CHAPTER 44

JAVA 2D

Objectives

- To obtain a Graphics2D object for rendering Java 2D shapes (§44.2).
- To use geometric models to separate modeling of shapes from rendering (§44.3).
- To know the hierarchy of shapes (§44.3).
- To model lines, rectangles, ellipses, arcs using Line2D, Rectangle2D, RoundRectangle2D, Ellipse2D, and Arc2D (§44.4).
- To perform coordinate transformation using the translate, rotate, and scale methods (§44.5).
- To specify the attributes of lines using the BasicStroke class (§44.6).
- To define a varying color using GradientPaint and define an image paint using TexturePaint (§44.7).
- To model quadratic curves and cubic curves using the QuadCurve2D and CubicCurve2D classes (§44.8).
- To model an arbitrary geometric path using Path2D and to define interior points using the WIND_EVEN_ODD and WIND_NON_ZERO rules (§44.9).
- To perform constructive area geometry using the Area class (§44.10).

44.1 Introduction

Using the methods in the Graphics class, you learned how to draw lines, rectangles, ovals, arcs, and polygons. This chapter introduces Java 2D, which enables you to draw advanced and complex two-dimensional graphics.

Note

This chapter introduces the basic and commonly used features in Java 2D. For a complete coverage of Java 2D, please see *Computer Graphics Using Java 2D and 3D* by Hong Zhang and Y. Daniel Liang, published by Prentice Hall.

44.2 Obtaining a **Graphics2D** Object

You used the drawing methods in the Graphics class in the text. The Graphics class is primitive. The Java 2D API provides the java.awt.Graphics2D class, which extends java.awt.Graphics with advanced capabilities for rendering graphics. Normally, you write the code to draw graphics in the paintComponent method in a GUI component. The coding template for the method is as follows:

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);

  // Use the method in Graphics to draw graphics
  ...
}
```

The parameter passed to the paintComponent method is actually an instance of Graphics2D. So, to obtain a Graphics2D reference, you may simply cast the parameter g to Graphics2D as follows:

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);

  Graphics2D g2d = (Graphics2D)g; // Get a Graphics2D object

  // Use the method in Graphics2D to draw graphics
  ...
}
```

Since Graphics2D is a subclass of Graphics, all the methods in Graphics can be used in Graphics2D. Additionally, you can use the methods in Graphics2D.

44.3 Geometric Models

You have used the methods in the Graphics class to draw lines, rectangles, arcs, ellipses, and polygons. The Java 2D API uses the model-view controller architecture to separate rendering from modeling. This approach enables you to create shapes and perform manipulations, such as transforming and rotating, to combine shapes using models, and to use Graphics2D to render shapes.

Java 2D provides facilities to construct basic shapes and to combine them to form more complex shapes. Figure 44.1 shows various shapes supported in Java 2D.

methods in **Shape**

The Shape interface defines the common features for shapes and provides the contains method to test whether a point or a rectangle is inside a shape, and the intersects method to test whether the shape overlaps with a rectangle, as shown in Figure 44.2. These methods are often useful in geometrical programming.

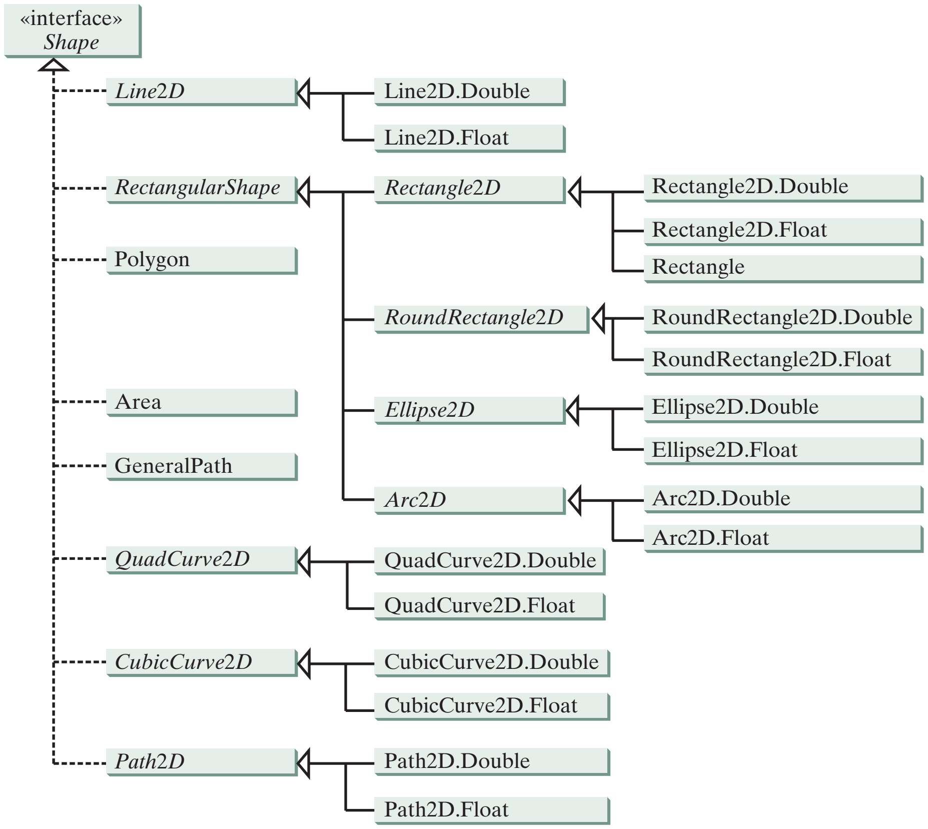
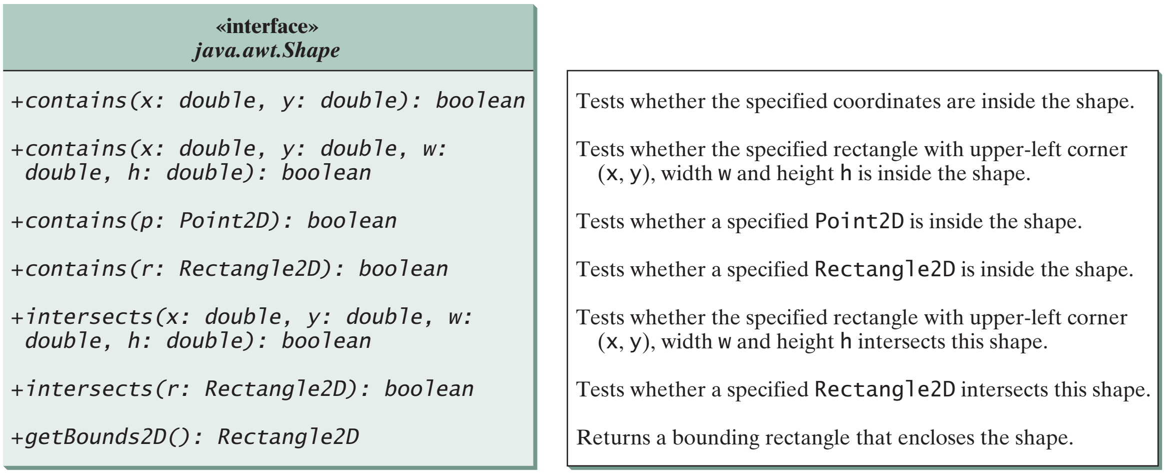


FIGURE 44.1 Java 2D defines various shapes.

concrete shape classes

Classes `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Arc2D`, `Ellipse2D`, `QuadCurve2D`, `CubicCurve2D`, and `Path2D` are abstract classes. Each contains two concrete static inner classes named `Double` and `Float` for `double` and `float` coordinates, respectively. For example, `Line2D.Double` refers to the static inner class `Double` defined in the `Line2D` class. You can use either `Line2D.Double` or `Line2D.Float` to create an object for modeling



«interface»
java.awt.Shape

+contains(x: double, y: double): boolean — Tests whether the specified coordinates are inside the shape.

+contains(x: double, y: double, w: double, h: double): boolean — Tests whether the specified rectangle with upper-left corner (x, y), width w and height h is inside the shape.

+contains(p: Point2D): boolean — Tests whether a specified Point2D is inside the shape.

+contains(r: Rectangle2D): boolean — Tests whether a specified Rectangle2D is inside the shape.

+intersects(x: double, y: double, w: double, h: double): boolean — Tests whether the specified rectangle with upper-left corner (x, y), width w and height h intersects this shape.

+intersects(r: Rectangle2D): boolean — Tests whether a specified Rectangle2D intersects this shape.

+getBounds2D(): Rectangle2D — Returns a bounding rectangle that encloses the shape.

FIGURE 44.2 `Shape` is the root interface for all Java 2D shapes.

a line, depending on whether you want to use double or float for coordinates. These inner classes are also subclasses of their respective outer classes. So Line2D.Double is a subclass of Line2D.

Point2D

A point can be modeled using the abstract Point2D class. It contains two concrete static inner classes Point2D.Double and Point2D.Float for double and float coordinates, respectively. Point2D.Double and Point2D.Float are also subclasses of Point2D. The Point class was introduced in JDK 1.1 and now is included in Java 2D for backward compatibility. Point is now defined as a subclass of Point2D. Point2D contains the methods for finding the distance between two points.

create a shape

To create a shape, use the constructor of a concrete shape class. For example, to model a line from (x1, y1) to (x2, y2), you may create a Line2D object with double data type using the following constructor:

create a line

```
Line2D line = new Line2D.Double(x1, y1, x2, y2);
```

The Graphics2D class contains the draw(Shape s) method to draw the boundary of the shape and the fill(Shape s) method to fill the interior of the shape. To render the line on a GUI component, use

render a line

```
g2d.draw(line);
```

where g2d is a Graphics2D object for the GUI component.

44.4 Rectangle2D, RoundRectangle2D, Arc2D, and Ellipse2D

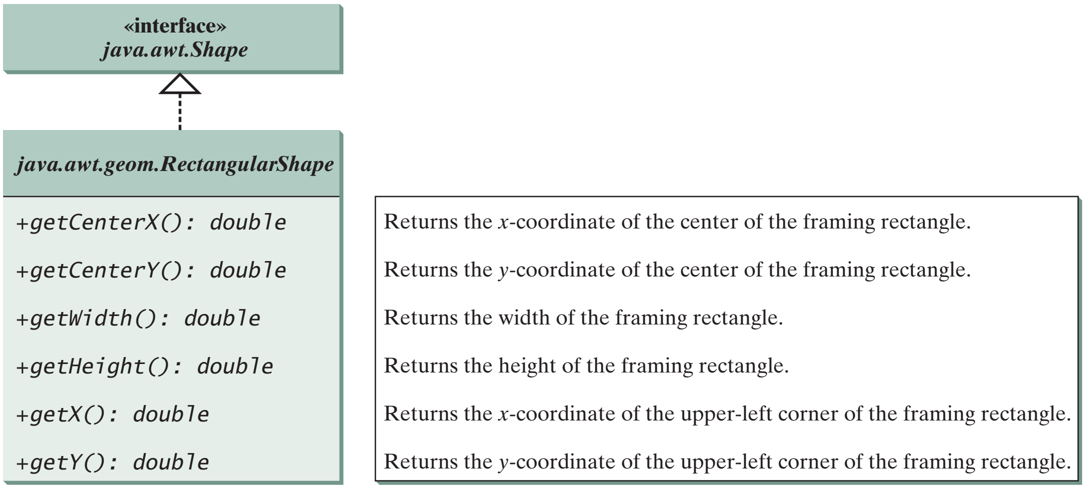
RectangularShape

RectangularShape is an abstract base class for Rectangle2D, RoundRectangle2D, Arc2D, and Ellipse2D, whose geometry is defined by a rectangular frame. Figure 44.3 shows the UML diagram for RectangularShape.

Rectangle2D

Rectangle2D models a rectangle with horizontal and vertical sides. The Rectangle class was introduced in JDK 1.1 and now is included in Java 2D for backward compatibility. Rectangle is now defined as a subclass of Rectangle2D. It models a rectangle with integer coordinates, while Rectangle2D.Double and Rectangle2D.Float model a rectangle with double and float coordinates, respectively. You can construct a Rectangle using

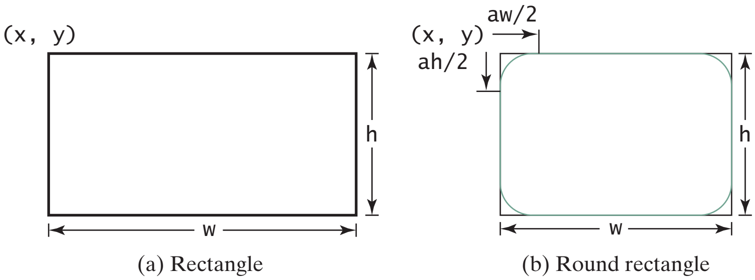
```
new Rectangle(x, y, w, h)
```



+getCenterX(): double	Returns the *x*-coordinate of the center of the framing rectangle.
+getCenterY(): double	Returns the *y*-coordinate of the center of the framing rectangle.
+getWidth(): double	Returns the width of the framing rectangle.
+getHeight(): double	Returns the height of the framing rectangle.
+getX(): double	Returns the *x*-coordinate of the upper-left corner of the framing rectangle.
+getY(): double	Returns the *y*-coordinate of the upper-left corner of the framing rectangle.

FIGURE 44.3 RectangularShape defines a shape with a bounding rectangle.

The parameters x and y represent the upper-left corner of the rectangle, and w and h are its width and height (see Figure 44.4(a)).



(a) Rectangle (b) Round rectangle

Figure 44.4 (a) A rectangle is defined in four parameters. (b) A round rectangle is defined in six parameters.

The following code creates three Rectangle2D objects with integer, double, and float coordinates, respectively. The upper-left corner of the rectangle is at (20, 40) with width 100 and height 200.

```
Rectangle2D ri = new Rectangle(20, 40, 100, 200);
Rectangle2D rd = new Rectangle.Double(20D, 40D, 100D, 200D);
Rectangle2D rf = new Rectangle.Double(20F, 40F, 100F, 200F);
```

RoundRectangle2D models a rectangle with round corners. You can construct a RoundRectangle using

RoundRectangle2D

```
new RoundRectangle2D.Double(x, y, w, h, aw, ah)
```

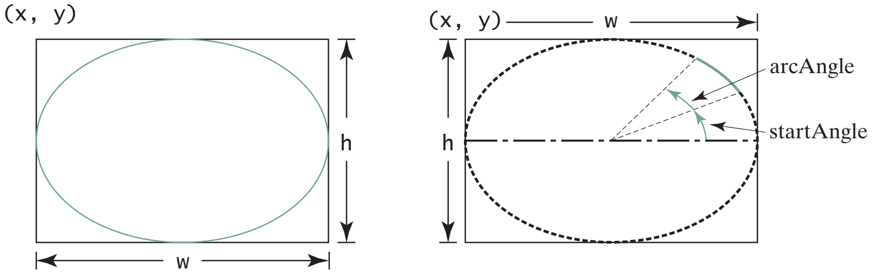
Parameters x, y, w, and h specify a rectangle, parameter aw is the horizontal diameter of the arcs at the corner, and ah is the vertical diameter of the arcs at the corner (see Figure 44.4(b)). In other words, aw and ah are the width and the height of the oval that produces a quarter-circle at each corner.

Ellipse2D models an ellipse. You can construct an Ellipse2D using

Ellipse2D

```
new Ellipse2D.Double(x, y, w, h)
```

Parameters x, y, w and h specify the bounding rectangle for the ellipse, as shown in Figure 44.5(a).



(a) Ellipse (b) Arc

Figure 44.5 An ellipse or oval is defined by its bounding rectangle.

Arc2D

Arc2D models an elliptic arc. You can construct an Arc2D using

```
new Arc2D.Double(x, y, w, h, startAngle, arcAngle, type)
```

Parameters x, y, w and h specify the bounding rectangle for the arc; parameter startAngle is the starting angle; arcAngle is the spanning angle (i.e., the angle covered by the arc). Angles are measured in degrees and follow the usual mathematical conventions (i.e., 0 degrees is in the easterly direction, and positive angles indicate counterclockwise rotation from the easterly direction); see Figure 44.5(b).

Parameter type is Arc2D.OPEN, Arc2D.CHORD, or Arc2D.PIE. Arc2D.OPEN specifies that the arc is open. Arc2D.CHORD specifies that the arc is connected by drawing a line segment from the start the arc to the end of the arc. Arc2D.PIE specifies that the arc is connected by drawing straight line segments from the start of the arc segment to the center of the full ellipse and from that point to the end of the arc segment.

Listing 44.1 gives a program that demonstrates how to draw various shapes using Graphics2D. Figure 44.6 shows a sample run of the program.

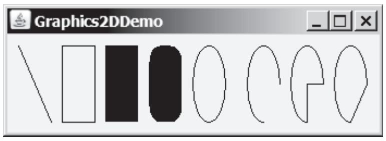


Figure 44.6 You can draw various shapes using Java 2D.

Listing 44.1 Graphics2DDemo.java

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class Graphics2DDemo extends JApplet {
  public Graphics2DDemo() {
    add(new ShapePanel());
  }

  static class ShapePanel extends JPanel {
    protected void paintComponent(Graphics g) {
      super.paintComponent(g);

      Graphics2D g2d = (Graphics2D)g;

      g2d.draw(new Line2D.Double(10, 10, 40, 80));
      g2d.draw(new Rectangle2D.Double(50, 10, 30, 70));
      g2d.fill(new Rectangle2D.Double(90, 10, 30, 70));
      g2d.fill(new RoundRectangle2D.Double(130, 10, 30, 70, 20, 30));
      g2d.draw(new Ellipse2D.Double(170, 10, 30, 70));
      g2d.draw(
        new Arc2D.Double(220, 10, 30, 70, 0, 270, Arc2D.OPEN));
      g2d.draw(new Arc2D.Double(260, 10, 30, 70, 0, 270, Arc2D.PIE));
      g2d.draw(
        new Arc2D.Double(300, 10, 30, 70, 0, 270, Arc2D.CHORD));
    }
  }
}
```

import for shape classes

applet

Graphics2D reference

draw a line

draw a rectangle

fill a rectangle

round rectangle

draw an ellipse

draw an arc

main method omitted

The shape classes `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Arc2D`, and `Ellipse2D` are in the `java.awt.geom` package. So, they are imported in line 2.

Line2D

A `Graphics2D` reference is obtained in line 14 in order to invoke the methods in `Graphics2D`. The statement `new Line2D.Double(10, 10, 40, 80)` (line 16) creates an instance of `Line2D.Double`, which is also an instance of `Line2D` and `Shape`. The instance models a line from (`10`, `10`) to (`40`, `80`).

Rectangle2D

The statement `new Rectangle2D.Double(50, 10, 30, 70)` (line 17) creates an instance of `Rectangle2D.Double`, which is also an instance of `Rectangle2D` and `Shape`. The instance models a rectangle whose upper-left corner point is (`50`, `10`) with width `30` and height `70`.

fill

The `fill(Shape)` method (line 18) renders a filled rectangle.

RoundRectangle2D

The statement `new RoundRectangle2D.Double(130, 10, 30, 70, 20, 30)` (line 19) creates an instance of `RoundRectangle2D.Double`, which is also an instance of `RoundRectangle2D` and `Shape`. The instance models a round-cornered rectangle whose parameters are the same as in the `drawRoundRect(int x, int y, int w, int h, int aw, int ah)` method in the `Graphics` class.

Ellipse2D

The statement `new Ellipse2D.Double(300, 10, 30, 70)` (line 20) creates an instance of `Ellipse2D.Double`, which is also an instance of `Ellipse2D` and `Shape`. The instance models an ellipse. The parameters in this constructor are the same as the parameters in the `drawOval(int x, int y, int w, int h)` method in the `Graphics` class.

Arc2D

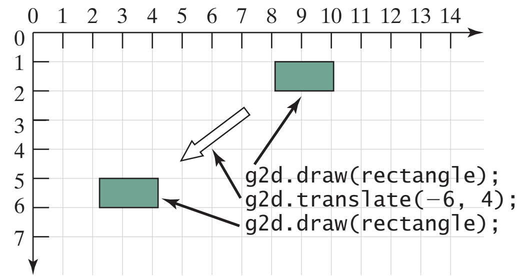
The statement `new Arc2D.Double(170, 10, 30, 70, 0, 270, Arc2D.OPEN)` (line 21) creates an instance of `Arc2D.Double`, which is also an instance of `Arc2D` and `Shape`. The instance models an open arc. The parameters in this constructor are similar to the parameters in the `drawArc(int x, int y, int w, int h, int startAngle, int arcAngle)` method in the `Graphics` class, except that the last parameter specifies whether the arc is open or closed. The value `Arc2D.OPEN` specifies that the arc is open. The value `Arc2D.PIE` (line 23) specifies that the arc is closed by drawing straight line segments from the start of the arc segment to the center of the full ellipse and from that point to the end of the arc segment. The value `Arc2D.CHORD` (line 25) specifies that the arc is closed by drawing a straight line segment from the start of the arc segment to the end of the arc segment.

44.5 Coordinate Transformations

Java 2D provides the classes for modeling geometric objects. It also supports coordinate transformations using translation, rotation, and scaling.

44.5.1 Translations

You can use the `translate(double x, double y)` method in the `Graphics` class to move the subsequent rendering by the specified distance relative to the previous position. For example, `translate(5, -10)` moves subsequent rendering `5` pixels to the right and `10` pixels up from the previous position, and `translate(-5, 10)` moves all shapes `5` pixels to the left and `10` pixels down from the previous position. Figure 44.7 shows a rectangle displayed



```
g2d.draw(rectangle);
g2d.translate(-6, 4);
g2d.draw(rectangle);
```

Figure 44.7 (a) After applying `g2d.translate(-6, 4)`, the subsequent rendering of the rectangle is moved by the specified distance relative to the previous position.

before and after applying translation. After invoking `g2d.translate(-6, 4)`, the rectangle is displayed 6 pixels to the left and 4 pixels down from the previous position.

Listing 44.2 gives a program that demonstrates the effect of translation of coordinates. Figure 44.8 shows a sample run of the program.

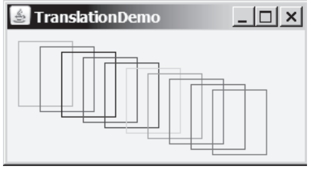


FIGURE 44.8 The rectangles are displayed successively in new locations.

LISTING 44.2 TranslationDemo.java

import for shape classes

applet

Graphics2D reference

a rectangle

random number

set a new color

display rectangle

translate

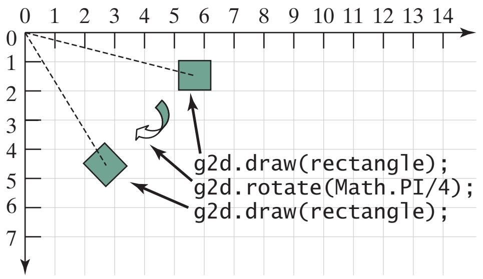
main method omitted

```
 1 import java.awt.*;
 2 import java.awt.geom.*;
 3 import javax.swing.*;
 4
 5 public class TranslateDemo extends JApplet {
 6   public TranslateDemo() {
 7     add(new ShapePanel());
 8   }
 9
10 class ShapePanel extends JPanel {
11   protected void paintComponent(Graphics g) {
12     super.paintComponent(g);
13
14     Graphics2D g2d = (Graphics2D)g;
15     Rectangle2D rectangle = new Rectangle2D.Double(10, 10, 50, 60);
16
17     java.util.Random random = new java.util.Random();
18     for (int i = 0; i < 10; i++) {
19       g2d.setColor(new Color(random.nextInt(256),
20         random.nextInt(256), random.nextInt(256)));
21       g2d.draw(rectangle);
22       g2d.translate(20, 5);
23     }
24   }
25 }
```

Line 17 creates a `Random` object. The `Random` class was introduced in §8.6.2, "The `Random` Class." Invoking `random.nextInt(256)` (line 19) returns a random `int` value between `0` and `255`. The `setColor` method (line 19) sets a new color for subsequent rendering. Line 21 draws a rectangle. The `translate(20, 5)` method in line 22 moves the subsequent rendering `20` pixels to the right and `5` pixels down.

44.5.2 Rotations

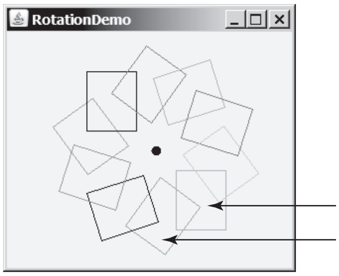
You can use the `rotate(double theta)` method in the `Graphics2D` class to rotate subsequent rendering by `theta` degrees from the origin clockwise, where `theta` is a double value in radians. By default the origin is (`0`, `0`). You can use the `translate(x, y)` method to move the origin to a specified location. For example, `rotate(Math.PI / 4)` rotates subsequent rendering `45` degrees counterclockwise along the northern direction from the origin, as shown in Figure 44.9.



```
g2d.draw(rectangle);
g2d.rotate(Math.PI/4);
g2d.draw(rectangle);
```

FIGURE 44.9 After performing `g2d.rotate(Math.PI / 4)`, the rectangle is rotated in 45 degrees from the origin.

Listing 44.3 gives a program that demonstrates the effect of rotation of coordinates. Figure 44.10 shows a sample run of the program.



This rectangle is drawn first

This rectangle is drawn next

FIGURE 44.10 After the `rotate` method is invoked, the rectangles are displayed successively in new locations.

LISTING 44.3 RotationDemo.java

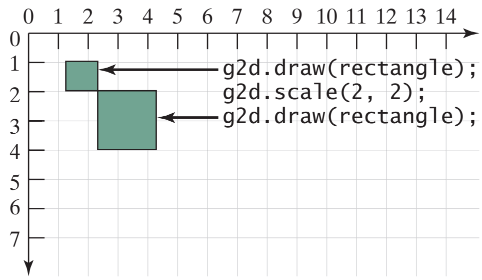
```java
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class RotationDemo extends JApplet {
  public RotationDemo() {
    add(new ShapePanel());
  }

  class ShapePanel extends JPanel {
    protected void paintComponent(Graphics g) {
      super.paintComponent(g);

      Graphics2D g2d = (Graphics2D)g;
      Rectangle2D rectangle = new Rectangle2D.Double(20, 20, 50, 60);

      g2d.translate(150, 120); // Move origin to the center
      g2d.fill(new Ellipse2D.Double(-5, -5, 10, 10));
      java.util.Random random = new java.util.Random();
      for (int i = 0; i < 10; i++) {
        g2d.setColor(new Color(random.nextInt(256),
          random.nextInt(256), random.nextInt(256)));
        g2d.draw(rectangle);
        g2d.rotate(Math.PI / 5);
25        }
26      }
27    }
28  }
```

main method omitted

The `translate(150, 120)` method moves the origin from (0, 0) to (150, 120) (line 17). The loop is repeated ten times. Each iteration sets a new color randomly (line 21), draws the rectangle (line 23), and rotates 36 degrees from the new origin (line 24).

44.5.3 Scaling

You can use the `scale(double sx, double sy)` method in the `Graphics2D` class to resize subsequent rendering by the specified scaling factors. For example, `scale(2, 2)` resizes the object by doubling the *x*- and *y*-coordinates in the object, as shown in Figure 44.11.



```
g2d.draw(rectangle);
g2d.scale(2, 2);
g2d.draw(rectangle);
```

FIGURE 44.11 After performing `g2d.scale(2, 2)`, the x- and y-coordinates in the original rectangle are doubled.

Listing 44.4 gives a program that demonstrates the effect of using scaling. Figure 44.12 shows a sample run of the program.

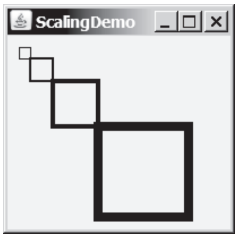


FIGURE 44.12 After scaling is applied, the rectangles are displayed successively.

LISTING 44.4 ScalingDemo.java

import for shape classes

applet

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class ScalingDemo extends JApplet {
  public ScalingDemo() {
    add(new ShapePanel());
  }

  class ShapePanel extends JPanel {
    protected void paintComponent(Graphics g) {
      super.paintComponent(g);
13
14        Graphics2D g2d = (Graphics2D)g;
15        Rectangle2D rectangle = new Rectangle2D.Double(10, 10, 10, 10);
16
17        for (int i = 0; i < 4; i++) {
18          g2d.draw(rectangle);
19          g2d.scale(2, 2);
20        }
21      }
22    }
23 }
```

Graphics2D reference

a rectangle

display rectangle

scale

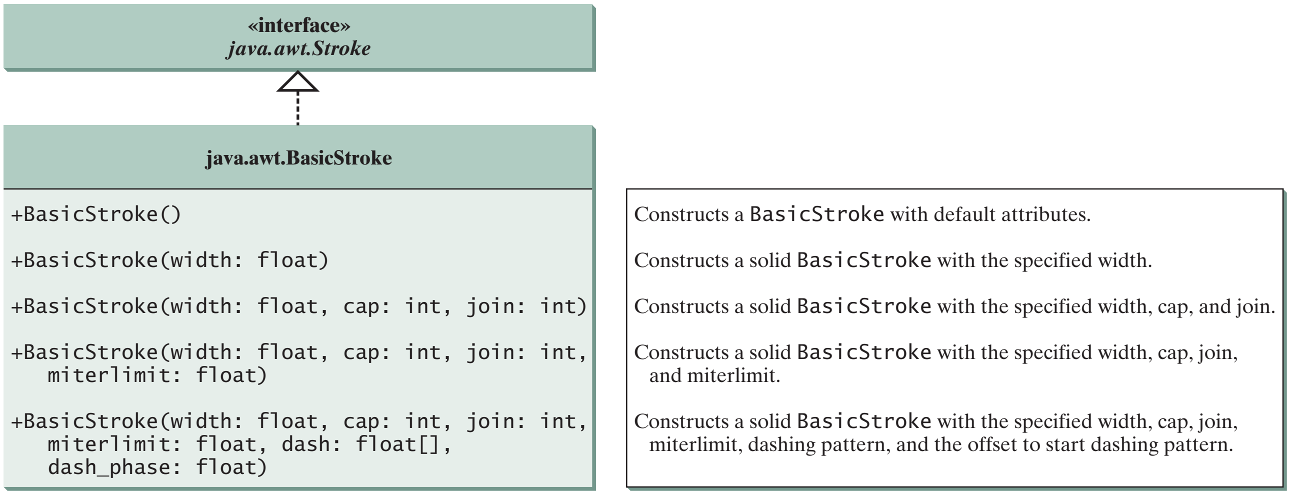
main method omitted

The program draws four rectangles. The upper-left corner of the first rectangle is at (10, 10). After invoking **scale(2, 2)** (line 19) on the **Graphics2D** object **g2d** in the first iteration of the loop, the upper-left corner of the second rectangle is at (20, 20), since this **scale** method causes the coordinates in the current object to be doubled. After invoking **scale(2, 2)** (line 19) on the **Graphics2D** object **g2d** in the second iteration of the loop, the upper-left corner of the third rectangle is at (40, 40). After invoking **scale(2, 2)** (line 19) on the **Graphics2D** object **g2d** in the third iteration of the loop, the upper-left corner of the fourth rectangle is at (80, 80).

It is interesting to note that the thickness of line segments also doubles each time **scale(2, 2)** is invoked. We will discuss the thickness of lines in the next section.

44.6 Strokes

Java 2D allows you to specify the attributes of lines, called *strokes*. You can specify the width of the line, how the line ends (called *end caps*), how lines join together (called *line joins*), and whether the line is dashed. These attributes are defined in a **Stroke** object. You can create a **Stroke** object using the **BasicStroke** class, as shown in Figure 44.13.



«interface»
java.awt.Stroke
java.awt.BasicStroke
+BasicStroke()
+BasicStroke(width: float)
+BasicStroke(width: float, cap: int, join: int)
+BasicStroke(width: float, cap: int, join: int, miterlimit: float)
+BasicStroke(width: float, cap: int, join: int, miterlimit: float, dash: float[], dash_phase: float)
Constructs a BasicStroke with default attributes.
Constructs a solid BasicStroke with the specified width.
Constructs a solid BasicStroke with the specified width, cap, and join.
Constructs a solid BasicStroke with the specified width, cap, join, and miterlimit.
Constructs a solid BasicStroke with the specified width, cap, join, miterlimit, dashing pattern, and the offset to start dashing pattern.

FIGURE 44.13 You can create a **Stroke** using the **BasicStroke** class.

The parameter **width** specifies the thickness of the stroke with a default value **1.0**. The parameter **cap** is one of three values:

- **BasicStroke.CAP_ROUND** for round cap.
- **BasicStroke.CAP_SQUARE** for square cap.
- **BasicStroke.CAP_BUTT** for no added decorations.

The parameter join is one of three values:

- BasicStroke.JOIN_BEVEL for joining the outer corners of their wide outlines with a straight segment.
- BasicStroke.JOIN_MITER for joining path segments by extending their outside edges until they meet.
- BasicStroke.JOIN_ROUND for joining path segments by rounding off the corner at a radius of half the line width.

The parameter miterlimit sets a limit for JOIN_MITER to prevent a very long join when the angle between the two lines is small.

The parameter dash array defines a dash pattern by alternating between opaque and transparent sections. The dash_phase parameter specifies the offset to start the dashing pattern.

To set a stroke in Graphics2D, use

```
void setStroke(Stroke stroke)
```

Listing 44.5 gives a program that demonstrates the effect of using basic strokes. Figure 44.14 shows a sample run of the program.

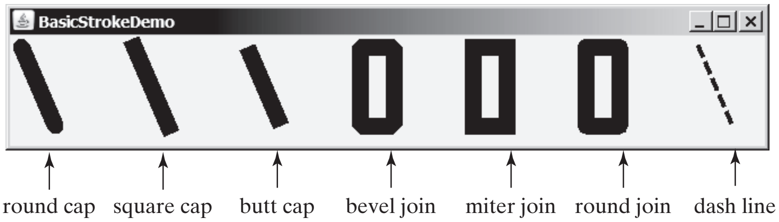


Figure 44.14 You can specify the attributes for strokes.

Listing 44.5 BasicStrokeDemo.java

import for shape classes

applet

Graphics2D reference

set a stroke

draw a line

translate

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class BasicStrokeDemo extends JApplet {
  public BasicStrokeDemo() {
    add(new ShapePanel());
  }

  class ShapePanel extends JPanel {
    protected void paintComponent(Graphics g) {
      super.paintComponent(g);

      Graphics2D g2d = (Graphics2D)g;

      g2d.setStroke(new BasicStroke(15.0f, BasicStroke.CAP_ROUND,
        BasicStroke.JOIN_BEVEL) );
      g2d.draw(new Line2D.Double(10, 10, 40, 80));

      g2d.translate(100, 0);
      g2d.setStroke(new BasicStroke(15.0f, BasicStroke.CAP_SQUARE,
          BasicStroke.JOIN_BEVEL));
23       g2d.draw(new Line2D.Double(10, 10, 40, 80));
24
25       g2d.translate(100, 0);
26       g2d.setStroke(new BasicStroke(15.0f, BasicStroke.CAP_BUTT,
27         BasicStroke.JOIN_BEVEL));
28       g2d.draw(new Line2D.Double(10, 10, 40, 80));
29
30       g2d.translate(100, 0);
31       g2d.draw(new Rectangle2D.Double(10, 10, 30, 70));
32
33       g2d.translate(100, 0);
34       g2d.setStroke(new BasicStroke(15.0f, BasicStroke.CAP_ROUND,
35        BasicStroke.JOIN_MITER));
36       g2d.draw(new Rectangle2D.Double(10, 10, 30, 70));
37
38       g2d.translate(100, 0);
39       g2d.setStroke(new BasicStroke(15.0f, BasicStroke.CAP_SQUARE,
40        BasicStroke.JOIN_ROUND));
41       g2d.draw(new Rectangle2D.Double(10, 10, 30, 70));
42
43       g2d.translate(100, 0);
44       g2d.setStroke(new BasicStroke(4.0f, BasicStroke.CAP_SQUARE,
45        BasicStroke.JOIN_ROUND, 1.0f, new float[]{8}, 0));
46       g2d.draw(new Line2D.Double(10, 10, 40, 80));
47     }
48   }
49 }
```

draw a rectangle

`main` method omitted

The statement *`new BasicStroke(15.0f, BasicStroke.CAP_ROUND, BasicStroke.JOIN_BEVEL)`* (line 16) creates an instance of `BasicStroke`, which is also an instance of the `Stroke` interface. The `setStroke(Stroke)` method sets a `Stroke` object for the `Graphics2D` context. The program sets new `Stroke` objects in lines 21, 26, 34, 39, 44. Line 44 sets a new `Stroke` object with width `4.0f`, round square cap, round join, miter limit `1.0`, dashing pattern {`8`}, and dash phase `0`.

44.7 Paint

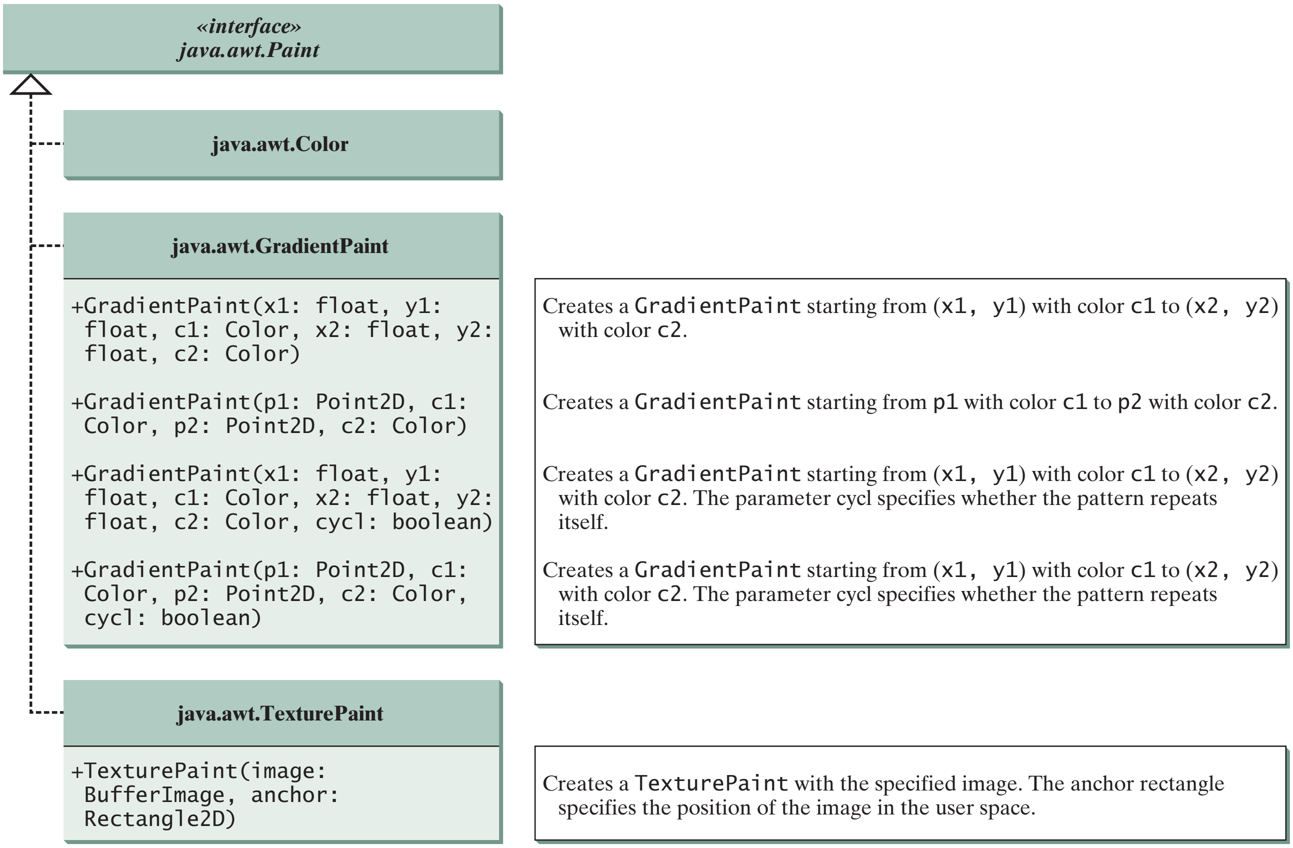
You can use the `setColor(Color c)` method in the `Graphics` class to set a color. It sets only a solid color. `Graphics2D` provides the `setPaint(Paint p)` method to set a paint. `Paint` is a generalization of color. It can represent more attributes than simple solid colors.

`Paint` is an interface for three concrete classes including `Color`, as shown in Figure 44.15.

`GradientPaint` defines a varying color, specified by two points and two colors. As the location moves from the first point to the second, the paint changes gradually from the first color to the second. A `GradientPaint` can be cyclic or acyclic. A cyclic paint repeats the same pattern periodically.

`TexturePaint` defines an image to fill a shape or characters. The parameter `image` is specified as a `BufferedImage`. The `anchor` parameter specifies a rectangle on which the image is anchored. The image is repeated around the anchor rectangle, as shown in Figure 44.16.

Listing 44.6 gives a program that demonstrates the effect of using `GradientPaint` and `TexturePaint`. Figure 44.17 shows a sample run of the program.



«interface»
java.awt.Paint

java.awt.Color

java.awt.GradientPaint

```
+GradientPaint(x1: float, y1:
 float, c1: Color, x2: float, y2:
 float, c2: Color)

+GradientPaint(p1: Point2D, c1:
 Color, p2: Point2D, c2: Color)

+GradientPaint(x1: float, y1:
 float, c1: Color, x2: float, y2:
 float, c2: Color, cycl: boolean)

+GradientPaint(p1: Point2D, c1:
 Color, p2: Point2D, c2: Color,
 cycl: boolean)
```

Creates a `GradientPaint` starting from (`x1, y1`) with color `c1` to (`x2, y2`) with color `c2`.

Creates a `GradientPaint` starting from `p1` with color `c1` to `p2` with color `c2`.

Creates a `GradientPaint` starting from (`x1, y1`) with color `c1` to (`x2, y2`) with color `c2`. The parameter cycl specifies whether the pattern repeats itself.

Creates a `GradientPaint` starting from (`x1, y1`) with color `c1` to (`x2, y2`) with color `c2`. The parameter cycl specifies whether the pattern repeats itself.

java.awt.TexturePaint

```
+TexturePaint(image:
 BufferImage, anchor:
 Rectangle2D)
```

Creates a `TexturePaint` with the specified image. The anchor rectangle specifies the position of the image in the user space.

FIGURE 44.15 A `Paint` object specifies colors.

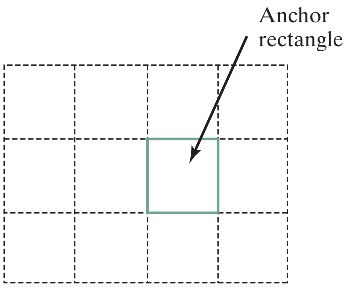


FIGURE 44.16 A `TexturePaint` is specified by an image in an anchor rectangle.

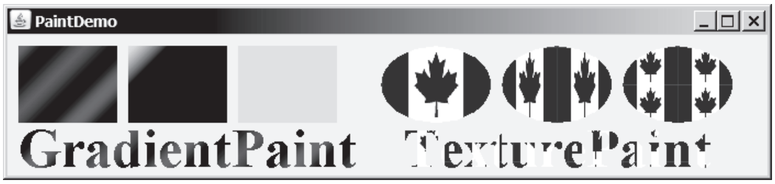


FIGURE 44.17 Shapes and characters are drawn with gradient paint, solid color, and texture paint.

LISTING 44.6 PaintDemo.java

```java
import java.awt.*;
import java.awt.geom.*;
import javax.imageio.ImageIO;
import javax.swing.*;

public class PaintDemo extends JApplet {
  public PaintDemo() {
    add(new ShapePanel());
  }

  class ShapePanel extends JPanel {
    protected void paintComponent(Graphics g) {
      super.paintComponent(g);

      Graphics2D g2d = (Graphics2D)g;

      g2d.setPaint(new GradientPaint(10, 10, Color.RED, 40, 40,
        Color.BLUE, true));
      g2d.fill(new Rectangle2D.Double(10, 10, 90, 70));
      g2d.setFont(new Font("Serif", Font.BOLD, 50));
      g2d.drawString("GradientPaint", 10, 120);

      g2d.translate(100, 0);
      g2d.setPaint(new GradientPaint(10, 10, Color.YELLOW, 40, 40,
        Color.BLACK));
      g2d.fill(new Rectangle2D.Double(10, 10, 90, 70));

      g2d.translate(100, 0);
      g2d.setPaint(Color.YELLOW);
      g2d.fill(new Rectangle2D.Double(10, 10, 90, 70));

      try {
        java.net.URL url =
          getClass().getClassLoader().getResource("image/ca.gif");
        java.awt.image.BufferedImage image = ImageIO.read(url);
        TexturePaint texturePaint = new TexturePaint(image,
            new Rectangle2D.Double(10, 10, 100, 70));
        g2d.translate(130, 0);
        g2d.setPaint(texturePaint);
        g2d.fill(new Ellipse2D.Double(10, 10, 100, 70));

        texturePaint = new TexturePaint(image,
          new Rectangle2D.Double(10, 10, 50, 70));
        g2d.translate(110, 0);
        g2d.setPaint(texturePaint);
        g2d.fill(new Ellipse2D.Double(10, 10, 100, 70));

        texturePaint = new TexturePaint(image,
            new Rectangle2D.Double(10, 10, 50, 35));
        g2d.translate(110, 0);
        g2d.setPaint(texturePaint);
        g2d.fill(new Ellipse2D.Double(10, 10, 100, 70));
        g2d.drawString("TexturePaint", -190, 120);
      }
      catch (java.io.IOException ex) {
        ex.printStackTrace();
      }
    }
```

GradientPaint

solid color

get URL

TexturePaint

set paint

main method

```
59   }
60 }
```

The statement in lines 17–18

```
g2d.setPaint(new GradientPaint(10, 10, Color.RED, 40, 40,
  Color.BLUE, true));
```

creates an instance of GradientPaint and sets the paint in g2d.

The program sets a new Paint object (lines 17, 24, 29) before drawing a filled rectangle (lines 19, 26, 30). Note that you can use the setPaint method to set a Color object (line 29) or use the setColor method in the Graphics class to set a color.

As you see in Figure 44.17, the gradient colors are repeated in the first rectangle, since the GradientPaint is cyclic (lines 17–18). The gradient colors are not repeated in the second rectangle, since the GradientPaint is acyclic (lines 24–25).

To create a TexturePaint, you need to create a BufferedImage from an image file. The URL of the image file is created in lines 33–34. This subject was introduced in §18.10, "Locating Resources Using the URL Class." You can use the static method read in the ImageIO class to obtain a BufferedImage from the URL of the image (line 35).

The statement in lines 36–37

```
TexturePaint texturePaint = new TexturePaint(image,
  new Rectangle2D.Double(10, 10, 100, 70));
```

creates a TexturePaint with the image anchored in the rectangle whose upper-left corner is (10, 10) and width and height are 100 and 70. This TexturePaint object is set in g2d in line 39. Line 40 fills an ellipse with this TexturePaint, as shown in Figure 44.18(a).

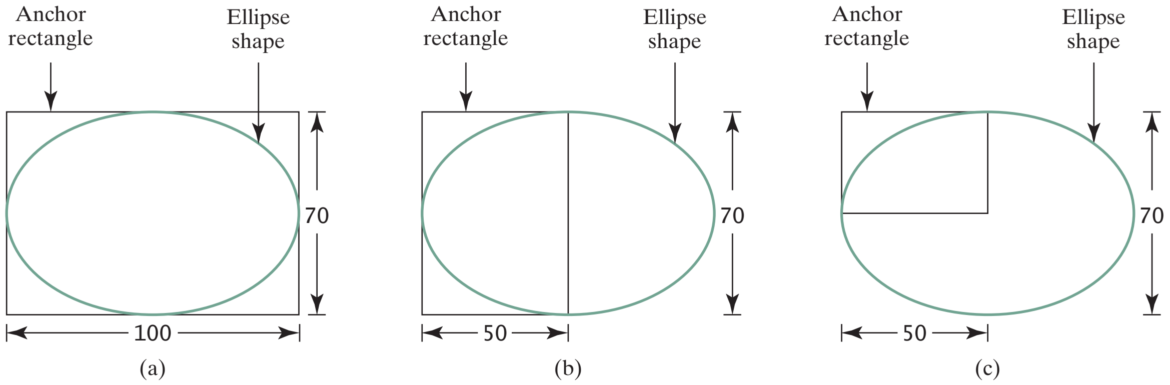


FIGURE 44.18 The anchor rectangle defines the size and position of the starting image.

The statement in lines 42-43

```
texturePaint = new TexturePaint(image,
  new Rectangle2D.Double(10, 10, 50, 70));
```

creates a TexturePaint with the image anchored in the rectangle whose upper-left corner is (10, 10) and width and height are 50 and 70. This TexturePaint object is set in g2d in line 45. Line 46 fills an ellipse with this TexturePaint, as shown in Figure 44.18(b). As you see in the sample output in Figure 44.17, the texture paint is repeated from the anchor rectangle.

Line 53 displays a string. The characters are filled with the paint set in line 51.

44.8 QuadCurve2D and CubicCurve2D

Java 2D provides the QuadCurve2D and CubicCurve2D classes for modeling quadratic curves and cubic curves. QuadCurve2D.Double and QuadCurve2D.Float are two concrete subclasses of QuadCurve2D. CubicCurve2D.Double and CubicCurve2D.Float are two concrete subclasses of CubicCurve2D.

A quadratic curve is mathematically defined as a quadratic polynomial. To create a QuadCurve2D.Double, use the following constructor:

```
QuadCurve2D.Double(double x1, double y1,
  double ctrlx, double ctrly, double x2, double y2)
```

where (x1, y1) and (x2, y2) specify two endpoints and (ctrlx, ctrly) is a control point. The control point is usually not on the curve instead of defining the trend of the curve, as shown in Figure 44.19(a).

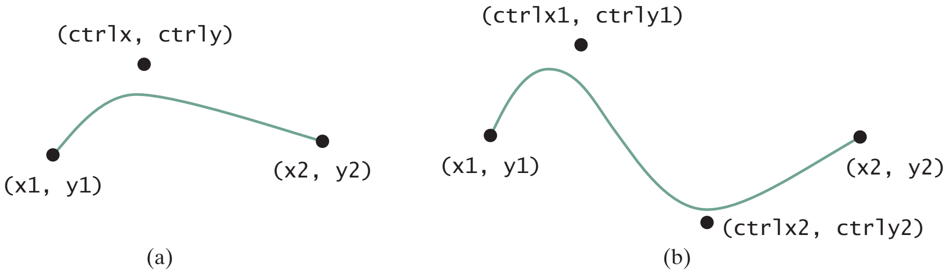


FIGURE 44.19 (a) A quadratic curve is specified using three points. (b) A cubic curve is specified using four points.

A cubic curve is mathematically defined as a cubic polynomial. To create a CubicCurve2D.Double, use the following constructor:

```
CubicCurve2D.Double(double x1, double y1, double ctrlx1,
  double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)
```

where (x1, y1) and (x2, y2) specify two endpoints and (ctrlx1, ctrly1) and (ctrlx2, ctrly2) are two control points. The control points are usually not on the curve instead of defining the trend of the curve, as shown in Figure 44.19(b).

Listing 44.7 gives a program that demonstrates how to draw quadratic curves and cubic curves. Figure 44.20 shows a sample run of the program.

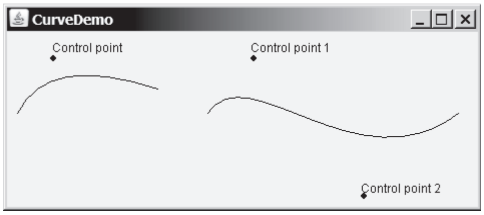


FIGURE 44.20 You can draw quadratic and cubic curves using Java 2D.

LISTING 44.7 CurveDemo.java

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
```

import for shape classes

applet

Graphics2D reference

quadratic curve

cubic curve

main method omitted

```
 4
 5 public class CurveDemo extends JApplet {
 6   public CurveDemo() {
 7     add(new CurvePanel());
 8   }
 9
10   static class CurvePanel extends JPanel {
11     protected void paintComponent(Graphics g) {
12       super.paintComponent(g);
13
14       Graphics2D g2d = (Graphics2D)g;
15
16       // Draw a quadratic curve
17       g2d.draw(new QuadCurve2D.Double(10, 80, 40, 20, 150, 56));
18       g2d.fillOval(40 + 3, 20 + 3, 6, 6);
19       g2d.drawString("Control point", 40 + 5, 20);
20
21       // Draw a cubic curve
22       g2d.draw(new CubicCurve2D.Double
23         (200, 80, 240, 20, 350, 156, 450, 80));
24       g2d.fillOval(240 + 3, 20 + 3, 6, 6);
25       g2d.drawString("Control point 1", 240 + 3, 20);
26       g2d.fillOval(350 + 3, 156 + 3, 6, 6);
27       g2d.drawString("Control point 2", 350 + 3, 156 + 3);
28     }
29   }
30 }
```

QuadCurve2D

A `Graphics2D` reference is obtained in line 14 in order to invoke the methods in `Graphics2D`. The statement `new QuadCurve2D.Double(10, 80, 40, 20, 150, 56)` (line 17) creates an instance of `QuadCurve2D.Double`, which is also an instance of `QuadCurve2D` and `Shape`. The instance models a quadratic curves with two endpoints (`10`, `80`), (`150`, `56`) and a control point (`40`, `20`).

The `fillOval` (line 18) and `drawString` (line 19) methods are defined in the `Graphics` class and so can be used in the `Graphics2D` class.

CubicCurve2D

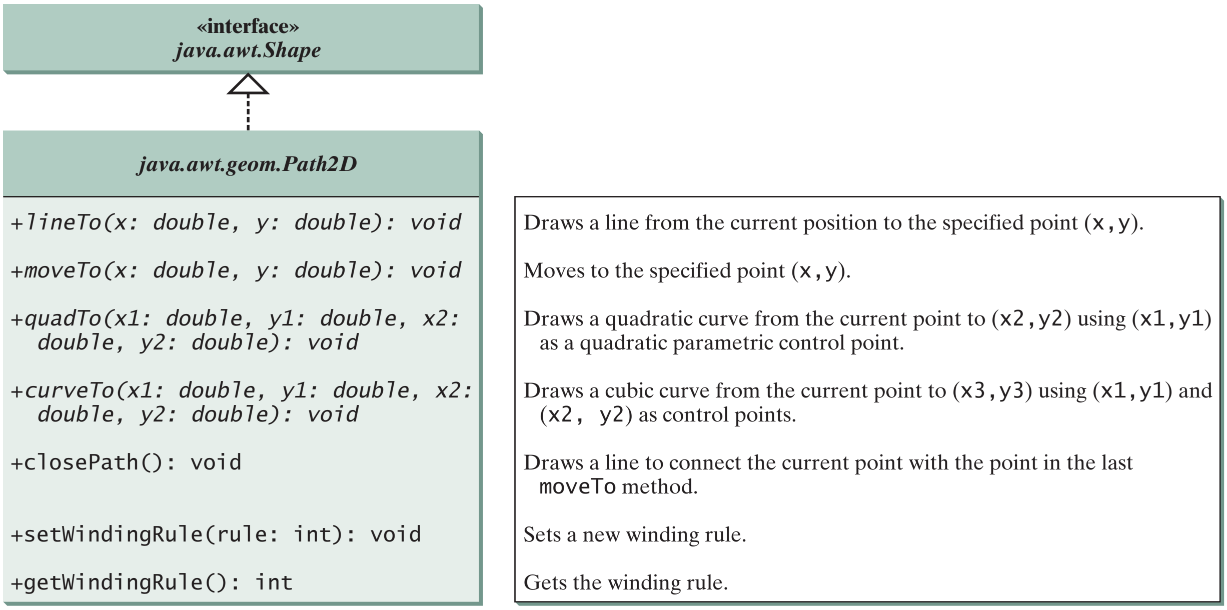
The statement `new CubicCurve2D.Double(200, 80, 240, 20, 350, 156, 450, 80))` (lines 22–23) creates an instance of `CubicCurve2D.Double`, which is also an instance of `QuadCurve2D` and `Shape`. The instance models a quadratic curves with two endpoints (`200`, `80`), (`450`, `80`) and two control points (`240`, `20`), (`450`, `80`).

44.9 Path2D

The `Path2D` class models an arbitrary geometric path. `Path2D.Double` and `Path2D.Float` are two concrete subclasses of `Path2D`. Java 2D also contains the `GeneralPath` class which is now superseded by `Path2D.Float`.

You can construct path segments using the methods, as shown in Figure 44.21.

You may create a `Path2D` using a constructor from `Path2D.Double` and `Path2D.Float`. The process of the path construction can be viewed as drawing with a pen. At any moment, the pen has a current position. You can use the `moveTo(x, y)` method to move the pen to the new position at point (`x`, `y`), use the `lineTo(x, y)` to add a point (`x`, `y`) to the path by drawing a straight line from the current point to this new point, use the `quadTo(ctrlx, ctrly, x, y)` method to draw a quadratic curve from the current location to (`x`, `y`) using (`ctrlx`, `ctrly`) as the control point, use the `curveTo(ctrlx1, ctrly1, ctrlx2, ctrly2, x, y)` method to draw a cubic curve from the current location to (`x`, `y`) using (`ctrlx1`, `ctrly1`) and (`ctrlx2`, `ctrly2`) as the control points, and use the `closePath()` method to connect the current point with the point in the last `moveTo` method.



+lineTo(x: double, y: double): void	Draws a line from the current position to the specified point (x,y).
+moveTo(x: double, y: double): void	Moves to the specified point (x,y).
+quadTo(x1: double, y1: double, x2: double, y2: double): void	Draws a quadratic curve from the current point to (x2,y2) using (x1,y1) as a quadratic parametric control point.
+curveTo(x1: double, y1: double, x2: double, y2: double): void	Draws a cubic curve from the current point to (x3,y3) using (x1,y1) and (x2, y2) as control points.
+closePath(): void	Draws a line to connect the current point with the point in the last moveTo method.
+setWindingRule(rule: int): void	Sets a new winding rule.
+getWindingRule(): int	Gets the winding rule.

FIGURE 44.21 The Path2D class contains the methods for constructing path segments.

Listing 44.8 gives a program that demonstrates how to draw a shape using Path2D. Figure 44.22 shows a sample run of the program.

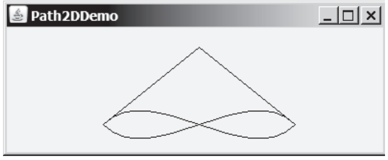


FIGURE 44.22 You can draw an arbitrary shape using the Path2D class.

LISTING 44.8 Path2DDemo.java

```java
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class Path2DDemo extends JApplet {
  public Path2DDemo() {
    add(new ShapePanel());
  }

  class ShapePanel extends JPanel {
    protected void paintComponent(Graphics g) {
      super.paintComponent(g);

      Graphics2D g2d = (Graphics2D)g;
      Path2D path = new Path2D.Double();
      path.moveTo(100, 100);
      path.curveTo(150, 50, 250, 150, 300, 100);
      path.moveTo(100, 100);
```

import for shape classes

applet

Graphics2D reference

new position

draw a cubic curve

new position

draw a cubic curve
draw a line
close path

display path

main method omitted

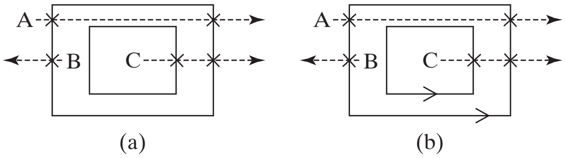
```
19      path.curveTo(150, 150, 250, 50, 300, 100);
20      path.lineTo(200, 20);
21      path.closePath();
22
23      g2d.draw(path);
24    }
25  }
26 }
```

The statement **new Path2D.Double()** (line 15) creates an empty path. The **moveTo(100, 100)** method (line 16) sets the current pen position at (**100**, **100**). Invoking **path.curveTo(150, 50, 250, 150, 300, 100)** (line 17) creates a cubic curve from (**100**, **100**) to (**300**, **100**) with control points (**150**, **50**) and (**250**, **150**). Invoking **path.moveTo(100, 100)** (line 18) moves the pen position back to (**100**, **100**). Invoking **path.curveTo(150, 150, 250, 50, 300, 100)** (line 19) creates a cubic curve from (**100**, **100**) to (**300**, **100**) with control points (**150**, **150**) and (**250**, **50**). Now the current position is at (**300**, **100**). Invoking **path.lineTo(200, 20)** (line 20) creates a line from (**300**, **100**) to (**200**, **20**). Invoking **path.closePath()** (line 21) draws a line connecting the current position (i.e., (**200**, **20**)) with the last **moveTo** position (i.e., (**100**, **100**)). Finally, Invoking **g2d.draw(path)** (line 23) draws the path.

For a simple shape, it is easy to decide which point is inside a shape. A path may form many shapes. It is not easy to decide which point is inside an enclosed path. Java 2D uses the winding rules to define the interior points. There are two winding rules: **WIND_EVEN_ODD** and **WIND_NON_ZERO**.

WIND_EVEN_ODD

The **WIND_EVEN_ODD** rule defines a point as inside a path if a ray from the point toward infinity in an arbitrary direction intersects the path an odd number of times. Consider the path in Figure 44.23(a). Points **A** and **C** are outside the path, because the ray intersects the path twice. Point **B** is inside the path, because the ray intersects the path once.



(a)

(b)

Figure 44.23 The **WIND_EVEN_ODD** and **WIND_NON_ZERO** rules define interior points.

WIND_NON_ZERO

With the **WIND_NON_ZERO** rule, the direction of the path is taken into consideration. A point is inside a path if a ray from the point toward infinity in an arbitrary direction intersects the path an unequal number of opposite directions. Consider the path in Figure 44.23(b). Point **A** is outside the path, because the ray intersects the path twice in opposite directions. Point **B** is inside the path, because the ray intersects the path once. Point **C** is inside the path, because the ray intersects the path twice in the same directions. By default, a **Path2D** is created using the **WIND_NON_ZERO** rule. You can use the **setWindingRule** method to set a new winding rule.

Listing 44.9 gives a program that demonstrates winding rules in **Path2D**. Figure 44.24 shows a sample run of the program.

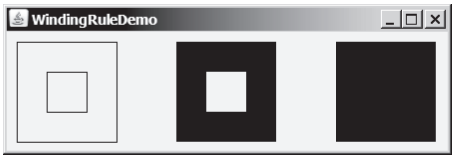


Figure 44.24 The winding rule defines the interior points.

LISTING 44.9 WindingRuleDemo.java

```
 1 import java.awt.*;
 2 import java.awt.geom.*;
 3 import javax.swing.*;
 4
 5 public class WindingRuleDemo extends JApplet {
 6   public WindingRuleDemo() {
 7     add(new ShapePanel());
 8   }
 9
10   class ShapePanel extends JPanel {
11     protected void paintComponent(Graphics g) {
12       super.paintComponent(g);
13
14       Graphics2D g2d = (Graphics2D)g; // Get Graphics2D
15
16       g2d.translate(10, 10); // Translate to a new origin
17       g2d.draw(createAPath()); // Create and draw a path
18
19       g2d.translate(160, 0); // Translate to a new origin
20       Path2D path2 = createAPath(); // Create a path
21       path2.setWindingRule(Path2D.WIND_EVEN_ODD); // Set a new rule
22       g2d.fill(path2); // Create and fill a path
23
24       g2d.translate(160, 0); // Translate to a new origin
25       Path2D path3 = createAPath(); // Create a path
26       path3.setWindingRule(Path2D.WIND_NON_ZERO); // Set a new rule
27       g2d.fill(path3); // Create and fill a path
28     }
29
30     private Path2D createAPath() {
31       // Define the outer rectangle
32       Path2D path = new Path2D.Double();
33       path.moveTo(0, 0);
34       path.lineTo(0, 100);
35       path.lineTo(100, 100);
36       path.lineTo(100, 0);
37       path.lineTo(0, 0);
38
39       // Define the inner rectangle
40       path.moveTo(30, 30);
41       path.lineTo(30, 70);
42       path.lineTo(70, 70);
43       path.lineTo(70, 30);
44       path.lineTo(30, 30);
45
46       return path;
47     }
48   }
49 }
```

import for shape classes

applet

Graphics2D reference

new origin

draw path

new origin

create a path

new winding rule

fill path

new origin

create a path

new winding rule

fill path

create a path

main method omitted

createAPath

The **createAPath()** method creates a path for two rectangles. The outer rectangle is created in lines 33–37 and the inner rectangle in lines 40–44.

The program translates the coordinate's origin to (**10**, **10**) in line 16, invokes **createAPath** to create a path, and displays it in line 17.

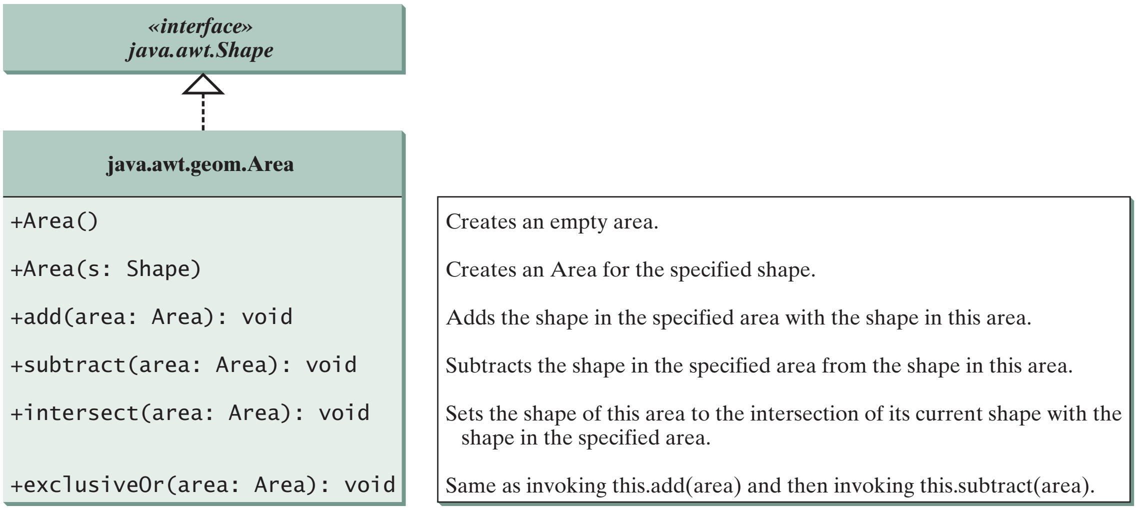
The program translates the coordinate's origin to (**160**, **0**) in line 19, creates a new path (line 20), sets the path winding rule to **WIND_EVEN_ODD** (line 21), and displays it in line 22.

The program translates the coordinate's origin to (160, 0) in line 24, creates a new path (line 25), sets the path winding rule to WIND_NON_ZERO (line 26), and displays it in line 27.

Note that if a path is unclosed, the fill method implicitly closes it and draws a filled path.

44.10 Constructive Area Geometry

Shapes can be combined to create new shapes. This is known as *constructive area geometry*. Java 2D provides class Area to perform constructive area geometry, as shown in Figure 44.25.



«interface»
java.awt.Shape

java.awt.geom.Area

+Area() — Creates an empty area.

+Area(s: Shape) — Creates an Area for the specified shape.

+add(area: Area): void — Adds the shape in the specified area with the shape in this area.

+subtract(area: Area): void — Subtracts the shape in the specified area from the shape in this area.

+intersect(area: Area): void — Sets the shape of this area to the intersection of its current shape with the shape in the specified area.

+exclusiveOr(area: Area): void — Same as invoking this.add(area) and then invoking this.subtract(area).

FIGURE 44.25 The Area class contains the methods for constructing new areas.

Area implements Shape and provides the methods add, subtract, intersect, and exclusiveOr to perform set-theoretic operations union, difference, intersection, and symmetric difference. These operations perform on the shapes stored in the areas. The union of two areas consists of all points that are in either area. The difference of two areas consists of the points that are in the first area, but not in the second area. The intersection of two areas consists of all points that are in both areas. The symmetric difference consists of the points that are in exactly one of the two areas.

Listing 44.10 gives a program that demonstrates constructive geometry using the Area class. Figure 44.26 shows a sample run of the program.

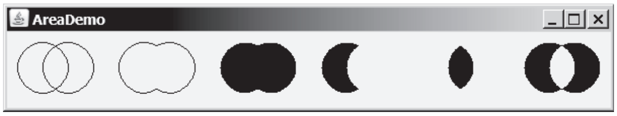


FIGURE 44.26 The Area class can be used to perform constructive geometry.

LISTING 44.10 AreaDemo.java

import for shape classes

applet

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class AreaDemo extends JApplet {
 6    public AreaDemo() {
 7      add(new ShapePanel());
 8    }
 9
10    class ShapePanel extends JPanel {
11      protected void paintComponent(Graphics g) {
12        super.paintComponent(g);
13
14        Graphics2D g2d = (Graphics2D)g; // Get Graphics2D
15
16        // Create two shapes
17        Shape shape1 = new Ellipse2D.Double(0, 0, 50, 50);
18        Shape shape2 = new Ellipse2D.Double(25, 0, 50, 50);
19        g2d.translate(10, 10); // Translate to a new origin
20        g2d.draw(shape1); // Draw the shape
21        g2d.draw(shape2); // Draw the shape
22
23        Area area1 = new Area(shape1); // Create an area
24        Area area2 = new Area(shape2);
25        area1.add(area2); // Add area2 to area1
26        g2d.translate(100, 0); // Translate to a new origin
27        g2d.draw(area1); // Draw the outline of the shape in the area
28
29        g2d.translate(100, 0); // Translate to a new origin
30        g2d.fill(area1); // Fill the shape in the area
31
32        area1 = new Area(shape1);
33        area1.subtract(area2); // Subtract area2 from area1
34        g2d.translate(100, 0); // Translate to a new origin
35        g2d.fill(area1); // Fill the shape in the area
36
37        area1 = new Area(shape1);
38        area1.intersect(area2); // Intersection of area2 with area1
39        g2d.translate(100, 0); // Translate to a new origin
40        g2d.fill(area1); // Fill the shape in the area
41
42        area1 = new Area(shape1);
43        area1.exclusiveOr(area2); // Exclusive or of area2 with area1
44        g2d.translate(100, 0); // Translate to a new origin
45        g2d.fill(area1); // Fill the shape in the area
46      }
47    }
48  }
```

Graphics2D reference

two shapes

new origin

draw shapes

add

fill area

subtract

fill area

intersect

fill area

exclusiveOr

fill area

main method omitted

The program creates two ellipses (lines 17–18) and displays them (lines 20–21). The program creates two areas and invokes **add** (line 25), **subtract** (line 33), **intersect** (line 38), and **exclusiveOr** (line 43) to perform constructive area geometry.

Key Terms

constructive area geometry	44–22
cubic curves	44–17
gradient paint	44–13
quadratic curves	44–17
rotation	44–6
scaling	44–10
stroke	44–11
texture paint	44–16
translation	44–8
WIND_EVEN_ODD	44–20
WIND_NON_ZERO	44–20

Chapter Summary

1. The Java 2D API provides the `java.awt.Graphics2D` class, which extends `java.awt.Graphics` with advanced capabilities for rendering graphics.

2. The Java 2D API provides an object-oriented approach that separates rendering from modeling. All shapes are defined under the `Shape` interface.

3. Classes `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Arc2D`, `Ellipses2D`, `QuadCurve2D`, `CubicCurve2D`, and `Path2D` are abstract classes. Each contains two concrete static inner classes named `Double` and `Float` for `double` and `float` coordinates, respectively. The inner classes are subclasses of their respective abstract classes.

4. A point can be modeled using the abstract `Point2D` class. It contains two concrete static inner classes `Point2D.Double` and `Point2D.Float`, which are subclasses of `Point2D`.

5. The `Graphics2D` class is for rendering shapes. You can invoke its `draw(Shape)` method to render the boundary of the shape and `fill(Shape)` method to fill the interior of the shape.

6. You can use the `translate(double x, double y)` method in the `Graphics` class to move the subsequent rendering by the specified distance relative to the previous position.

7. You can use the `rotate(double theta)` method in the `Graphics2D` class to rotate subsequent rendering by `theta` degrees from the origin, where `theta` is a double value in radians.

8. You can use the `scale(double sx, double sy)` method in the `Graphics2D` class to resize subsequent rendering by the specified scaling factors.

9. Java 2D allows you to specify the attributes of lines, called *strokes*.

10. You can specify the width of the line, how the line ends (called *end caps*), how lines join together (called *line joins*), and whether the line is dashed. These attributes are defined in a `Stroke` object.

11. You can create a `Stroke` object using the `BasicStroke` class.

12. To set a stroke, use the `setStroke(Stroke)` method in the `Graphics2D` class.

13. `Graphics2D` provides the `setPaint(Paint)` method to set a paint. `Paint` is a generalization of color. It has more attributes than simple solid colors.

14. `GradientPaint` defines a varying color, specified by two points and two colors. As the location moves from the first point to the second, the paint changes gradually from the first color to the second.

15. A `GradientPaint` can cyclic or acyclic. A cyclic paint repeats the same pattern periodically.

16. `TexturePaint` defines an image to fill a shape or characters. A texture paint is defined by an image anchored in a rectangle.

17. Java 2D provides the `QuadCurve2D` and `CubicCurve2D` classes for modeling quadratic curves and cubic curves.

18. A quadratic curve is mathematically defined as a quadratic polynomial.

19. A cubic curve is mathematically defined as a cubic polynomial.

20. The `Path2D` class models an arbitrary geometric path. `Path2D.Double` and `Path 2D.Float` are two concrete subclasses of `Path 2D`.

21. The winding rule defines interior points in a path.

22. The `WIND_EVEN_ODD` rule defines a point as inside a path if a ray from the point toward infinity in an arbitrary direction intersects the path an odd number of times.

23. With the `WIND_NON_ZERO` rule, the direction of the path is taken into consideration. A point is inside a path if a ray from the point toward infinity in an arbitrary direction intersects the path an unequal number of opposite directions.

24. Java 2D provides class `Area` to perform constructive area geometry.

25. `Area` implements `Shape` and provides the methods `add`, `subtract`, `intersect`, and `exclusiveOr` to perform set-theoretic operations union, difference, intersection, and symmetric difference.

Review Questions

Sections 44.2–44.3

44.1 How do you obtain a reference to a `Graphics2D` object?

44.2 List some methods defined in the `Shape` interface.

44.3 How do you create a `Line2D` object?

44.4 Are `Line2D.Double` and `Line2D.Float` inner classes of `Line2D`? Are they also subclasses of `Line2D`?

44.5 How do you render a `Shape` object?

44.6 What are the relationships among `Point2D`, `Point2D.Double`, `Point2D.Float`, and `Point`? Check Java API to see what methods are defined in `Point2D`.

Section 44.4

44.7 What are the relationships among `Rectangle2D`, `Rectangle2D.Double`, `Rectangle2D.Float`, and `Rectangle`?

44.8 You can draw basic shapes such as lines, rectangles, ellipses, and arcs using the drawing/filling methods in the `Graphics` class or create a `Shape` object and render them using the `draw(Shape)` or `fill(Shape)`. What are the advantages of using the latter?

Section 44.5

44.9 Suppose a rectangle is created using `new Rectangle2D.Double(2, 3, 4, 5)`. Where is it displayed after applying `g2d.translate(10, 10)` and `g2d.draw(rectangle)`?

44.10 Suppose a rectangle is created using `new Rectangle2D.Double(2, 3, 4, 5)`. Where is it displayed after applying `g2d.rotate(Math.PI / 5)` and `g2d.draw(rectangle)`?

44.11 Suppose a rectangle is created using `new Rectangle2D.Double(2, 3, 4, 5)`. Where is it displayed after applying `g2d.scale(10, 10)` and `g2d.draw(rectangle)`?

Sections 44.6–44.7

44.12 How do you create a Stroke and set a stroke in Graphics2D?

44.13 How do you create a Paint and set a paint in Graphics2D?

44.14 What is a gradient paint? How do you create a GradientPaint?

44.15 What is a texture paint? How do you create a TexturePaint?

Sections 44.8–44.10

44.16 How do you create a QuadCurve2D? How do you create a CubicCurve2D?

44.17 Describe the methods in Path2D?

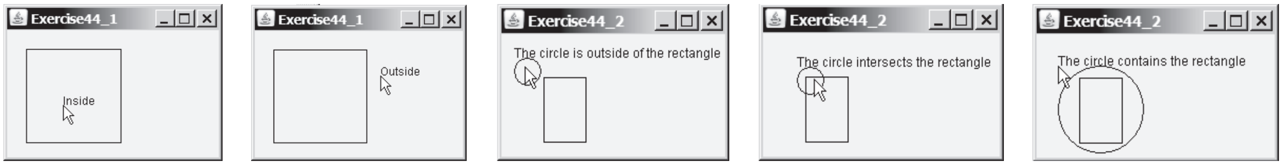
44.18 What is the winding rule? What is WIND_EVEN_ODD? What is WIND_NON_ZERO?

44.19 How do you create an Area from a shape? Describe the add, subtract, intersect, and exclusiveOr methods in the Area class.

Programming Exercises

Section 44.4

44.1* (*Inside a rectangle?*) Write a program that displays a rectangle with upper-left corner point at (20, 20), width 100, and height 100. Whenever you move the mouse, display a message indicating whether the mouse point is inside the rectangle, as shown in Figure 44.27(a)–(b).



The circle is outside of the rectangle

The circle intersects the rectangle

The circle contains the rectangle

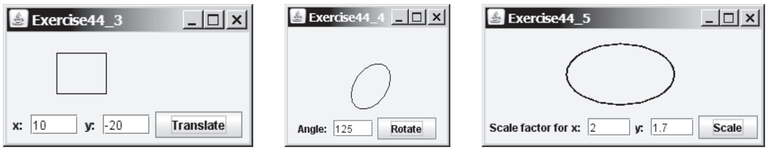
(a) (b) (c) (d) (e)

Figure 44.27 (a)–(b) Exercise 44.1 detects whether a point is inside a rectangle. (c)–(e) Exercise 44.2 detects whether a circle contains, intersects, or is outside a rectangle.

44.2* (*Contains, intersects, or outside?*) Write a program that displays a rectangle with upper-left corner point at (40, 40), width 40, and height 60. Display a circle. The circle's upper-left corner of the bounding rectangle is at the mouse point. pressing the up/down arrow key increases/decreases the circle radius by 5 pixels by. Display a message at the mouse point to indicate whether the circle contains, intersects, or is outside of the rectangle, as shown in Figure 44.27(c)–(e).

44.3* (*Translation*) Write a program that displays a rectangle with upper-left corner point at (40, 40), width 50, and height 40. Enter the values in the text fields x and y and press the *Translate* button to translate the rectangle to a new location, as shown in Figure 44.28(a).

44.4* (*Rotation*) Write a program that displays an ellipse. The center of the ellipse is at (0, 0) with width 60 and height 40. Use the translate method to move the origin to (100, 70). Enter the value in the text field Angle and press the *Rotate* button to rotate the ellipse to a new location, as shown in Figure 44.28(b).



x: 10 y: -20

Angle: 125

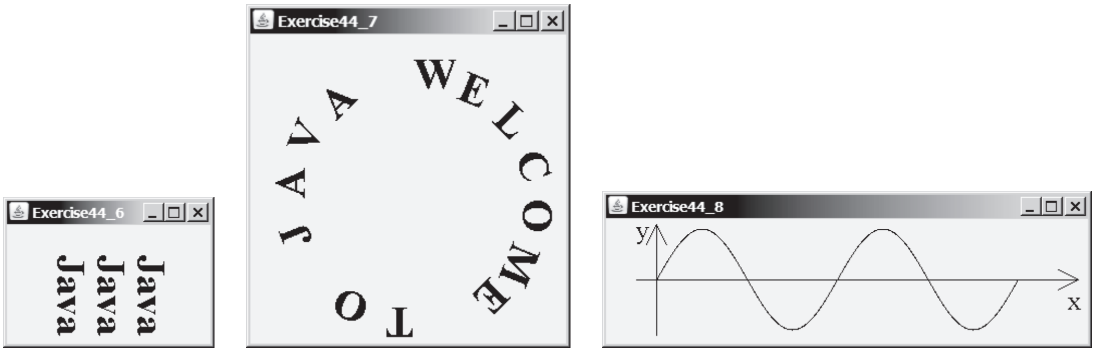
Scale factor for x: 2 y: 1.7

(a) (b) (c)

FIGURE 44.28 (a) Exercise 44.3 translates coordinates. (b) Exercise 44.4 rotates coordinates. (c) Exercise 44.5 scales coordinates.

44.5* (*Scaling*) Write a program that displays an ellipse. The center of the ellipse is at (`0`, `0`) with width `60` and height `40`. Use the `translate` method to move the origin to (`150`, `50`). Enter the scaling factors in the text fields and press the *Scale* button to scale the ellipse, as shown in Figure 44.28(c).

44.6* (*Vertical strings*) Write a program that displays three strings vertically, as shown in Figure 44.29(a).



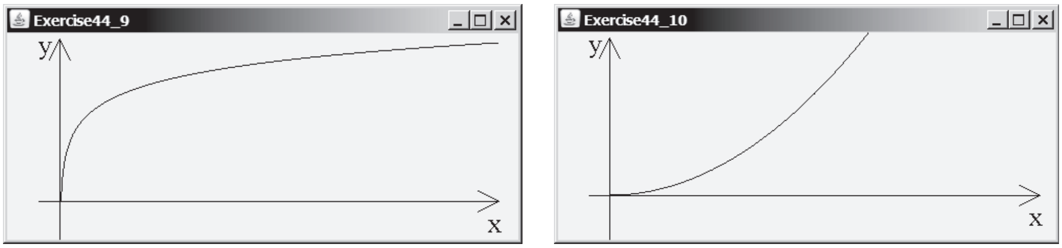
(a) (b) (c)

FIGURE 44.29 (a) Exercise 44.6 displays strings vertically. (b) Exercise 4.7 displays characters around the circle. (c) Exercise 44.8 displays a sine function.

44.7* (*Characters around circle*) Write a program that displays a string around the circle, as shown in Figure 44.29(b).

44.8* (*Plotting the sine function*) Write a program that plots the sine function, as shown in Figure 44.29(c).

44.9* (*Plotting the log function*) Write a program that plots the log function, as shown in Figure 44.30(a).



(a) (b)

FIGURE 44.30 (a) Exercise 44.9 displays the log function. (b) Exercise 4.10 displays the n^2 function.

44.10* (*Plotting the n^2 function*) Write a program that plots the n^2 function, as shown in Figure 44.30(b).

44.11* (*Plotting the log, n, nlogn, and n^2 functions*) Write a program that plots the log, *n*, *n*log*n*, and n^2 functions, as shown in Figure 44.31(a).

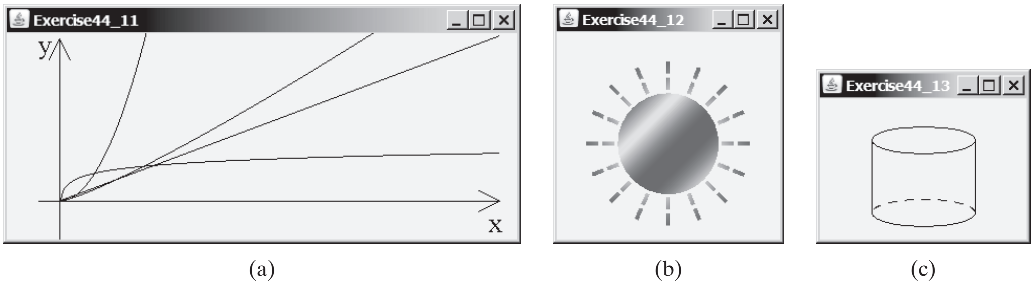


FIGURE 44.31 (a) Exercise 44.11 displays several functions. (b) Exercise 4.12 displays the sunshine. (c) Exercise 44.13 displays a cylinder.

44.12* (*Sunshine*) Write a program that displays a circle filled with a gradient color to animate a sun and display light rays coming out from the sun using dashed lines, as shown in Figure 44.31(b).

44.13* (*Displaying a cylinder*) Write a program that displays a cylinder, as shown in Figure 44.31(c). Use dashed strokes to draw the dashed arc.

44.14* (*Filled cylinder*) Write a program that displays a filled cylinder, as shown in Figure 44.32(a).

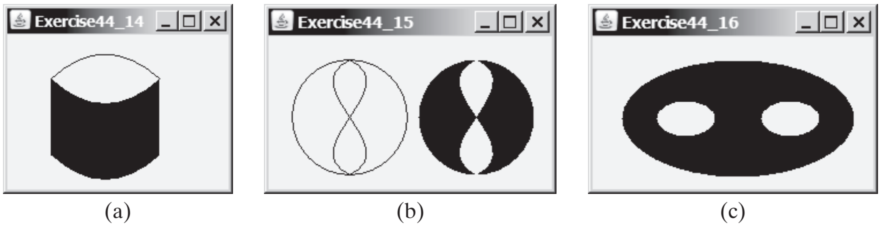


FIGURE 44.32 (a) Exercise 44.14 displays a filled cylinder. (b) Exercise 4.15 displays symmetric difference of two areas. (c) Exercise 4.16 displays two eyes.

44.15* (*Area geometry*) Write a program that creates two areas: a circle and a path consisting of two cubic curves. Draw the areas and fill the symmetric difference of the areas, as shown in Figure 44.32(b).

44.16* (*Eyes*) Write a program that displays two eyes in an oval, as shown in Figure 44.32(c).