```

**Figure 16-20** The `JStar` class

In the `JStar` program, two parallel arrays are assigned x- and y-coordinates. It is almost impossible to create a program like this without sketching the desired shape on a piece of graph paper to discover appropriate coordinate values. The `drawPolygon()` method uses the two arrays and the length of one of the arrays for the number of points. The program's output appears in Figure 16-21.

**Figure 16-21** Output of the `JStar` program

You can use the **`fillPolygon()` method** to draw a solid shape. The major difference between the `drawPolygon()` and `fillPolygon()` methods is that if the beginning and ending points used with the `fillPolygon()` method are not identical, the two endpoints are connected by a straight line before the polygon is filled with color.

Rather than providing the `fillPolygon()` method with three arguments, you can also create a `Polygon` object and pass the constructed object as a single argument to the `fillPolygon()` method. The `Polygon` constructor requires an array of x-coordinates, an array of y-coordinates, and a size. For example, you can create a filled polygon using the following statements:

```
Polygon someShape = new Polygon(xPoints, yPoints, xPoints.length);
gr.fillPolygon(someShape);
```

The `Polygon` class also has a default constructor, so you can instantiate an empty `Polygon` object (with no points) using the following statement:

```
Polygon someFutureShape = new Polygon();
```

Whether you use the default constructor or not, you can add points to a polygon after construction. For example, you might want to add points that are determined by user input or mathematical calculations. You use the **`addPoint()` method** in statements such as the following to add points to the polygon later:

```
someFutureShape.addPoint(100, 100);
someFutureShape.addPoint(150, 200);
someFutureShape.addPoint(50, 250);
```

Points can be added to a polygon indefinitely.

## Copying an Area

After you create a graphics image, you might want to create copies of the image. For example, you might want a company logo to appear several times in an application. Of course, you can redraw the picture, but you can also use the **`copyArea()` method** to copy any rectangular area to a new location. The `copyArea()` method requires six parameters:

- The x- and y-coordinates of the upper-left corner of the area to be copied
- The width and height of the area to be copied
- The horizontal and vertical displacement of the destination of the copy

For example, the following line of code causes a `Graphics` object named `gr` to copy an area 20 pixels wide by 30 pixels tall from the upper-left corner of your `JFrame` (coordinates 0, 0) to an area that begins 100 pixels to the right and 50 pixels down:

```
gr.copyArea(0, 0, 20, 30, 100, 50);
```

## Using the `paintComponent()` Method with `JPanel`s

When you create drawings on a `JPanel` (or other `JComponent`) instead of on a `JFrame`, you should use the `paintComponent()` method rather than the `paint()` method. A `JFrame`'s `paint()` method automatically calls `paintComponent()` for its components, but `JFrame` is not a child of `JComponent`, so it does not have its own `paintComponent()` method.

For example, Figure 16-22 shows a `JGraphicsPanel` class that overrides `JPanel`. Its constructor accepts a color to use as a background color. Its only method is a `paintComponent()` method that overrides the `paintComponent()` method in the `JPanel` class. The parameter `Graphics` object is passed to the parent class constructor, the drawing color is set to yellow, and two small circles are drawn.

```
import javax.swing.*;
import java.awt.*;
import java.awt.Color;
public class JGraphicsPanel extends JPanel
{
   public JGraphicsPanel(Color color)
   {
      setBackground(color);
   }
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      g.setColor(Color.YELLOW);
      g.fillOval(10, 5, 40, 40);
      g.fillOval(60, 5, 40, 40);
   }
}
```

**Figure 16-22** The `JGraphicsPanel` class

Figure 16-23 contains a program that adds two `JGraphicsPanel` objects to a `JFrame`—one with a blue background and the other with a red background. Figure 16-24 shows the output, which displays two `JPanels` that are placed side by side using a `GridLayout`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.Color;
public class JGraphicsPanelFrame extends JFrame
{
   Container con = null;
   JGraphicsPanel p1 = new JGraphicsPanel(Color.BLUE);
   JGraphicsPanel p2 = new JGraphicsPanel(Color.RED);

   public JGraphicsPanelFrame ()
   {
      con = this.getContentPane();
      con.setLayout(new GridLayout(2,1));
      con.add(p1);
      con.add(p2);
```

**Figure 16-23** The `JGraphicsPanelFrame` class *(continues)*

(*continued*)

```
        setSize(250, 250);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String args[])
    {
        JGraphicsPanelFrame app = new JGraphicsPanelFrame();
        app.setVisible(true);
        app.setSize(140, 140);
    }
}
```

**Figure 16-23** The `JGraphicsPanelFrame` class

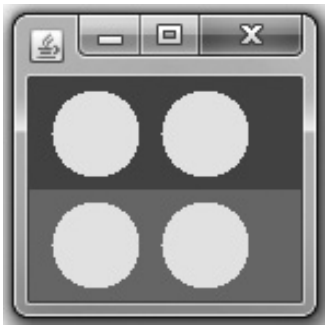

**Figure 16-24** Output of the `JGraphicsPanelFrame` program

Watch the video *Drawing Lines and Shapes*.

## TWO TRUTHS & A LIE

### Drawing Lines and Shapes

1. You can use the `drawLine()` method to draw a straight line between any two points on the screen.
2. You can use methods named `drawRect()`, `fillRect()`, `clearRect()`, `drawOval()`, and `fillOval()` to create a variety of shapes.
3. When you draw an arc, the zero-degree position is at 12 o'clock on an imaginary clock, and the 90-degree position is at three o'clock.

The false statement is #3. When you draw an arc, the zero-degree position is at three o'clock, and the degree values increase as you move counterclockwise in a 360-degree circle, so the 90-degree position is at 12 o'clock.

## You Do It

*Creating a Drawing*

Next, you add a simple line drawing to the `JDemoCreateGraphicsObject2` program. The drawing appears after the user clicks the `JButton` enough times to disable the `JButton`.

1. Open the **JDemoCreateGraphicsObject2** file, and immediately save it as **JDemoCreateGraphicsObject3.java**. Change the class name, constructor name, and two references in the `main()` method to match.
2. Replace the current `if...else` structure that tests whether `y` is less than `LIMIT` in the `actionPerformed()` method. Instead, use the following code, which tests the value of `y` and either draws the quote or disables the `JButton` and draws a picture. Set the drawing color to black, and create a simple drawing of a stick person that includes a head, torso, and two legs:

```
if(y < LIMIT)
   pen.drawString(movieQuote, x += GAP, y += GAP);
else
{
   moveButton.setEnabled(false);
   pen.setColor(Color.BLACK);
```

*(continues)*

*(continued)*

```
        pen.drawOval(50, 170, 70, 70);
        pen.drawLine(85, 240, 90, 280);
        pen.drawLine(90, 280, 60, 380);
        pen.drawLine(90, 280, 110, 380);
    }
```

3. Save the file, compile it, and execute it. After the movie quote moves to the `LIMIT` value, the `JButton` is disabled and the drawing appears, as shown in Figure 16-25.

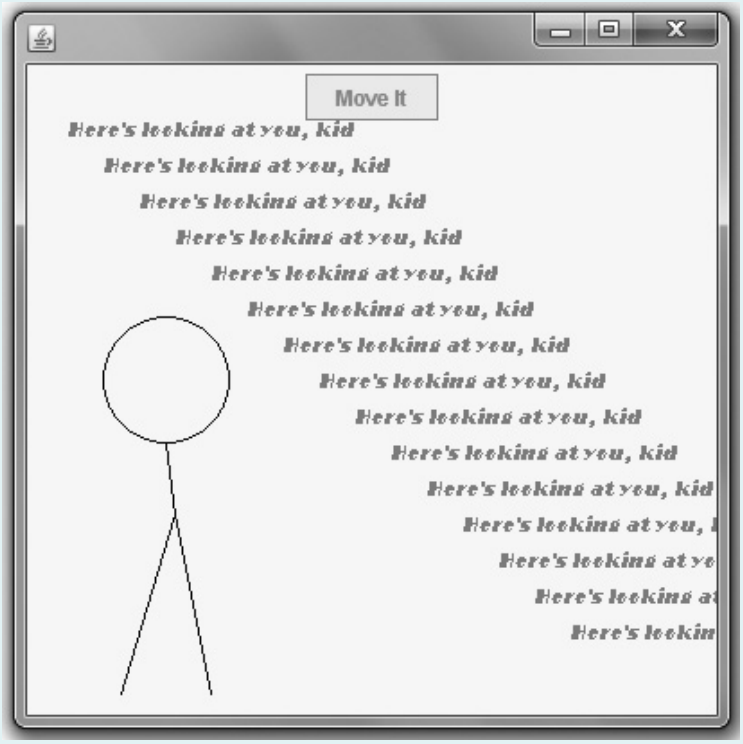


**Figure 16-25** The `JDemoCreateGraphicsObject3` program after the `JButton` is disabled

4. Close the application.
5. Modify the application so the drawing has more details, such as arms, feet, and a simple face. Save the revised application as **JDemoCreateGraphicsObject4.java**.

*(continues)*

*(continued)*

*Copying an Area*

Next, you learn how to copy an area containing a shape that you want to appear several times on a `JFrame`. By copying, you do not have to re-create the shape each time.

1. Open a new text file in your text editor, and then enter the beginning statements for a `JFrame` that uses the `copyArea()` method:

```
import javax.swing.*;
import java.awt.*;
import java.awt.Color;
public class JThreeStars extends JFrame
{
```

2. Add the following statements, which create a polygon in the shape of a star:

```
int xPoints[] = {42, 52, 72, 52,
   60, 40, 15, 28, 9, 32, 42};
int yPoints[] = {38, 62, 68, 80,
   105, 85, 102, 75, 58, 60, 38};
Polygon aStar = new Polygon(xPoints, yPoints, xPoints.length);
```

3. Add a constructor that sets the default close operation:

```
public JThreeStars()
{
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

4. Add the following `paint()` method, which sets a color, draws a star, and then draws two additional identical stars:

```
public void paint(Graphics star)
{
   super.paint(star);
   star.setColor(Color.BLUE);
   star.drawPolygon(aStar);
   star.copyArea(0, 0, 75, 105, 80, 40);
   star.copyArea(0, 0, 75, 105, 40, 150);
}
```

5. Add a `main()` method that instantiates a `JThreeStars` object and sets its size and visibility. Add a closing brace to end the class:

```
   public static void main(String[] args)
   {
      JThreeStars frame = new JThreeStars();
      frame.setSize(200, 300);
      frame.setVisible(true);
   }
}
```

*(continues)*

*(continued)*

6. Save the file as **JThreeStars.java**, and then compile the program. When you run the program, the output looks like Figure 16-26.

**Figure 16-26** Output of the `JThreeStars` program with one star

7. Close the frame to end the application.
8. Modify the program to add two more stars in any location you choose, save and compile the program, and confirm that the stars are copied to your desired locations.

## Learning More About Fonts

As you add more components in your GUI applications, positioning becomes increasingly important. In particular, when you draw `String`s using different fonts, it is difficult to place them correctly so they don't overlap, making them difficult or impossible to read. In addition, the number of available fonts varies greatly across operating systems, so even when you define a font using a string argument such as "Arial" or "Courier", you have no guarantee that the font will be available on every computer that runs your application. If a user's computer does not have the requested font loaded, Java chooses a default replacement font, so you can never be completely certain how your output will look. Fortunately, Java provides many useful methods for obtaining information about the fonts you use.

You can discover the fonts that are available on your system by using the **`getAvailableFontFamilyNames()` method**, which is part of the `GraphicsEnvironment` class defined in the `java.awt` package. The `GraphicsEnvironment` class describes the collection of `Font` objects and `GraphicsDevice` objects available to a Java application on a particular

platform. The `getAvailableFontFamilyNames()` method returns an array of `String` objects that are the names of available fonts. For example, the following statements declare a `GraphicsEnvironment` object named `ge`, and then use the object with the `getAvailableFontFamilyNames()` method to store the font names in a string array:



```
GraphicsEnvironment ge =
   GraphicsEnvironment.getLocalGraphicsEnvironment();
String[] fontnames = ge.getAvailableFontFamilyNames();
```

Notice in the preceding example that you can't instantiate the `GraphicsEnvironment` object directly. Instead, you must get a reference object to the current computer environment by calling the static `getLocalGraphicsEnvironment()` method. Figure 16-27 shows a `JFrame` that lists all the available font names on the computer on which the program was executed. After the `GraphicsEnvironment` object is created and the `getAvailableFontFamilyNames()` method is used to retrieve the array of font names, the names are displayed on the screen using a `for` loop in which the horizontal coordinate where each font `String` is drawn is increased by a fixed value so that five columns are displayed equally spaced across the `JFrame` surface. After five items are displayed, the horizontal coordinate is set back to 10 and the vertical coordinate is increased so that the next five-column row is displayed below the previous one. Typical output is shown in Figure 16-28.

```
import javax.swing.*;
import java.awt.*;
public class JFontList extends JFrame
{
   public void paint(Graphics gr)
   {
      super.paint(gr);
      int i, x, y = 40;
      final int VERTICAL_SPACE = 10;
      final int HORIZONTAL_SPACE = 160;
      GraphicsEnvironment ge =
         GraphicsEnvironment.getLocalGraphicsEnvironment();
      String[] fontnames = ge.getAvailableFontFamilyNames();
      for(i = 0; i < fontnames.length; i += 5)
      {
         x = 10;
         gr.setFont(new Font("Arial", Font.PLAIN, 10));
         gr.drawString(fontnames[i], x, y);
         if(i + 1 < fontnames.length)
            gr.drawString(fontnames[i + 1], x += HORIZONTAL_SPACE, y);
         if(i + 2 < fontnames.length)
            gr.drawString(fontnames[i + 2], x += HORIZONTAL_SPACE, y);
         if(i + 3 < fontnames.length)
            gr.drawString(fontnames[i + 3], x += HORIZONTAL_SPACE, y);
```

**Figure 16-27** The `JFontList` class *(continues)*

(*continued*)

```
            if(i + 4 < fontnames.length)
               gr.drawString(fontnames[i + 4], x += HORIZONTAL_SPACE, y);
            y = y + VERTICAL_SPACE;
         }
      }
      public static void main(String[] args)
      {
         JFontList frame = new JFontList();
         frame.setSize(820, 620);
         frame.setVisible(true);
      }
   }
```

**Figure 16-27** The `JFontList` class

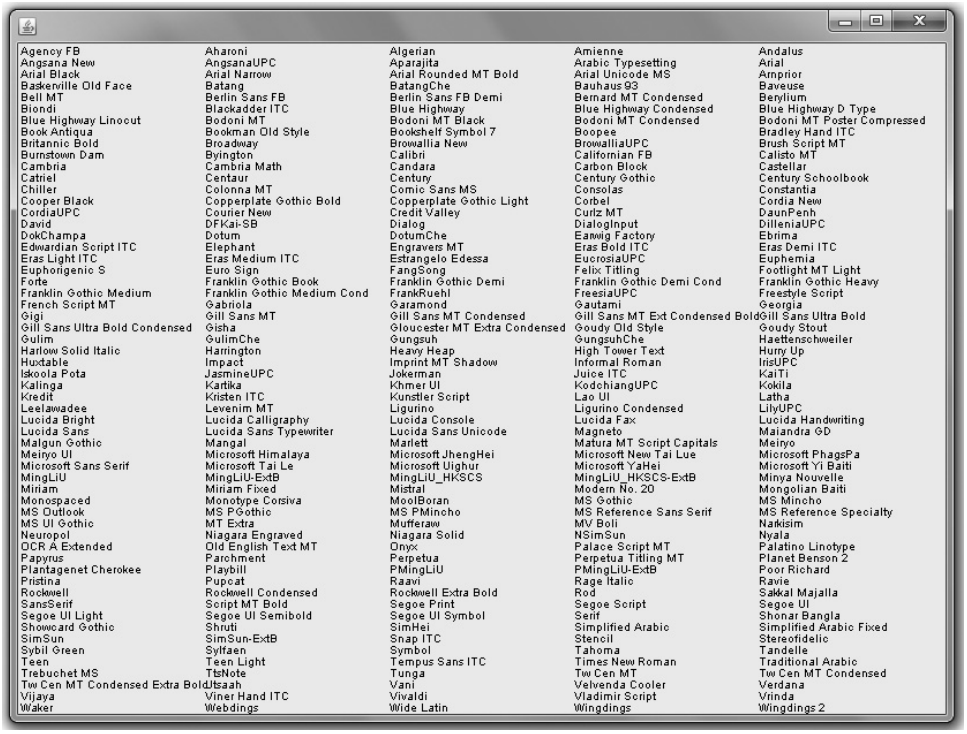


**Figure 16-28** Typical output of the `JFontList` program

## Discovering Screen Statistics Using the `Toolkit` Class



Frequently, before you can determine the best `Font` size to use, it is helpful to know statistics about the screen on which the `Font` will be displayed. For example, you can discover the resolution and screen size on your system by using the `getScreenResolution()` and `getScreenSize()` methods, which are part of the `Toolkit` class.

The **`getDefaultToolkit()` method** provides information about the system in use. The **`getScreenResolution()` method** returns the number of pixels as an integer. You can create a `Toolkit` object and get the screen resolution using the following code:

```
Toolkit tk = Toolkit.getDefaultToolkit();
int resolution = tk.getScreenResolution();
```

The `Dimension` class is useful for representing the width and height of a user interface component, such as a `JFrame` or a `JButton`. The `Dimension` class has three constructors:

- The `Dimension()` method creates an instance of `Dimension` with a width of 0 and a height of 0.
- `Dimension(Dimension d)` creates an instance of `Dimension` whose width and height are the same as for the specified dimension.
- `Dimension(int width, int height)` constructs a `Dimension` and initializes it to the specified width and height.

The **`getScreenSize()` method**, a member of the `Toolkit` object, returns an object of type `Dimension`, which specifies the width and height of the screen in pixels. Knowing the number of pixels for the width and height of your display is useful to set the coordinates for the position, width, and height of a window. For example, the following code stores the width and height of a screen in separate variables:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Dimension screen = tk.getScreenSize();
int width = screen.width;
int height = screen.height;
```

## Discovering Font Statistics

Typesetters and desktop publishers measure the height of every font in three parts: ascent, descent, and leading. **Ascent** is the height of an uppercase character from a baseline to the top of the character. **Descent** measures the part of characters that "hang below" the baseline, such as the tails on the lowercase letters *g* and *j*. **Leading** (pronounced *ledding*) is the amount of space between the bottom of the descent of one line and the top of the characters in the successive line of type. The **height of a font** is the sum of the leading, ascent, and descent. Figure 16-29 labels each of these measurements.

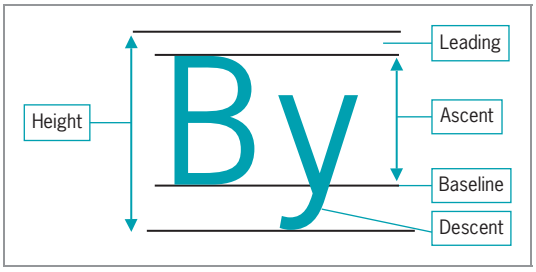


**Figure 16-29** Parts of a font's height

You can discover a font's statistics by first using the `Graphics` class **`getFontMetrics()` method** to return a `FontMetrics` object, and then by using one of the following `FontMetrics` class methods with the object to return one of a `Font`'s statistics:

- `public int getLeading()`
- `public int getAscent()`
- `public int getDescent()`
- `public int getHeight()`

Another method, `getLineMetrics()`, is more complicated to use but returns similar font statistics. For more details, see the Java Web site.

Each of these methods returns an integer value representing the font size in points (one point measures 1/72 of an inch) of the requested portion of the `Font` object. For example, if you define a `Font` object named `myFont` and a `Graphics` object named `paintBrush`, you can set the current font for the `Graphics` object by using the following statements:

```
paintBrush.setFont(myFont);
int heightOfFont = paintBrush.getFontMetrics().getHeight();
```

When you define a `Font` object, you use point size. However, when you use the `FontMetrics get` methods, the sizes are returned in pixels.

Then the `heightOfFont` variable holds the total height of `myFont` characters.

A practical use for discovering the height of a font is to space `Strings` correctly as you display them. For example, instead of placing every `String` in a series vertically equidistant from the previous `String` with a statement such as the following:

```
pen.drawString("Some string", x, y += INCREASE);
```

(where INCREASE has been defined as a constant), you can make the actual increase in the vertical position dependent on the font. If you code the following, you are assured that each String has enough room, regardless of which font is currently in use by the Graphics pen object:



```
pen.drawString("Some string",
   x, y += pen.getFontMetrics().getHeight());
```

When you create a String, you know how many characters are in the String. However, you cannot be certain which font Java will use or substitute, and because fonts have different measurements, it is difficult to know the exact width of the String that appears in a JFrame. Fortunately, the FontMetrics class contains a **stringWidth() method** that returns the integer width of a String. As an argument, the stringWidth() method requires the name of a String. For example, if you create a String named myString, you can retrieve the width of myString with the following code:

```
int width = gr.getFontMetrics().stringWidth(myString);
```

Watch the video *Font Methods*.

## TWO TRUTHS & A LIE

### Learning More About Fonts

1. Java is widely used partly because its fonts are guaranteed to look the same on all computers.
2. You can discover the resolution and screen size on your system by using the getScreenResolution() and getScreenSize() methods, which are part of the Toolkit class.
3. Ascent is the height of an uppercase character from a baseline to the top of the character, and descent measures the part of characters that "hang below" the baseline, such as the tail on the lowercase letter *y*.

The false statement is #1. If a user's computer does not have a font you have requested, Java chooses a default replacement font, so you can never be completely certain how your output will look.

## You Do It

### *Using `FontMetrics` Methods to Compare Fonts*

Next, you write a program to demonstrate `FontMetrics` methods. You will create three `Font` objects and display their metrics.

1. Open a new text file in your text editor, and then enter the first few lines of the `JDemoFontMetrics` program:

```
import javax.swing.*;
import java.awt.*;
public class JDemoFontMetrics extends JFrame
{
```

2. Type the following code to create a `String` and a few fonts to use for demonstration purposes:

```
String movieQuote =
   new String("Go ahead, make my day");
Font courierItalic = new Font("Courier New", Font.ITALIC, 16),
   timesPlain = new Font("Times New Roman", Font.PLAIN, 16),
   scriptBold = new Font("Freestyle Script", Font.BOLD, 16);
```

3. Add the following code to define four integer variables to hold the four font measurements, and two integer variables to hold the current horizontal and vertical output positions within the `JFrame`:

```
int ascent, descent, height, leading;
int x = 20, y = 50;
```

4. Within the `JFrame`, you will draw `String`s positioned 40 pixels apart vertically. After each of those `String`s, the `String`s that hold the statistics will be 15 pixels apart. Type the following statements to create constants to hold these vertical increase values:

```
final int INCREASE_SMALL = 15;
final int INCREASE_LARGE = 40;
```

5. Add a constructor as follows:

```
public JDemoFontMetrics()
{
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

6. Add the following statements to start writing a `paint()` method. Within the method, you set the `Font` to `courierItalic`, draw the `phrase String` to show a working example of the font, and then call a `displayMetrics()` method

*(continues)*

*(continued)*

that you will write in Step 7. Pass the `Graphics` object to the `displayMetrics()` method, so the method can discover the sizes associated with the current font. Perform the same three steps using the `timesPlain` and `scriptBold` fonts.

```
public void paint(Graphics pen)
{
   super.paint(pen);
   pen.setFont(courierItalic);
   pen.drawString(moviequote, x, y);
   displayMetrics(pen);
   pen.setFont(timesPlain);
   pen.drawString(moviequote, x, y += INCREASE_LARGE);
   displayMetrics(pen);
   pen.setFont(scriptBold);
   pen.drawString(moviequote, x, y += INCREASE_LARGE);
   displayMetrics(pen);
}
```

7. Next, add the header and opening curly brace for the `displayMetrics()` method. The method will receive a `Graphics` object from the `paint()` method. Add the following statements to call the four `getFontMetrics()` methods to obtain values for the leading, ascent, descent, and height variables:

```
public void displayMetrics(Graphics metrics)
{
   leading = metrics.getFontMetrics().getLeading();
   ascent = metrics.getFontMetrics().getAscent();
   descent = metrics.getFontMetrics().getDescent();
   height = metrics.getFontMetrics().getHeight();
```

8. Add the following four `drawString()` statements to display the values. Use the expression `y += INCREASE_SMALL` to change the vertical position of each `String` by the constant.

```
   metrics.drawString("Leading is " + leading,
      x, y += INCREASE_SMALL);
   metrics.drawString("Ascent is " + ascent,
      x, y += INCREASE_SMALL);
   metrics.drawString("Descent is " + descent,
      x, y += INCREASE_SMALL);
   metrics.drawString("Height is " + height,
      x, y += INCREASE_SMALL);
}
```

*(continues)*

*(continued)*

9. Add a `main()` method, and include a closing curly brace for the class:

```
    public static void main(String[] args)
    {
       JDemoFontMetrics frame = new JDemoFontMetrics();
       frame.setSize(250, 350);
       frame.setVisible(true);
    }
}
```

10. Save the file as **JDemoFontMetrics.java**, and then compile it. When you run the program, the output should look like Figure 16-30. Notice that even though each `Font` object was constructed with a size of 16, the individual statistics vary for each `Font` object.

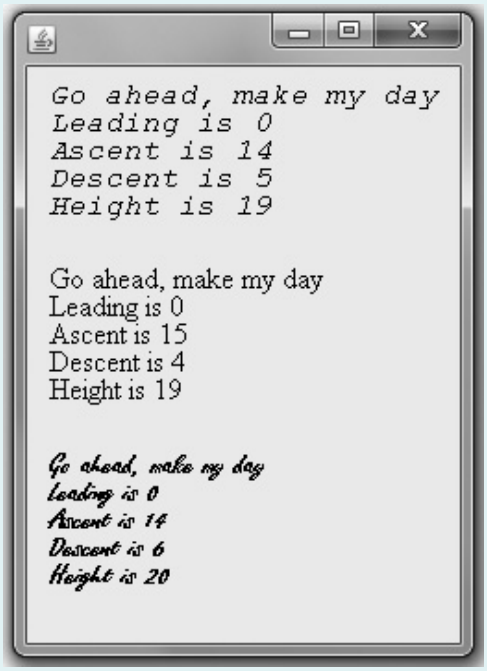


**Figure 16-30** Output of the `JDemoFontMetrics` program

11. Close the frame to end the program.

*(continues)*

*(continued)*

*Using `FontMetrics` Methods to Place a Border Around a `String`*

Next, you use the `FontMetrics` methods to draw a rectangle around a `String`. Instead of guessing at appropriate pixel positions, you can use the height and width of the `String` to create a box with borders placed symmetrically around the `String`.

1. Open a new file in your text editor, and enter the first few lines of a `JBoxAround JFrame`:

```
import javax.swing.*;
import java.awt.*;
public class JBoxAround extends JFrame
{
```

2. Enter the following statements to add a `String`, a `Font`, and variables to hold the font metrics and x- and y-coordinates:

```
String movieQuote =
   new String("An offer he can't refuse");
Font serifItalic = new Font("Serif", Font.ITALIC, 20);
int leading, ascent, height, width;
int x = 40, y = 60;
```

3. Create the following named constant that holds a number indicating the dimensions in pixels of the rectangle that you draw around the `String`:

```
static final int BORDER = 5;
```

4. Add a constructor as follows:

```
public JBoxAround()
{
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

5. Add the following `paint()` method, which sets the font, draws the `String`, and obtains the font metrics:

```
public void paint(Graphics gr)
{
   super.paint(gr);
   gr.setFont(serifItalic);
   gr.drawString(movieQuote, x, y);
   leading = gr.getFontMetrics().getLeading();
   ascent = gr.getFontMetrics().getAscent();
   height = gr.getFontMetrics().getHeight();
   width = gr.getFontMetrics().stringWidth(movieQuote);
```

*(continues)*

*(continued)*

6. Draw a rectangle around the `String` using the following `drawRect()` method. In Figure 16-31, the x- and y-coordinates of the upper-left edge are set at `x - BORDER, y - (ascent + leading + BORDER)`. The proper width and height are then determined to draw a uniform rectangle around the string.

   The values of the x- and y-coordinates used in the `drawString()` method indicate the left side of the baseline of the first character in the `String`. You want to position the upper-left corner of the rectangle five pixels to the left of the `String`, so the first argument to `drawRect()` is five less than `x`, or `x - BORDER`. The second argument to `drawRect()` is the y-coordinate of the `String` minus the ascent of the `String`, minus the leading of the `String`, minus five, or `y - (ascent + leading + BORDER)`. The final two arguments to `drawRect()` are the width and height of the rectangle. The width is the `String`'s width plus five pixels on the left and five pixels on the right. The height of the rectangle is the `String`'s height, plus five pixels above the `String` and five pixels below the `String`.

```
      gr.drawRect(x - BORDER, y - (ascent + leading + BORDER),
         width + 2 * BORDER, height + 2 * BORDER);
   }
```

7. Add the following `main()` method and a closing brace for the class:

```
      public static void main(String[] args)
      {
         JBoxAround frame = new JBoxAround();
         frame.setSize(330, 100);
         frame.setVisible(true);
      }
   }
```

8. Save the file as **JBoxAround.java**. Compile and execute it. Your output should look like Figure 16-31.

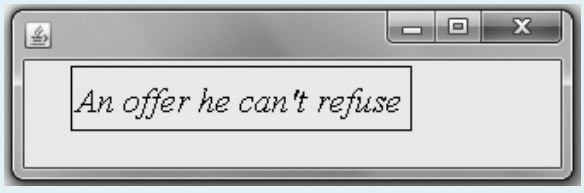


**Figure 16-31** Output of the `JBoxAround` program

9. Experiment with changing the contents of the `String`, the `x` and `y` starting coordinates, and the value of the `BORDER` constant. Confirm that the rectangle is drawn symmetrically around any `String` object.

# Drawing with Java 2D Graphics



Drawing operations earlier in this chapter were called using a `Graphics` object—either an automatically generated one that was passed to the `paint()` method or one the programmer instantiated. In addition, you can call drawing operations using an object of the **`Graphics2D` class**. The advantage of using Java 2D objects is the higher-quality, two-dimensional (2D) graphics, images, and text they provide.

Features of some of the 2D classes include:

- Fill patterns, such as gradients
- Strokes that define the width and style of a drawing stroke
- Anti-aliasing, a graphics technique for producing smoother screen graphics

`Graphics2D` is found in the `java.awt` package. A `Graphics2D` object is produced by casting, or converting and promoting, a `Graphics` object. For example, in a `paint()` method that automatically receives a `Graphics` object, you can cast the object to a `Graphics2D` object using the following code to start the method:

```
public void paint(Graphics pen)
{
   Graphics2D newpen = (Graphics2D)pen;
```

The process of drawing with Java 2D objects includes:

- Specifying the rendering attributes
- Setting a drawing stroke
- Creating objects to draw

## Specifying the Rendering Attributes

The first step in drawing a 2D object is to specify how a drawn object is rendered. Whereas drawings that are not 2D can only use the attribute `Color`, with 2D you can designate other attributes, such as line width and fill patterns. You specify 2D colors by using the `setColor()` method, which works like the `Graphics` method of the same name. Using a `Graphics2D` object, you can set the color to black using the following code:

```
gr2D.setColor(Color.BLACK);
```

**Fill patterns** control how a drawing object is filled in. In addition to using a solid color, 2D fill patterns can be a gradient fill, a texture, or even a pattern that you devise. A fill pattern is created by using the `setPaint()` method of `Graphics2D` with a fill pattern object as the only argument. Classes from which you can construct a fill pattern include `Color`, `TexturePaint`, and `GradientPaint`.

A **gradient fill** is a gradual shift from one color at one coordinate point to a different color at a second coordinate point. If the color shift occurs once between the points—for example, slowly changing from yellow to red—you are using an **acyclic gradient**, one that does not

cycle between the colors. If the shift occurs repeatedly, such as from yellow to red and back to yellow again, you are using a **cyclic gradient**, one that does cycle between the colors.

Figure 16-32 shows an application that demonstrates acyclic and cyclic gradient fills. The first shaded `setPaint()` method call sets a gradient that begins at coordinates 20, 40 in `LIGHT_GRAY` and ends at coordinates 180, 100 in `DARK_GRAY`. The last argument to the `GradientPaint()` constructor is `false`, indicating an acyclic gradient. After the `Graphics2D` object's paint is applied, a filled rectangle is drawn over the same area. These statements produce the rectangle on the left in Figure 16-33, which gradually shifts from light gray to dark gray, moving down and to the right. The second shaded `setPaint()` statement in Figure 16-32 establishes a new gradient beginning farther to the right. In this statement, the final argument to `GradientPaint()` is `true`, creating a cyclic gradient. As you can see on the right side in Figure 16-33, this rectangle's shading changes gradually across its surface.

Later in this chapter, you will learn about the `Rectangle2D.Double` class used to create the rectangles in this application.

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
import java.awt.Color;
public class JGradient extends JFrame
{
   public void paint(Graphics gr)
   {
      super.paint(gr);
      int x = 20, y = 40, x2 = 180, y2 = 100;
      Graphics2D gr2D = (Graphics2D)gr;
      gr2D.setPaint(new GradientPaint(x, y, Color.LIGHT_GRAY,
         x2, y2, Color.DARK_GRAY, false));
      gr2D.fill(new Rectangle2D.Double(x, y, x2, y2));
      x = 210;
      gr2D.setPaint(new GradientPaint(x, y, Color.LIGHT_GRAY,
         x2, y2, Color.DARK_GRAY, true));
      gr2D.fill(new Rectangle2D.Double(x, y, x2, y2));
   }
   public static void main(String[] args)
   {
      JGradient frame = new JGradient();
      frame.setSize(440, 180);
      frame.setVisible(true);
   }
}
```

**Figure 16-32** The `JGradient` class

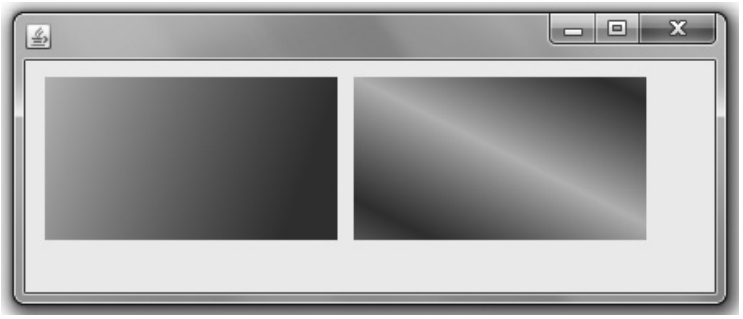

**Figure 16-33** Output of the `JGradient` application

## Setting a Drawing Stroke

All lines in non-2D graphics operations are drawn as solid, with square ends and a line width of one pixel. With the 2D methods, the drawing line is a **stroke**, which represents a single movement as if you were using a drawing tool, such as a pen or a pencil. In Java 2D, you can change a stroke's width using the **`setStroke()` method**. `Stroke` is actually an interface; the class that defines line types and implements the `Stroke` interface is named **`BasicStroke`**. A `BasicStroke` constructor takes three arguments:

- A `float` value representing the line width
- An `int` value determining the type of cap decoration at the end of a line
- An `int` value determining the style of juncture between two line segments

`BasicStroke` class variables determine the endcap and juncture style arguments. **Endcap styles** apply to the ends of lines that do not join with other lines, and include `CAP_BUTT`, `CAP_ROUND`, and `CAP_SQUARE`. **Juncture styles**, for lines that join, include `JOIN_MITER`, `JOIN_ROUND`, and `JOIN_BEVEL`.

The following statements create a `BasicStroke` object and make it the current stroke:

```
BasicStroke aLine = new BasicStroke(1.0f,
   BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
```

Figure 16-34 shows a program that draws a rectangle using a very wide stroke.

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
public class JStroke extends JFrame
{
   public void paint(Graphics gr)
   {
      super.paint(gr);
      Graphics2D gr2D = (Graphics2D)gr;
      BasicStroke aStroke = new BasicStroke(15.0f,
         BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
      gr2D.setStroke(aStroke);
      gr2D.draw(new Rectangle2D.Double(40, 40, 100, 100));
   }
   public static void main(String[] args)
   {
      JStroke frame = new JStroke();
      frame.setSize(180, 180);
      frame.setVisible(true);
   }
}
```

**Figure 16-34** The `JStroke` class

The shaded statement in the `JStroke` class sets the `BasicStroke` width to 15 pixels using round endcap and juncture parameters. Notice that the line width value is followed by an `f`, making the value a `float` instead of a `double`. Figure 16-35 shows the drawn rectangle.

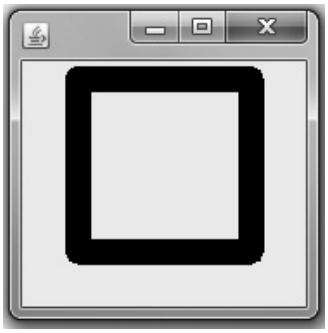

**Figure 16-35** Output of the `JStroke` program

## Creating Objects to Draw

After you have created a `Graphics2D` object and specified the rendering attributes, you can create different objects to draw. Objects that are drawn in Java 2D are first created by defining

them as geometric shapes using the `java.awt.geom` package classes. You can define the shape of lines, rectangles, ovals, and arcs; after you define the shape, you use it as an argument to the `draw()` or `fill()` methods. The `Graphics2D` class does not have different methods for each shape you can draw.



## Lines

Lines are created using the `Line2D.Float` class or the `Line2D.Double` class. Each of these classes has a constructor that takes four arguments, which are the x- and y-coordinates of the line endpoints. For example, to create a line from the endpoint 60, 5 to the endpoint 13, 28, you could write the following:

```
Line2D.Float line = new Line2D.Float(60F, 5F, 13F, 28F);
```

It also is possible to create lines based on points. You can use the `Point2D.Float` or `Point2D.Double` class to create points that have both x- and y-coordinates. For example, you could create two `Point2D.Float` points using the following code:

```
Point2D.Float pos1 = new Point2D.Float(60F, 5F);
Point2D.Float pos2 = new Point2D.Float(13F, 28F);
```

Then the code to create a line might be:

```
Line2D.Float line = new Line2D.Float(pos1, pos2);
```

## Rectangles

You can create rectangles by using a `Rectangle2D.Float` or a `Rectangle2D.Double` class. As with the `Line` and `Point` classes, these two classes are distinguished by the type of argument used to call their constructors: `float` or `double`. Both `Rectangle2D.Float` and `Rectangle2D.Double` can be created using four arguments representing the x-coordinate, y-coordinate, width, and height. For example, the following code creates a `Rectangle2D.Float` object named `rect` at 10, 10 with a width of 50 and height of 40:

```
Rectangle2D.Float rect = new Rectangle2D.Float(10F, 10F, 50F, 40F);
```

## Ovals

You can create `Oval` objects with the `Ellipse2D.Float` or `Ellipse2D.Double` class. The `Ellipse2D.Float` constructor requires four arguments representing the x-coordinate, y-coordinate, width, and height. The following code creates an `Ellipse2D.Float` object named `ell` at 10, 73 with a width of 40 and height of 20:

```
Ellipse2D.Float ell = new Ellipse2D.Float(10F, 73F, 40F, 20F);
```

## Arcs

You can create arcs with the `Arc2D.Float` or `Arc2D.Double` class. The `Arc2D.Float` constructor takes seven arguments. The first four arguments represent the x-coordinate, y-coordinate, width, and height that apply to the ellipse of which the arc is a part. The remaining three arguments are as follows:

- The starting position of the arc
- The number of degrees it travels
- An integer field indicating how it is closed

The starting position is expressed in degrees in the same way as in the `Graphics` class `drawArc()` method; for example, 0 is the three o'clock position. The number of degrees traveled by the arc is specified in a counterclockwise direction using positive numbers. The final argument uses one of the three class fields:

- `Arc2D.PIE` connects the arc to the center of an ellipse and looks like a pie slice.
- `Arc2D.CHORD` connects the arc's endpoints with a straight line.
- `Arc2D.OPEN` is an unclosed arc.

To create an `Arc2D.Float` object named `ac` at 10, 133 with a width of 30 and height of 33, a starting degree of 30, 120 degrees traveled, and using the class variable `Arc2D.PIE`, you use the following statement:

```
Arc2D.Float ac = new Arc2D.Float(10,133,30,33,30,120,Arc2D.PIE);
```

## Polygons

You create a `Polygon` object by defining movements from one point to another. The movement that creates a polygon is a `GeneralPath` object; the `GeneralPath` class is found in the `java.awt.geom` package.

- The statement `GeneralPath pol = new GeneralPath();` creates a `GeneralPath` object named `pol`.
- The `moveTo()` method of `GeneralPath` is used to create the beginning point of the polygon. Thus, the statement `pol.moveTo(10F, 193F);` starts the polygon named `pol` at the coordinates 10, 193.
- The `lineTo()` method is used to create a line that ends at a new point. The statement `pol.lineTo(25F, 183F);` creates a second point using the arguments of 25 and 183 as the x- and y-coordinates of the new point.
- The statement `pol.lineTo(100F, 223F);` creates a third point. The `lineTo()` method can be used to connect the current point to the original point. Alternatively, you can use the `closePath()` method without any arguments.

## TWO TRUTHS & A LIE

### Drawing with Java 2D Graphics

1. The advantage of using Java 2D objects is the higher-quality, 2D graphics, images, and text they provide.
2. With Java's 2D graphics, you can designate attributes such as color, line width, and fill patterns.
3. With Java's 2D methods, the drawing line is a brush that represents a single movement as if you were using a drawing tool, such as a pen or a pencil.

The false statement is #3. With Java's 2D methods, the drawing line is a stroke that represents a single movement as if you were using a drawing tool, such as a pen or a pencil.

## You Do It

*Using Drawing Strokes*

Next, you create a line with a drawing stroke to illustrate how it can have different end types and juncture types where lines intersect.

1. Open a new file in your text editor, and then enter the first few lines of a `J2DLine JFrame`. (Note that you are importing the `java.awt.geom` package.)

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
public class J2DLine extends JFrame
{
```

2. Add a constructor:

```
public J2DLine()
{
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

3. Enter the following statements to create a `paint()` method, create a `Graphics` environment `gr`, and cast the `Graphics` environment to a `Graphics2D` environment `gr2D`. Create x- and y-points with the `Point2D.Float` class.

*(continues)*

*(continued)*

```
public void paint(Graphics gr)
{
   super.paint(gr);
   Graphics2D gr2D = (Graphics2D)gr;
   Point2D.Float pos1 = new Point2D.Float(80, 50);
   Point2D.Float pos2 = new Point2D.Float(20, 100);
```

4. Create a `BasicStroke` object, and then create a drawing stroke named `aStroke`. Note that the line width is set to 15 pixels, and the endcap style and juncture style are set to `CAP_ROUND` and `JOIN_MITER`, respectively.

```
BasicStroke aStroke = new BasicStroke(15.0f,
   BasicStroke.CAP_ROUND, BasicStroke.JOIN_MITER);
```

5. Add the following code to create a line between the points `pos1` and `pos2`, and draw the line:

```
   gr2D.setStroke(aStroke);
   Line2D.Float line = new Line2D.Float(pos1, pos2);
   gr2D.draw(line);
}
```

6. Add a `main()` method and the closing curly brace for the class:

```
   public static void main(String[] args)
   {
      J2DLine frame = new J2DLine();
      frame.setSize(100, 120);
      frame.setVisible(true);
   }
}
```

7. Save the file as **J2DLine.java**, and then compile and execute it. Your output should look like Figure 16-36.

**Figure 16-36** Output of the `J2DLine` program

*(continues)*

(continued)

8. Experiment by making the `JFrame` size larger and adding more lines to create an interesting design.



*Working with Shapes*

Next, you use the Java 2D drawing object types to create a `JFrame` that illustrates sample rectangles, ovals, arcs, and polygons.

1. Open a new file in your text editor, and then enter the first few lines of a `JShapes2D JFrame`:

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
public class JShapes2D extends JFrame
{
```

2. Add a constructor that sets the default close operation as follows:

```
public JShapes2D()
{
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

3. Enter the following statements to create a `paint()` method, create a `Graphics` environment `gr`, and cast the `Graphics` environment to a `Graphics2D` environment `gr2D`:

```
public void paint(Graphics gr)
{
   super.paint(gr);
   Graphics2D gr2D = (Graphics2D)gr;
```

4. Create two `Rectangle2D.Float` objects named `rect` and `rect2`. Draw the `rect` object and fill the `rect2` object:

```
Rectangle2D.Float rect =
   new Rectangle2D.Float(20F, 40F, 40F, 40F);
Rectangle2D.Float rect2 =
   new Rectangle2D.Float(20F, 90F, 40F, 40F);
gr2D.draw(rect);
gr2D.fill(rect2);
```

(continues)

*(continued)*

5. Create two `Ellipse2D.Float` objects named `ellipse` and `ellipse2`. Draw the `ellipse` object and fill the `ellipse2` object:

```
Ellipse2D.Float ellipse = new
   Ellipse2D.Float(20F, 140F, 40F, 40F);
Ellipse2D.Float ellipse2 = new
   Ellipse2D.Float(20F, 190F, 40F, 40F);
gr2D.draw(ellipse);
gr2D.fill(ellipse2);
```

6. Create two `Arc2D.Float` objects named `ac` and `ac2`. Draw the `ac` object and fill the `ac2` object:

```
Arc2D.Float ac = new
   Arc2D.Float(20, 240, 50, 50, 30, 120, Arc2D.PIE);
Arc2D.Float ac2 = new
   Arc2D.Float(20, 290, 50, 50, 30, 120, Arc2D.PIE);
gr2D.draw(ac);
gr2D.fill(ac2);
```

7. Create a new `GeneralPath` object named `pol`. Set the starting point of the polygon and create two additional points. Use the `closePath()` method to close the polygon by connecting the current point to the starting point. Draw the `pol` object, and then end the method with a curly brace:

```
   GeneralPath pol = new GeneralPath();
   pol.moveTo(20F,320F);
   pol.lineTo(40F,380F);
   pol.lineTo(100F,400F);
   pol.closePath();
   gr2D.draw(pol);
}
```

8. Add a `main()` method and the final curly brace for the class:

```
   public static void main(String[] args)
   {
      JShapes2D frame = new JShapes2D();
      frame.setSize(100, 420);
      frame.setVisible(true);
   }
}
```

9. Save the file as **JShapes2D.java**, and compile and execute the program. Your output should look like Figure 16-37. When you are ready, close the window, and then experiment with making changes to the program to produce different shapes.

*(continues)*

*(continued)*

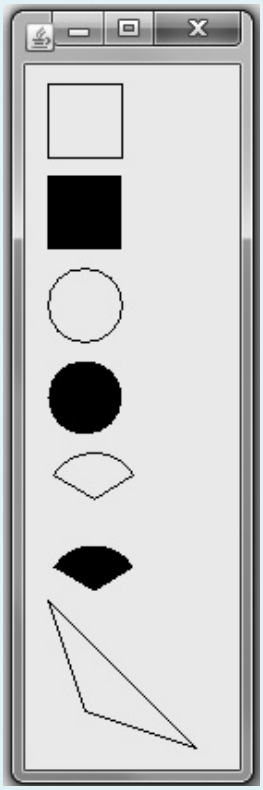

**Figure 16-37** Output of the `JShapes2D` program

## Don't Do It

- Don't forget to call `super.paint()` as the first statement in the `paint()` method when you write a class that extends `JFrame`. Failing to do so can cause odd results, especially when you combine GUI widgets with graphics.
- Don't forget that the `setLocation()` method works correctly only when it is used after the layout manager has finished positioning all the application's components (or in cases where no layout manager is functioning).

- Don't forget that the lower-left corner of a `String` is placed at the coordinates used when you call `drawString()`.
- Don't forget to use `paintComponent()` rather than `paint()` when creating graphics on a `JPanel`.



# Key Terms

To **rerender** a drawing is to repaint or redisplay it.

**Painting** is the act of redisplaying a surface.

**System-triggered painting** operations occur when the system requests a component to render its contents.

**Application-triggered painting** operations occur when the internal state of a component has changed.

The **`paint()` method** runs when Java displays a screen; you can write your own `paint()` method to override the automatically supplied one whenever you want to paint graphics such as shapes on the screen.

The **`Graphics` class** is an abstract class that descends directly from `Object` and holds data about graphics operations and methods for drawing shapes, text, and images.

The **`repaint()` method** updates a window when it contains new images.

The **`setLocation()` method** allows you to place a component at a specific location within a `JFrame`'s content pane.

The **`drawString()` method** allows you to draw a `String` in a `JFrame` or other component.

The **`drawLine()` method** draws a straight line between any two points on the screen.

The **`drawRect()` method** draws the outline of a rectangle.

The **`fillRect()` method** draws a solid, or filled, rectangle.

The **`clearRect()` method** draws a rectangle using the background color to create what appears to be an empty or "clear" rectangle.

The **`drawRoundRect()` method** draws rectangles with rounded corners.

An **arc** is a portion of a circle.

The **`draw3DRect()` method** draws a rectangle that appears to have "shadowing" on two of its edges—the effect is that of a rectangle that is lit from the upper-left corner and slightly raised or slightly lowered.

The **`fill3DRect()` method** creates filled, 3D rectangles.

The **`drawOval()` method** draws an oval.

The **`fillOval()` method** draws a solid, filled oval.

The **`drawArc()` method** draws an arc.

The **`fillArc()` method** creates a solid arc.

The **`drawPolygon()` method** draws complex shapes.

The **`fillPolygon()` method** draws a solid complex shape.

The **`addPoint()` method** adds points to a `Polygon` object.

The **`copyArea()` method** copies any rectangular area to a new location.

The **`getAvailableFontFamilyNames()` method** returns the fonts that are available on your system.

The **`getDefaultToolkit()` method** provides information about the system in use.

The **`getScreenResolution()` method** returns the screen resolution on the current system.

The **`getScreenSize()` method** returns the screen size as a `Dimension` object.

**Ascent** is one of three measures of a `Font`'s height; it is the height of an uppercase character from a baseline to the top of the character.

**Descent** is one of three measures of a `Font`'s height; it measures the part of characters that "hang below" the baseline, such as the tails on the lowercase letters *g* and *j*.

**Leading** is one of three measures of a `Font`'s height; it is the amount of space between the bottom of the descent of one line and the top of the characters in the successive line of type.

The **height of a font** is the sum of its leading, ascent, and descent.

The **`getFontMetrics()` method** in the `Graphics` class returns a `FontMetrics` object; with it you can discover many characteristics of a `Font` object.

The **`stringWidth()` method** in the `FontMetrics` class contains the integer width of a `String`.

The **`Graphics2D` class** provides tools for 2D drawing.

**Fill patterns** control how a drawing object is filled in.

A **gradient fill** is a gradual shift from one color at one coordinate point to a different color at a second coordinate point.

An **acyclic gradient** is a fill pattern in which a color shift occurs once between two points.

A **cyclic gradient** is a fill pattern in which a shift between colors occurs repeatedly between two points.

A **stroke** is a line-drawing feature in Java 2D that represents a single movement as if you were using a drawing tool, such as a pen or a pencil.

The **`setStroke()` method** changes a stroke's width in Java 2D.

**BasicStroke** is the class that defines line types and implements the Stroke interface.

**Endcap styles** apply to the ends of lines that do not join with other lines, and include CAP_BUTT, CAP_ROUND, and CAP_SQUARE.

**Juncture styles**, for lines that join, include JOIN_MITER, JOIN_ROUND, and JOIN_BEVEL.



## Chapter Summary

- Painting operations can be system triggered or application triggered. Painting operations are performed by a Component's paint() method, which takes a Graphics argument that renders output. You override the paint() method in your programs when you want specific actions to take place when components must be rendered. The setLocation() method allows you to place a component at a specific location within a JFrame's content pane.
- The drawString() method allows you to draw a String. The method requires three arguments: a String, an x-axis coordinate, and a y-axis coordinate. The drawString() method is a member of the Graphics class, so you need to use a Graphics object to call it. You can improve the appearance of strings drawn using Graphics objects by using the setFont() and setColor() methods.
- Java provides you with several methods for drawing a variety of lines and geometric shapes, such as drawLine(), drawRect(), drawOval(), drawPolygon(), and others. You can also use the copyArea() method to copy any rectangular area to a new location.
- If a user's computer does not have a requested font, Java chooses a default replacement font. You can discover the fonts that are available on your system by using the getAvailableFontFamilyNames() method, which is part of the GraphicsEnvironment class. This class describes the collection of Font objects and GraphicsDevice objects available to a Java application on a particular platform. You can discover the resolution and screen size on your system by using the getScreenResolution() and getScreenSize() methods, which are part of the Toolkit class. The height of every font is the sum of three parts: ascent, descent, and leading.
- The advantage to using Graphics2D objects is the higher-quality 2D graphics, images, and text they provide. With 2D you can designate attributes such as line width and fill patterns.

## Review Questions

1. Repainting of a visible surface is triggered by _____.
   a. the operating system
   b. the application
   c. either of these
   d. none of these

2. The method that calls the `paint()` method for you is _____.

   a. `callPaint()`
   b. `repaint()`
   c. `requestPaint()`
   d. `draw()`

3. The `paint()` method header requires a(n) _____ argument.

   a. `void`
   b. `integer`
   c. `String`
   d. `Graphics`

4. The `setLocation()` method _____.

   a. is used to position a `JFrame` on the screen
   b. is used to set regional and national preferences for an application
   c. takes two integer arguments that represent position coordinates
   d. must be used with every component placed on a `JFrame`

5. The statement `g.drawString(someString, 50, 100);` places `someString`'s _____ corner at position 50, 100.

   a. upper-left
   b. lower-left
   c. upper-right
   d. lower-right

6. If you use the `setColor()` method to change a `Graphics` object's color to yellow, _____.

   a. only the next output from the object appears in yellow
   b. all output from the object for the remainder of the method always appears in yellow
   c. all output from the object for the remainder of the application always appears in yellow
   d. all output from the object appears in yellow until you change the color

7. The correct statement to instantiate a `Graphics` object named `picasso` is _____.

   a. `Graphics picasso;`
   b. `Graphics picasso = new Graphics();`
   c. `Graphics picasso = getGraphics();`
   d. `Graphics picasso = getGraphics(new);`

8. The statement `g.drawRoundRect(100, 100, 100, 100, 0, 0);` draws a shape that looks most like a _____.

   a. square
   b. round-edged rectangle
   c. circle
   d. straight line

9. If you draw an oval with the same value for width and height, you draw a(n) _____.

   a. circle
   b. square
   c. rounded square
   d. ellipsis

10. The zero-degree position for any arc is at the _____ o'clock position.

    a. three
    b. six
    c. nine
    d. twelve

11. The method you use to create a solid arc is _____.

    a. `solidArc()`
    b. `fillArc()`
    c. `arcSolid()`
    d. `arcFill()`

12. You use the _____ method to copy any rectangular area to a new location.

    a. `copyRect()`
    b. `copyArea()`
    c. `repeatRect()`
    d. `repeatArea()`

13. The measurement of an uppercase character from the baseline to the top of the character is its _____.

    a. ascent
    b. descent
    c. leading
    d. height

14. To be certain that a vertical series of `String`s has enough room to appear in a frame, you use which of the following statements?

    a.
    ```
    g.drawString("Some string",
       x, y += g.getFontMetrics().getHeight());
    ```
    b.
    ```
    g.drawString("Some string",
       x, y += g.getFontMetrics().getLeading());
    ```
    c.
    ```
    g.drawString("Some string", x,
       y += g.getFontMetrics().getAscent());
    ```
    d.
    ```
    g.drawString("Some string",
       x, y += g.getFontMetrics().getDescent());
    ```

15. You can discover the fonts that are available on your system by using the _____.

    a. `getAvailableFontFamilyNames()` method of the `GraphicsEnvironment` class
    b. `getFonts()` method of the `Graphics` class
    c. `getMyFonts()` method of the `GraphicsFonts` class
    d. `getAllFonts()` method of the `Fonts` class

16. The `getScreenResolution()` method and `getScreenSize()` method _____.
    a. both return the number of pixels as an `int` type
    b. respectively return the number of pixels as an `int` type and an object of type `Dimension`
    c. both return an object of type `Dimension`
    d. respectively return the number of pixels as a `double` type and an object of type `Dimension`

17. A `Graphics2D` object is produced by _____.
    a. the `setGraphics2D()` method
    b. the `Graphics2D newpen = Graphics2D()` statement
    c. the `Graphics2D = Graphics(g)` statement
    d. casting a `Graphics` object

18. The process of drawing with Java 2D objects includes _____.
    a. specifying the rendering attributes
    b. setting a drawing stroke
    c. both of the above
    d. none of the above

19. A gradient fill is a gradual change in _____.
    a. color
    b. font size
    c. drawing style
    d. line thickness

20. With the 2D methods, the drawing line is a _____.
    a. brush
    b. stroke
    c. belt
    d. draw

# Exercises

## Programming Exercises

1. Write an application that extends `JFrame` and that displays a phrase in every font size from 6 through 20. Save the file as **JFontSizes.java**.

2. a. Write an application that extends `JFrame` and that displays a phrase in one color the first time the user clicks a `JButton`. The next time the user clicks the `JButton`, make the first phrase seem to disappear. (*Hint*: Redraw it using the background color.) At the same time, draw the phrase again in a different color,

size, and horizontal position. The third click should change the color, size, and position of the phrase again. Save the file as **JChangeSizeAndColor.java**.

b. Modify the `JChangeSizeAndColor` application so that it continuously changes the size, color, and location of a phrase as long as the user continues to click the button. Save the application as **JChangeSizeAndColor2.java**.

3. Write an application that extends `JFrame` and that displays a phrase upside down when the user clicks a button. The phrase is displayed normally when the user clicks the button again. Save the application as **JUpsideDown.java**.

4. Write an application that extends `JFrame` and that displays eight nested rectangles, like those in Figure 16-38. You may use only one `drawRect()` statement in the program. (*Hint*: Use it in a loop.) Save the file as **JNestedBoxes.java**.

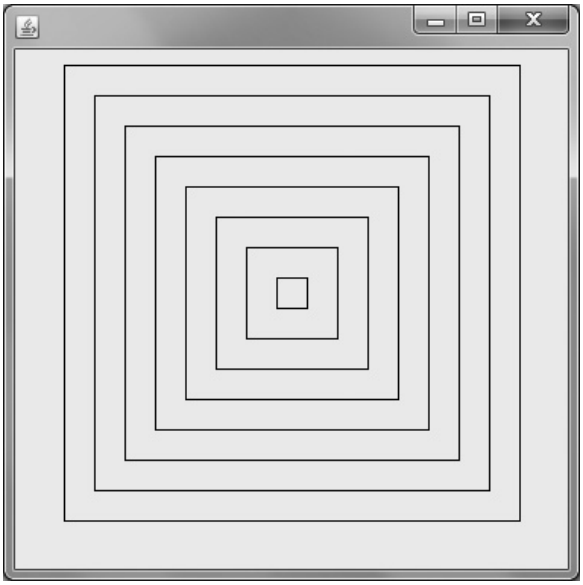

**Figure 16-38** Output of the `JNestedBoxes` program

5. Write an application that extends `JFrame` and that displays 15 nested circles, like those in Figure 16-39. You may use only one `drawOval()` statement in the program. Save the file as **JNestedCircles.java**.

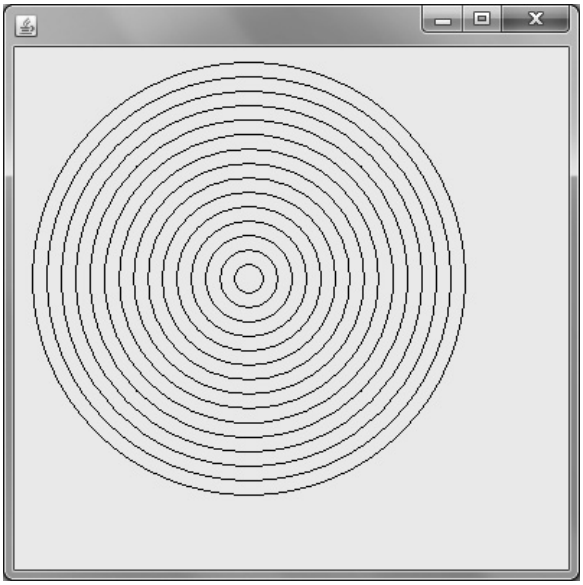

**Figure 16-39** Output of the `JNestedCircles` program

6. Write an application that extends `JFrame` and that displays diagonal lines in a square, like those in Figure 16-40. Save the file as **JDiagonalLines.java**.

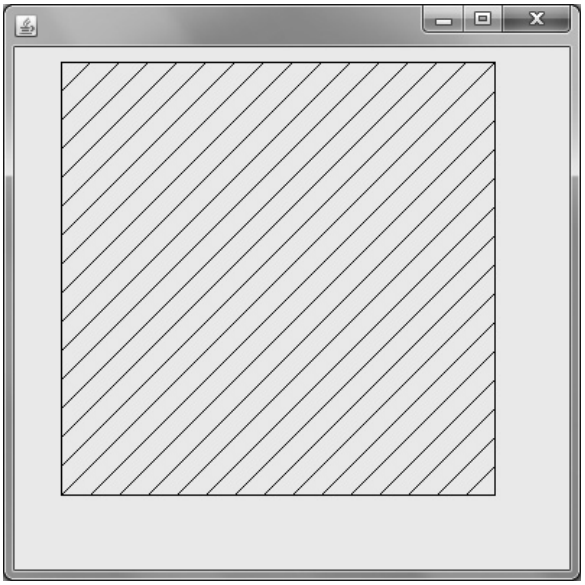

**Figure 16-40** Output of the `JDiagonalLines` program

7. a. Write an application that extends `JFrame` and that displays a yellow smiling face on the screen. Save the file as **JSmileFace.java**.

   b. Add a `JButton` to the `JSmileFace` program so the smile changes to a frown when the user clicks the `JButton`. Save the file as **JSmileFace2.java**.

8. a. Use polygons and lines to create a graphics image that looks like a fireworks display. Write an application that extends `JFrame` and that displays the fireworks. Save the file as **JFireworks.java**.

   b. Add a `JButton` to the `JFireworks` program. Do not show the fireworks until the user clicks the `JButton`. Save the file as **JFireworks2.java**.

9. a. Write an application that extends `JFrame` and that displays your name. Place boxes around your name at intervals of 10, 20, 30, and 40 pixels. Save the file as **JBorders.java**.

   b. Modify the `JBorders` program so that each of the four borders is a different color. Save the file as **JBorders2.java**.

10. Search the Web for the approximate value of the U.S. dollar in other currencies. Write an application that extends `JFrame` and that prompts the user to enter a value in U.S. dollars. Use `Graphics2D` methods to display the dollar amount as well as the equivalent values of two other currencies of your choice. Save the file as **JCurrencies.java**.

11. Write an application that extends `JFrame` and that uses the `Graphics2D` environment to create a `GeneralPath` object. Use the `GeneralPath` object to create the outline of your favorite state. Display the state name at the approximate center of the state boundaries. Save the file as **JFavoriteState.java**.

12. Write an application that extends `JFrame` and that draws a realistic-looking stop sign. Save the file as **JStopSign.java**.

13. Write an application that displays a `JFrame` that does the following:

    - Turns yellow when the user's mouse enters the frame
    - Turns black when the user's mouse exits the frame
    - Displays a larger circle at a point near where the user left-clicks
    - Displays a smaller circle at a point near where the user right-clicks

    At most, one circle should appear on the surface of the frame at a time. Save the file as **JMouseFrame.java**.

## Debugging Exercises

1. Each of the following files in the Chapter16 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugSixteen1.java will become FixDebugSixteen1.java.

   a. DebugSixteen1.java
   b. DebugSixteen2.java
   c. DebugSixteen3.java
   d. DebugSixteen4.java

## Game Zone

1. a. In Chapter 9, you created a Tic Tac Toe game in which you used a 2D array of characters to hold *X*s and *O*s for a player and the computer. Now create a `JFrame` that uses an array of nine `JButton`s to represent the Tic Tac Toe grid. When the user clicks a `JButton` that has not already been taken, place an *X* on the button and then allow the computer to place an *O* on a different button. Announce the winner when either the computer or the player achieves three marks in sequence, or announce that the game was a tie. Figure 16-41 shows a typical game in progress and after the player has won. Save the game as **JTicTacToe.java**.


**Figure 16-41** Typical execution of the `JTicTacToe` program

b. Add a graphic that displays a large letter representing the winning player of the game in Game Zone exercise 1a. Draw a large *X*, *O*, or, in case of a tie, an overlapping *X* and *O* in different colors. Save the game as **JTicTacToe2.java**.

2. Create an application that plays a card game named Lucky Seven. In real life, the game can be played with seven cards, each containing a number from 1 through 7, that are shuffled and dealt number-side down. To start the game, a player turns over any card. The exposed number on the card determines the position (reading from left to right) of the next card that must be turned over. For example, if the player turns over the first card and its number is 7, the next card turned must be the seventh card (counting from left to right). If the player turns over a card whose number denotes a position that was already turned, the player loses the game. If the player succeeds in turning over all seven cards, the player wins.

   Instead of cards, you will use seven buttons labeled 1 through 7 from left to right. Randomly associate one of the seven values 1 through 7 with each button. (In other words, the associated value might or might not be equivalent to the button's labeled value.) When the player clicks a button, reveal the associated hidden value. If the value represents the position of a button already clicked, the player loses. If the revealed number represents an available button, force the user to click it; that is, do not take any action until the user clicks the correct button. After a player clicks a button, remove the button from play. (After you remove a button, you can call `repaint()` to ensure that the image of the button is removed.)

   For example, a player might click Button 7, revealing a 4. Then the player clicks Button 4, revealing a 2. Then the player clicks Button 2, revealing a 7. The player loses because Button 7 was already used. Save the game as **JLuckySeven.java**.

3. a. In Chapters 7 and 8, you created a game named Secret Phrase in which the user guesses a randomly selected secret phrase by entering one letter at a time. Now create a GUI application that plays the game, allowing users to choose a letter by selecting one of 26 buttons. (*Hint*: Consider creating an array of buttons rather than 26 individually named buttons.)

      Disable a letter button once it has been guessed, and after the puzzle is complete, disable all the letters. Figure 16-42 shows a typical execution (1) after the user has guessed an *A*, which is in the phrase; (2) after the user has guessed a *D*, which is not in the phrase; and (3) after the user has completed the puzzle. Save the file as **JSecretPhrase.java**.


**Figure 16-42** Typical execution of the `JSecretPhrase` program

b. Make the `JSecretPhrase` game more like the traditional letter-guessing game Hangman by drawing a "hanged" person piece by piece with each missed letter. For example, when the user chooses a correct letter, place it in the appropriate position or positions in the phrase, but the first time the user chooses a letter that is not in the target phrase, draw a head for the "hanged" man. The second time the user makes an incorrect guess, add a torso. Continue with arms and legs. If the complete body is drawn before the user has guessed all the letters in the phrase, display a message indicating that the player has lost the game. If the user completes the phrase before all the body parts are drawn, display a message that the player has won. Save the game as **JSecretPhrase2.java**.

## Case Problems

1. In Chapters 14 and 15, you developed an interactive GUI application for Carly's Catering. Now, design a `JPanel` that uses graphics to display a logo for the company, and modify the GUI application to include it. Save the `JPanel` class as **JCarlysLogoPanel.java**, and save the GUI application as **JCarlysCatering.java**.
2. In Chapters 14 and 15, you developed an interactive GUI application for Sammy's Seashore Rentals. Now, design a `JPanel` that uses graphics to display a logo for the company, and modify the GUI application to include it. Save the `JPanel` class as **JSammysLogoPanel.java**, and save the GUI application as **JSammysSeashore.java**.