JAVA 2D: GRAPHICS IN JAVA 2

Topics in This Chapter

- Drawing 2D shapes
- Tiling an image inside a shape
- Using local fonts
- Drawing with custom pen settings
- Changing the opaqueness of objects
- Translating and rotating coordinate systems

Chapter 10

Anyone who has even lightly ventured into developing detailed graphical programs with the Abstract Windowing Toolkit (AWT) has quickly realized that the capabilities of the `Graphics` object are rather limited—not surprisingly, since Sun developed the AWT over a short period when moving Java from embedded applications to the World Wide Web. Shortcomings of the AWT include limited available fonts, lines drawn with a single-pixel width, shapes painted only in solid colors, and the inability to properly scale drawings prior to printing.

Java 2D is probably the second most significant addition to the Java 2 Platform, surpassed only by the Swing GUI components. The Java 2D API provides a robust package of drawing and imaging tools to develop elegant, professional, high-quality graphics. The following important Java 2D capabilities are covered in this chapter:

- Colors and patterns: graphics can be painted with color gradients and fill patterns.
- Transparent drawing: opaqueness of a shape is controlled through an alpha transparency value.
- Local fonts: all local fonts on the platform are available for drawing text.
- Explicit control of the drawing pen: thickness of lines, dashing patterns, and segment connection styles are available.
- Transformations of the coordinate system—translations, scaling, rotations, and shearing—are available.

These exciting capabilities come at a price—the Java 2D API is part of the Java Foundation Classes introduced in Java 2. Thus, unlike Swing, which can be added to the JDK 1.1, you cannot simply add Java 2D to the JDK 1.1. The Java Runtime Environment (JRE) for the Java 2 Platform is required for execution of 2D graphical applications, and a Java 2-capable browser or the Java Plug-In, covered in Section 9.9 (The Java Plug-In), is required for execution of 2D graphical applets. Complete documentation of the Java 2D API, along with additional developer information, is located at `http://java.sun.com/products/java-media/2D/`. Also, the JDK 1.3 includes a 2D demonstration program located in the installation directory: `root/jdk1.3/demo/jfc/Java2D/`. In addition, Java 2D also supports high-quality printing; this topic is covered in Chapter 15 (Advanced Swing).

10.1 Getting Started with Java 2D

In Java 2, the `paintComponent` method is supplied with a `Graphics2D` object, which contains a much richer set of drawing operations than the AWT `Graphics` object. However, to maintain compatibility with Swing as used in Java 1.1, the declared type of the `paintComponent` argument is `Graphics` (`Graphics2D` inherits from `Graphics`), so you must first cast the `Graphics` object to a `Graphics2D` object before drawing. Technically, in Java 2, all methods that receive a `Graphics` object (`paint`, `paintComponent`, `getGraphics`) actually receive a `Graphics2D` object.

The traditional approach for performing graphical drawing in Java 1.1 is reviewed in Listing 10.1. Here, every AWT `Component` defines a `paint` method that is passed a `Graphics` object (from the `update` method) on which to perform drawing. In contrast, Listing 10.2 illustrates the basic approach for drawing in Java 2D. All Swing components call `paintComponent` to perform drawing. Technically, you can use the `Graphics2D` object in the AWT `paint` method; however, the `Graphics2D` class is included only with the Java Foundations Classes, so the best course is to simply perform drawing on a Swing component, for example, a `JPanel`. Possible exceptions would include direct 2D drawing in the `paint` method of a `JFrame`, `JApplet`, or `JWindow`, since these are heavyweight Swing components without a `paintComponent` method.

Listing 10.1 Drawing graphics in Java 1.1

```
public void paint(Graphics g) {
  // Set pen parameters
  g.setColor(someColor);
  g.setFont(someLimitedFont);

  // Draw a shape
  g.drawString(...);
  g.drawLine(...)
  g.drawRect(...);     // outline
  g.fillRect(...);     // solid
  g.drawPolygon(...);  // outline
  g.fillPolygon(...);  // solid
  g.drawOval(...);     // outline
  g.fillOval(...);     // solid
  ...
}
```

Listing 10.2 Drawing graphics in the Java 2 Platform

```
public void paintComponent(Graphics g) {
  // Clear background if opaque
  super.paintComponent(g);
  // Cast Graphics to Graphics2D
  Graphics2D g2d = (Graphics2D)g;
  // Set pen parameters
  g2d.setPaint(fillColorOrPattern);
  g2d.setStroke(penThicknessOrPattern);
  g2d.setComposite(someAlphaComposite);
  g2d.setFont(anyFont);
  g2d.translate(...);
  g2d.rotate(...);
  g2d.scale(...);
  g2d.shear(...);
  g2d.setTransform(someAffineTransform);
  // Allocate a shape
  SomeShape s = new SomeShape(...);
  // Draw shape
  g2d.draw(s);  // outline
  g2d.fill(s);  // solid
}
```

The general approach for drawing in Java 2D is outlined as follows.

Cast the Graphics object to a Graphics2D object.

Always call the `paintComponent` method of the superclass first, because the default implementation of Swing components is to call the `paint` method of the associated `ComponentUI`; this approach maintains the component look and feel. In addition, the default `paintComponent` method clears the off-screen pixmap because Swing components implement double buffering. Next, cast the `Graphics` object to a `Graphics2D` object for Java 2D drawing.

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.doSomeStuff(...);
  ...
}
```

Core Approach

When overriding the `paintComponent` method of a Swing component, always call `super.paintComponent`.

Modify drawing parameters (optional).

Drawing parameters are applied to the `Graphics2D` object, not to the `Shape` object. Changes to the graphics context (`Graphics2D`) apply to every subsequent drawing of a `Shape`.

```
g2d.setPaint(fillColorOrPattern);
g2d.setStroke(penThicknessOrPattern);
g2d.setComposite(someAlphaComposite);
g2d.setFont(someFont);
g2d.translate(...);
g2d.rotate(...);
g2d.scale(...);
g2d.shear(...);
g2d.setTransform(someAffineTransform);
```

Create a Shape object.

```
Rectangle2D.Double rect = ...;
Ellipse2D.Double ellipse = ...;
Polygon poly = ...;
GeneralPath path = ...;
// Satisfies Shape interface
SomeShapeYouDefined shape = ...;
```

Draw an outlined or filled version of the Shape.
Pass in the `Shape` object to either the `draw` or `fill` method of the `Graphics2D` object. The graphic context (any paint, stroke, or transform applied to the `Graphics2D` object) will define exactly how the shape is drawn or filled.

```
g2d.draw(someShape);
g2d.fill(someShape);
```

The `Graphics2D` class extends the `Graphics` class and therefore inherits all the familiar AWT graphic methods covered in Section 9.11 (Graphics Operations). The `Graphics2D` class adds considerable functionality to drawing capabilities. Methods that affect the appearance or transformation of a `Shape` are applied to the `Graphics2D` object. Once the graphics context is set, all subsequent `Shapes` that are drawn will undergo the same set of drawing rules. Keep in mind that the methods that alter the coordinate system (`rotate`, `translate`, `scale`) are cumulative.

Useful Graphics2D Methods

The more common methods of the `Graphics2D` class are summarized below.

public void draw(Shape shape)
This method draws an outline of the `shape`, based on the current settings of the `Graphics2D` context. By default, a shape is bound by a `Rectangle` with the upper-left corner positioned at (0,0). To position a shape elsewhere, first apply a transformation to the `Graphics2D` context: `rotate`, `transform`, `translate`.

public boolean drawImage(BufferedImage image,
BufferedImageOp filter,
int left, int top)
This method draws the `BufferedImage` with the upper-left corner located at (`left`, `top`). A `filter` can be applied to the image. See Section 10.3 (Paint Styles) for details on using a `BufferedImage`.

public void drawString(String s, float left, float bottom)
The method draws a string in the bottom-left corner of the specified location, where the location is specified in floating-point units. The Java 2D API does not provide an overloaded `drawString` method that supports `double` arguments. Thus, the method call `drawString(s, 2.0, 3.0)` will not compile. Correcting the error requires explicit statement of floating-point, literal arguments, as in `drawString(s, 2.0f, 3.0f)`.

Java 2D supports fractional coordinates to permit proper scaling and transformations of the coordinate system. Java 2D objects live in the User Coordinate Space where the axes are defined by floating-point units. When the graphics are rendered on the screen or a printer, the User Coordinate Space is transformed to the Device Coordinate Space. The transformation maps 72 User Coordinate Space units to one physical inch on the output device. Thus, before the graphics are rendered on the physical device, fractional values are converted to their nearest integral values.

public void fill(Shape shape)

This method draws a solid version of the `shape`, based on the current settings of the `Graphics2D` context. See the `draw` method for details of positioning.

public void rotate(double theta)

This method applies a rotation of theta *radians* to the `Graphics2D` transformation. The point of rotation is about (`x`, `y`)=(0, 0). This rotation is *added* to any existing rotations of the `Graphics2D` context. See Section 10.7 (Coordinate Transformations).

public void rotate(double theta, double x, double y)

This method also applies a rotation of theta *radians* to the `Graphics2D` transformation. However, the point of rotation is about (`x`, `y`). See Section 10.7 (Coordinate Transformations) for details.

public void scale(double xscale, yscale)

This method applies a linear scaling to the x- and y-axis. Values greater than 1.0 expand the axis, and values less than 1.0 shrink the axis. A value of `-1` for `xscale` results in a mirror image reflected across the x-axis. A `yscale` value of `-1` results in a reflection about the y-axis.

public void setComposite(Composite rule)

This method specifies how the pixels of a new shape are combined with the existing background pixels. You can specify a custom composition `rule` or apply one of the predefined `AlphaComposite` rules: `AlphaComposite.Clear`, `AlphaComposite.DstIn`, `AlphaComposite.DstOut`, `AlphaComposite.DstOver`, `AlphaComposite.Src`, `AlphaComposite.SrcIn`, `AlphaComposite.SrcOut`, `AlphaComposite.ScrOver`.

To create a custom `AlphaComposite` rule, call `getInstance` as in

```
g2d.setComposite(AlphaComposite.SrcOver);
```

or

```
int type = AlphaComposite.SRC_OVER;
float alpha = 0.75f;
AlphaComposite rule =
   AlphaComposite.getInstance(type, alpha);
g2d.setComposite(rule);
```

The second approach permits you to set the alpha value associated with composite rule, which controls the transparency of the shape. By default, the transparency value is 1.0f (opaque). See Section 10.4 (Transparent Drawing) for details. Clarification of the mixing rules is given by T. Porter and T. Duff in "Compositing Digital Images," *SIGGRAPH* **84**, pp. 253–259.

public void setPaint(Paint paint)

This method sets the painting style of the `Graphics2D` context. Any style that implements the `Paint` interface is legal. Existing styles in the Java 2 Platform include a solid `Color`, a `GradientPaint`, and a `TexturePaint`.

public void setRenderingHints(Map hints)

This method allows you to control the quality of the 2D drawing. The AWT includes a `RenderingHints` class that implements the `Map` interface and provides a rich suite of predefined constants. Quality aspects that can be controlled include antialiasing of shape and text edges, dithering and color rendering on certain displays, interpolation between points in transformations, and fractional text positioning. Typically, antialiasing is turned on, and the image rendering is set to quality, not speed:

```
RenderingHints hints = new RenderingHints(
            RenderingHints.KEY_ANTIALIASING,
            RengeringHints.VALUE_ANTIALIAS_ON);
hints.add(new RenderingHints(
            RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY));
```

public void setStroke(Stroke pen)

The `Graphics2D` context determines how to draw the outline of a shape, based on the current `Stroke`. This method sets the drawing `Stroke` to the behavior defined by `pen`. A user-defined `pen` must implement the `Stroke` interface. The AWT includes a `BasicStroke` class to define the end styles of a line segment, to specify the joining styles of line segments, and to create dashing patterns. See Section 10.6 (Stroke Styles) for details.

public void transform(AffineTransform matrix)

This method applies the Affine transformation, `matrix`, to the existing transformation of the `Graphics2D` context. The Affine transformation can include both a translation and a rotation. See Section 10.7 (Coordinate Transformations).

public void translate(double x, double y)
This method translates the origin by (x, y) *units*. This translation is added to any prior translations of the Graphics2D context. The units passed to the drawing primitives initially represent 1/72nd of an inch, which on a monitor, amounts to one pixel. However, on a printer, one unit might map to 4 or 9 pixels (300 dpi or 600 dpi).

public void setPaintMode()
This method overrides the setPaintMode method of the Graphics object. This implementation also sets the drawing mode back to "normal" (vs. XOR) mode. However, when applied to a Graphics2D object, this method is equivalent to setComposite(AlphaComposite.SrcOver), which places the source shape on top of the destination (background) when drawn.

public void setXORMode(Color color)
This method overrides the setXORMode for the Graphics object. For a Graphics2D object, the setXORMode method defines a new compositing rule that is outside the eight predefined Porter-Duff alpha compositing rules (see Section 10.4). The XOR compositing rule does not account for transparency (alpha) values and is calculated by a bitwise XORing of the source color, destination color, and the passed-in XOR color. Using XOR twice in a row when you are drawing a shape will return the shape to the original color. The transparency (alpha) value is ignored under this mode, and the shape will always be opaque. In addition, antialiasing of shape edges is not supported under XOR mode.

10.2 Drawing Shapes

With the AWT, you generally drew a shape by calling the drawXxx or fillXxx method of the Graphics object. In Java 2D, you generally create a Shape object, then call either the draw or fill method of the Graphics2D object, supplying the Shape object as an argument. For example:

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  // Assume x, y, and diameter are instance variables.
  Ellipse2D.Double circle =
    new Ellipse2D.double(x, y, diameter, diameter);
  g2d.fill(circle);
  ...
}
```

Most of the Shape classes define both a Shape.Double and a Shape.Float version of the class. Depending on the version of the class, the coordinate locations are stored as either double precision numbers (Shape.Double) or single precision numbers (Shape.Float). The idea is that single precision coordinates might be slightly faster to manipulate on some platforms. You can still call the familiar drawXxx methods of the Graphics class if you like; the Graphics2D object inherits from the Graphics object. This approach is necessary for drawString and drawImage and possibly is convenient for draw3DRect.

Shape Classes

Arguments to the Graphics2D draw and fill methods must implement the Shape interface. You can create your own shapes, of course, but you can also use major built-in classes: Arc2D, Area, CubicCurve2D, Ellipse2D, GeneralPath, Line2D, QuadCurve2D, Rectangle2D, and RoundRectangle2D. Each of these classes is contained in the java.awt.geom package. Each of these classes, except for Area, Polygon, and Rectangle, has float and double constructors.

The classes Polygon and Rectangle, holdovers from Java 1.1, also implement the Shape interface. These two shapes are covered in Section 9.11 (Graphics Operations).

The most common constructors for these Shapes follow.

public Arc2D.Float(float left, float top, float width, float height, float startAngle, float deltaAngle, int closure)
public Arc2D.Double(double left, double top, double width, double height, double startAngle, double deltaAngle, int closure)

These constructors create an arc by selecting a portion of a full ellipse whose bounding rectangle has an upper-left corner located at the (left, top). The vertex of the arc (ellipse) is located at the origin of the bounding rectangle. The reference for the start angle is the positive x-axis. Angles are specified in *degrees* and represent *arc* degrees, not true degrees. Arc angles are defined such that the 45 degree line runs from the ellipse center to the upper-right corner of the bounding rectangle. The arc closure is one of Arc2D.CHORD, Arc2D.OPEN, or Arc2D.PIE.

public Area(Shape shape)

This constructor creates an Area with the given Shape. Areas support geometrical operations, for example: add, subtract, intersect, and exclusiveOr.

public CubicCurve2D.Float(float xStart, float yStart,
float pX, float pY,
float qX, float qY,
float xEnd, float yEnd)
public CubicCurve2D.Double(double xStart, double yStart,
double pX, double pY,
double qX, double qY,
double xEnd, double yEnd)

These constructors create a `CubicCurve2D` shape representing a curve (spline) from (`xStart`, `yStart`) to (`xEnd`, `yEnd`). The curve has two control points (`pX`, `pY`) and (`qX`, `qY`) that impact the curvature of the line segment joining the two end points.

public Ellipse2D.Float(float left, float top, float width,
float height)
public Ellipse2D.Double(double left, double top,
double width, double height)

These constructors create an ellipse bounded by a rectangle of dimension `width` by `height`. The `Ellipse2D` class inherits from the `RectangularShape` class and contains the same methods as common to `Rectangle2D` and `RoundRectangle2D`.

public GeneralPath()

A `GeneralPath` is an interesting class because you can define all the line segments to create a brand-new `Shape`. This class supports a handful of methods to add lines and Bézier (cubic) curves to the path: `closePath`, `curveTo`, `lineTo`, `moveTo`, and `quadTo`. Appending a path segment to a `GeneralPath` without first performing an initial `moveTo` generates an `IllegalPathStateException`. An example of creating a `GeneralPath` follows:

```
GeneralPath path = new GeneralPath();
path.moveTo(100,100);
path.lineTo(300,205);
path.quadTo(205,250,340,300);
path.lineTo(340,350);
path.closePath();
```

public Line2D.Float(float xStart, float yStart, float xEnd,
float yEnd)
public Line2D.Double(double xStart, double yStart,
double xEnd, double yEnd)

These constructors create a `Line2D` shape representing a line segment from (`xStart`, `yStart`) to (`xEnd`, `yEnd`).

public Line2D.Float(Point p1, Point p2)
public Line2D.Double(Point p1, Point p2)

These constructors create a `Line2D` shape representing a line segment from `Point p1` to `Point p2`.

public QuadCurve2D.Float(float xStart, float yStart,
float pX, double pY,
float xEnd, float yEnd)
public QuadCurve2D.Double(double xStart, double yStart,
double pX, double pY,
double xEnd, double yEnd)

These constructors create a `Shape` representing a curve from (`xStart`, `yStart`) to (`xEnd`, `yEnd`). The point (`pX`, `pY`) represents a control point impacting the curvature of the line segment connecting the two end points.

public Rectangle2D.Float(float top, float left, float width,
float height)
public Rectangle2D.Double(double top, double left,
double width, double height)

These constructors create a `Rectangle2D` shape with the upper-left corner located at (`top`, `left`) and a dimension of `width` by `height`.

public RoundRectangle2D.Float(float top, float left,
float width, float height,
float arcX, float arcY)
public RoundRectangle2D.Double(double top, double left,
double width, double height,
double arcX, double arcY)

These two constructors create a `RectangleShape` with rounded corners. The upper-left corner of the rectangle is located at (`top`, `left`), and the dimension of the rectangle is `width` by `height`. The arguments `arcX` and `arcY` represent the distance from the rectangle corners (in the respective x direction and y direction) at which the rounded curve of the corners start.

An example of drawing a circle (`Ellispse2D` with equal width and height) and a rectangle (`Rectangle2D`) is presented in Listing 10.3. Here, the circle is filled completely, and an outline of the rectangle is drawn, both based on the default context settings of the `Graphics2D` object. Figure 10–1 shows the result. The method `getCircle` plays a role in other examples throughout this chapter. `ShapeExample` uses `WindowUtilities` in Listing 14.1 and `ExitListener` in Listing 14.2 to create a closable `JFrame` container for the drawing panel.

Most of the code examples throughout this chapter are presented as Java applications. To convert the examples to applets, follow the given template:

```
import java.awt.*;
import javax.swing.*;
public class YourApplet extends JApplet {
  public void init() {
    JPanel panel = new ChapterExample();
    panel.setBackground(Color.white);
    getContentPane().add(panel);
  }
}
```

The basic idea is to create a `JApplet` and `add` the chapter example, which is implemented as a `JPanel`, to the `contentPane` of the `JApplet`. Depending on the particular example you are converting, you may need to set the background color of the `JPanel`. Once the corresponding HTML file is created (with an applet of the same dimensions as the original `JFrame`), you can either use appletviewer or convert the HTML file to support the Java Plug-In. See Section 9.9 (The Java Plug-In) for details on converting the HTML file.

Listing 10.3 `ShapeExample.java`

```
import javax.swing.*;   // For JPanel, etc.
import java.awt.*;      // For Graphics, etc.
import java.awt.geom.*; // For Ellipse2D, etc.

/** An example of drawing/filling shapes with Java 2D in
 *  Java 1.2 and later.
 */

public class ShapeExample extends JPanel {
  private Ellipse2D.Double circle =
    new Ellipse2D.Double(10, 10, 350, 350);
  private Rectangle2D.Double square =
    new Rectangle2D.Double(10, 10, 350, 350);

  public void paintComponent(Graphics g) {
    clear(g);
    Graphics2D g2d = (Graphics2D)g;
    g2d.fill(circle);
    g2d.draw(square);
  }
```

(continued)

Listing 10.3 `ShapeExample.java` *(continued)*

```
  // super.paintComponent clears off screen pixmap,
  // since we're using double buffering by default.
  protected void clear(Graphics g) {
    super.paintComponent(g);
  }

  protected Ellipse2D.Double getCircle() {
    return(circle);
  }

public static void main(String[] args) {
    WindowUtilities.openInJFrame(new ShapeExample(), 380, 400);
  }
}
```

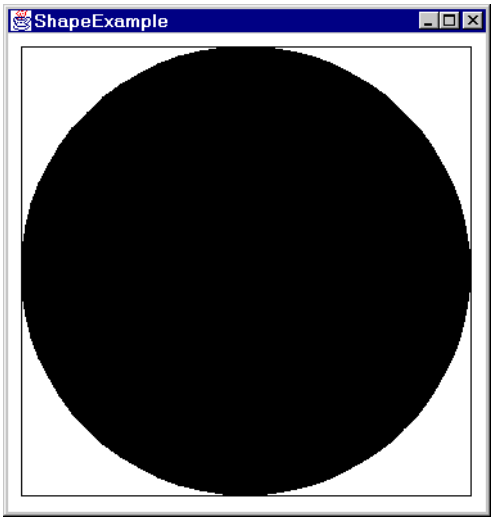

Figure 10–1 An ellipse (circle) drawn with a box outline in Java 2D.

10.3 Paint Styles

When you fill a `Shape`, the `Graphics2D` object uses the settings associated with the internal `Paint` attribute. The `Paint` setting can be a `Color` (solid color), a `GradientPaint` (gradient fill gradually combining two colors), a `TexturePaint` (tiled image), or a new version of `Paint` that you write yourself. Use `setPaint` and `get-`

Paint to change and retrieve the Paint settings. Note that setPaint and getPaint supersede the setColor and getColor methods that were used in Graphics.

Paint Classes

Arguments to the Graphics2D setPaint method (and return values of getPaint) must implement the Paint interface. Here are the major built-in Paint classes.

Color

The Color class defines the same Color constants (Color.red, Color.yellow, etc.) as the AWT version but provides additional constructors to account for a transparency (alpha) value. A Color is represented by a 4-byte int value, where the three lowest bytes represent the red, green, and blue component, and the highest-order byte represents the alpha component. By default, colors are opaque with an alpha value of 255. A completely transparent color has an alpha value of 0. The common Color constructors are described below.

public Color(int red, int green, int blue)
public Color(float int, float green, float blue)

These two constructors create an opaque Color with the specified red, green, and blue components. The int values should range from 0 to 255, inclusive. The float values should range from 0.0f to 1.0f. Internally, each float value is converted to an int being multiplied by 255 and rounded up.

public Color(int red, int green, int blue, int alpha)
public Color(float red, float green, float blue, float alpha)

These two constructors create a Color object where the transparency value is specified by the alpha argument. See the preceding constructor for legal ranges for the red, green, blue, and alpha values.

Before drawing, you can also set the transparency (opaqueness) of a Shape by first creating an AlphaComposite object, then applying the AlphaComposite object to the Graphics2D context through the setComposite method. See Section 10.4 (Transparent Drawing) for details.

GradientPaint

A GradientPaint represents a smooth transition from one color to a second color. Two points establish a gradient line in the drawing, with one color located at one end point of the line and the second color located at other end point of the line. The color

will smoothly transition along the gradient line, with parallel color bands extending orthogonally to the gradient line. Depending on the value of a boolean flag, the color pattern will repeat along the *extended* gradient line until the end of the shape is reached.

public GradientPaint(float xStart, float yStart,
Color colorStart, float xEnd, float yEnd,
Color colorEnd)

This constructor creates a `GradientPaint`, beginning with a color of `colorStart` at (`xStart`, `yStart`) and finishing with a color of `colorEnd` at (`xEnd`, `yEnd`). The gradient is nonrepeating (a single gradient cycle).

public GradientPaint(float xStart, float yStart,
Color colorStart, float xEnd, float yEnd,
Color colorEend, boolean repeat)

This constructor is the same as the preceding constructor, except that a boolean flag, `repeat`, can be set to produce a pattern that continues to repeat beyond the end point (cyclic).

TexturePaint

A `TexturePaint` is simply an image that is tiled across the shape. When creating a textured paint, you need to specify both the image on the tile and the tile size.

public TexturePaint(BufferedImage image,
Rectangle2D tilesize)

The `TexturePaint` constructor maps a `BufferedImage` to a `Rectangle2D` and then tiles the rectangle. Creating a `BufferedImage` from a GIF or JPEG file is a pain. First, load an `Image` normally, get the size of the image, create a `BufferedImage` that sizes with `BufferedImage.TYPE_INT_ARGB` as the image type, get the `BufferedImage`'s `Graphics` object through `createGraphics`, then draw the `Image` into the `BufferedImage` using `drawImage`.

Listing 10.4 is an example of applying a gradient fill prior to drawing a circle. The gradient begins with a red color (`Color.red`) located at (0, 0) and gradually changes to a yellow color (`Color.yellow`) located at (185, 185) near the center of the circle. The gradient fill pattern repeats across the remaining area of the circle, as shown in Figure 10–2.

Listing 10.4 GradientPaintExample.java

```
import java.awt.*;

/** An example of applying a gradient fill to a circle. The
 *  color definition starts with red at (0,0), gradually
 *  changing to yellow at (175,175).
 */

public class GradientPaintExample extends ShapeExample {
  private GradientPaint gradient =
    new GradientPaint(0, 0, Color.red, 175, 175, Color.yellow,
                      true); // true means to repeat pattern

  public void paintComponent(Graphics g) {
    clear(g);
    Graphics2D g2d = (Graphics2D)g;
    drawGradientCircle(g2d);
  }

  protected void drawGradientCircle(Graphics2D g2d) {
    g2d.setPaint(gradient);
    g2d.fill(getCircle());
    g2d.setPaint(Color.black);
    g2d.draw(getCircle());
  }

  public static void main(String[] args) {
    WindowUtilities.openInJFrame(new GradientPaintExample(),
                                 380, 400);
  }
}
```

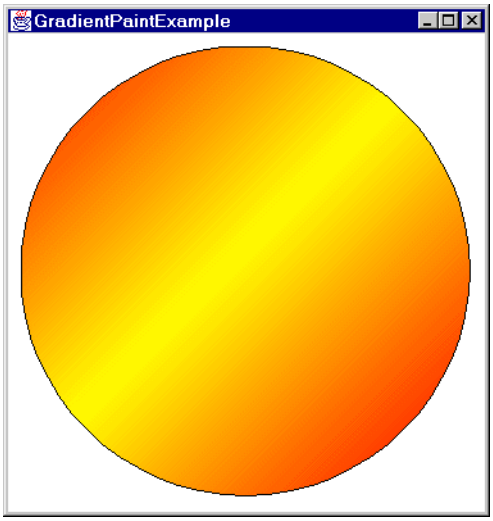

Figure 10–2 A circle drawn with a gradient fill in Java 2D.

Tiled Images as Fill Patterns

To use tiled images, you first create a `TexturePaint` object and pass the object to the `setPaint` method of `Graphics2D`, just as with solid colors and gradient fills. The `TexturePaint` constructor takes a `BufferedImage` and a `Rectangle2D` as arguments. The `BufferedImage` specifies what to draw, and the `Rectangle2D` specifies where the tiling starts. The rectangle also determines the size of the image that is drawn; the `BufferedImage` is scaled to the rectangle size before rendering. Creating a `BufferedImage` to hold a custom drawing is relatively straightforward: call the `BufferedImage` constructor with a width, a height, and a type of `BufferedImage.TYPE_INT_RGB`, then call `createGraphics` on the buffered image to get a `Graphics2D` with which to draw. For example,

```
int width =32;
int height=32;
BufferedImage bufferedImage =
  new BufferedImage(width, height
                    BufferedImage.TYPE_INT_RGB);
Graphics2D g2d = bufferedImage.createGraphics();
g2d.draw(someShape);
...
TexturePaint texture =
  new TexturePaint(bufferedImage,
                   new Rectangle(0, 0, width, height));
```

The `Graphics2D` object returned from `createGraphics` is bound to the `BufferedImage`. At that point, any drawing to the `Graphics2D` object is drawn to the `BufferedImage`. The texture "tile" in this example is a rectangle 32 pixels wide and 32 pixels high, where the drawing on the tile is what is contained in the buffered image.

Creating a `BufferedImage` from an image file is a bit harder. First, load an `Image` from an image file, then use `MediaTracker` to be sure that the image is loaded, then create an empty `BufferedImage` by using the `Image` width and height. Next, get the `Graphics2D` with `createGraphics`, then draw the `Image` onto the `BufferedImage`. This process has been wrapped up in the `getBufferedImage` method of the `ImageUtilities` class given in Listing 10.6.

An example of creating tiled images as fill patterns is shown in Listing 10.5. The result is presented in Figure 10–3. Two textures are created, one texture is an image of a blue drop, and the second texture is an image of Marty Hall contemplating another Java innovation while lounging in front of his vehicle. The first texture is applied before the large inverted triangle is drawn, and the second texture is applied before the centered rectangle is drawn. In the second case, the `Rectangle` is the same size as the `BufferedImage`, so the texture is tiled only once.

Listing 10.5 TiledImages.java

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

/** An example of using TexturePaint to fill objects with tiled
 *  images. Uses the getBufferedImage method of ImageUtilities
 *  to load an Image from a file and turn that into a
 *  BufferedImage.
 */

public class TiledImages extends JPanel {
  private String dir = System.getProperty("user.dir");
  private String imageFile1 = dir + "/images/marty.jpg";
  private TexturePaint imagePaint1;
  private Rectangle imageRect;
  private String imageFile2 = dir + "/images/bluedrop.gif";
  private TexturePaint imagePaint2;
  private int[] xPoints = { 30, 700, 400 };
  private int[] yPoints = { 30, 30, 600 };
  private Polygon imageTriangle = new Polygon(xPoints, yPoints, 3);
  public TiledImages() {
    BufferedImage image =
      ImageUtilities.getBufferedImage(imageFile1, this);
    imageRect = new Rectangle(235, 70, image.getWidth(),
                              image.getHeight());
    imagePaint1 = new TexturePaint(image, imageRect);
    image = ImageUtilities.getBufferedImage(imageFile2, this);
    imagePaint2 =
      new TexturePaint(image, new Rectangle(0, 0, 32, 32));
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    g2d.setPaint(imagePaint2);
    g2d.fill(imageTriangle);
    g2d.setPaint(Color.blue);
    g2d.setStroke(new BasicStroke(5));
    g2d.draw(imageTriangle);
    g2d.setPaint(imagePaint1);
    g2d.fill(imageRect);
    g2d.setPaint(Color.black);
    g2d.draw(imageRect);
  }

  public static void main(String[] args) {
    WindowUtilities.openInJFrame(new TiledImages(), 750, 650);
  }
}
```

Listing 10.6 ImageUtilities.java

```
import java.awt.*;
import java.awt.image.*;

/** A class that simplifies a few common image operations, in
 *  particular, creating a BufferedImage from an image file and
 *  using MediaTracker to wait until an image or several images
 *  are done loading.
 */

public class ImageUtilities {

  /** Create Image from a file, then turn that into a
   *  BufferedImage.
   */

  public static BufferedImage getBufferedImage(String imageFile,
                                               Component c) {
    Image image = c.getToolkit().getImage(imageFile);
    waitForImage(image, c);

    BufferedImage bufferedImage =
      new BufferedImage(image.getWidth(c), image.getHeight(c),
                        BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = bufferedImage.createGraphics();
    g2d.drawImage(image, 0, 0, c);
    return(bufferedImage);
  }

  /** Take an Image associated with a file, and wait until it is
   *  done loading (just a simple application of MediaTracker).
   *  If you are loading multiple images, don't use this
   *  consecutive times; instead, use the version that takes
   *  an array of images.
   */

  public static boolean waitForImage(Image image, Component c) {
    MediaTracker tracker = new MediaTracker(c);
    tracker.addImage(image, 0);
    try {
      tracker.waitForAll();
    } catch(InterruptedException ie) {}
    return(!tracker.isErrorAny());
  }
```

(continued)

Listing 10.6 `ImageUtilities.java` *(continued)*

```
  /** Take some Images associated with files, and wait until they
   *  are done loading (just a simple application of
   *  MediaTracker).
   */

  public static boolean waitForImages(Image[] images, Component c)
{
    MediaTracker tracker = new MediaTracker(c);
    for(int i=0; i<images.length; i++)
      tracker.addImage(images[i], 0);
    try {
      tracker.waitForAll();
    } catch(InterruptedException ie) {}
    return(!tracker.isErrorAny());
  }
}
```

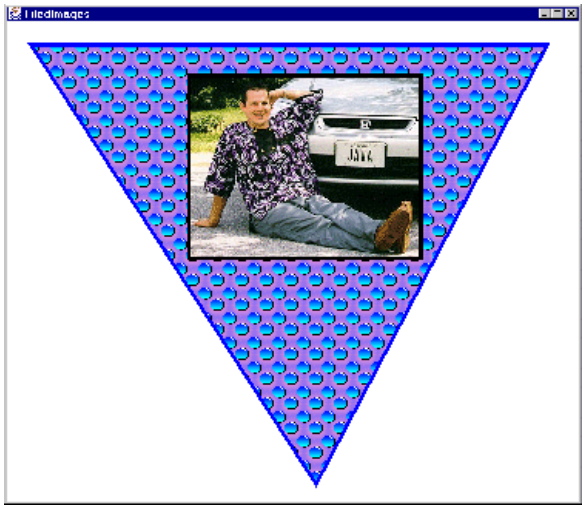

Figure 10–3 By creation of `TexturePaint` definition, images can be tiled across any shape.

10.4 Transparent Drawing

Java 2D permits you to assign transparency (alpha) values to drawing operations so that the underlying graphics partially shows through when you draw shapes or images. You set a transparency by creating an `AlphaComposite` object and then passing the `AlphaComposite` object to the `setComposite` method of the `Graphics2D` object. You create an `AlphaComposite` by calling `Alpha-`

`Composite.getInstance` with a mixing rule designator and a transparency (or "alpha") value. For example,

```
float alpha = 0.75f;
int type = AlphaComposite.SRC_OVER;
AlphaComposite composite =
   AlphaComposite.getInstance(type, alpha);
```

The `AlphaComposite` API provides eight built-in mixing rules, but the one normally used for drawing with transparency settings is `AlphaComposite.SRC_OVER`, a *source* over *destination* mixing rule that places the source (shape) over the destination (background). A complete definition of the mixing rule was provided by T. Porter and T. Duff in "Compositing Digital Images," *SIGGRAPH* **84**, pp. 253–259. Alpha values can range from 0.0f (completely transparent) to 1.0f (completely opaque).

Listing 10.7 demonstrates changing the transparency setting before drawing a red square that is partially overlapping a blue square. As shown in Figure 10–4, 11 opaque blue squares are drawn, equally spaced across the panel. Partially overlapping is a red square drawn with an initial alpha value of 0.0f at the far left. The red square is repeatedly drawn at new locations across the panel with alpha values that gradually increase by a step size of 0.1f until, finally, total opaqueness is reached at the far right with an alpha value of 1.0f.

Recall from Section 10.3 (Paint Styles) that the transparency (alpha) value of a color can be changed directly. Thus, for this example, the transparency of the red box could be directly set by a new color, as in:

```
private void drawSquares(Graphics2D g2d, float alpha) {
    g2d.setPaint(Color.blue);
    g2d.fill(blueSquare);
    Color color = new Color(1, 0, 0, alpha); //Red
    g2d.setPaint(color);
    g2d.fill(redSquare);
}
```

Here, the assumption is that the original compositing rule is `AlphaComposite.SRC_OVER`, which is the default for the `Graphics2D` object. If the alpha value is set both through an `AlphaComposite` object and a `Color` object, the alpha values are multiplied to obtain the final transparency value.

Listing 10.7 `TransparencyExample.java`

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

/** An illustration of the use of AlphaComposite to make
 *  partially transparent drawings.
 */
```

(continued)

Listing 10.7 `TransparencyExample.java` *(continued)*

```
public class TransparencyExample extends JPanel {
  private static int gap=10, width=60, offset=20,
                     deltaX=gap+width+offset;
  private Rectangle
    blueSquare = new Rectangle(gap+offset, gap+offset, width,
                               width),
    redSquare = new Rectangle(gap, gap, width, width);

  private AlphaComposite makeComposite(float alpha) {
    int type = AlphaComposite.SRC_OVER;
    return(AlphaComposite.getInstance(type, alpha));
  }

  private void drawSquares(Graphics2D g2d, float alpha) {
    Composite originalComposite = g2d.getComposite();
    g2d.setPaint(Color.blue);
    g2d.fill(blueSquare);
    g2d.setComposite(makeComposite(alpha));
    g2d.setPaint(Color.red);
    g2d.fill(redSquare);
    g2d.setComposite(originalComposite);
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    for(int i=0; i<11; i++) {
      drawSquares(g2d, i*0.1F);
      g2d.translate(deltaX, 0);
    }
  }

  public static void main(String[] args) {
    String title = "Transparency example: alpha of the top " +
                   "(red) square ranges from 0.0 at the left " +
                   "to 1.0 at the right. Bottom (blue) square " +
                   "is opaque.";
    WindowUtilities.openInJFrame(new TransparencyExample(),
                                 11*deltaX + 2*gap,
                                 deltaX + 3*gap,
                                 title, Color.lightGray);
  }
}
```

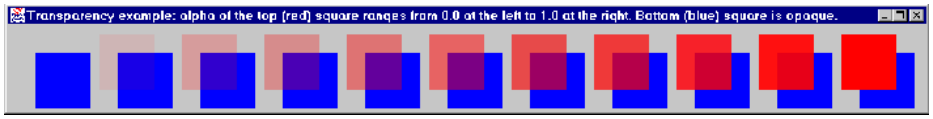

Figure 10–4 Changing the transparency setting can allow the background image behind the shape to show through.

10.5 Using Local Fonts

In Java 2D you can use the same logical font names as in Java 1.1, namely, Serif (e.g., Times), SansSerif (e.g., Helvetica or Arial), Monospaced (e.g., Courier), Dialog, and DialogInput. However, you can also use arbitrary local fonts installed on the platform if you first look up the entire list, which may take a few seconds. Look up the fonts with the `getAvailableFontFamilyNames` or `getAllFonts` methods of `GraphicsEnvironment`. For example:

```
GraphicsEnvironment env =
  GrapicsEnvironment.getLocalGraphicsEnvironment();
```

Then, add

```
env.getAvailableFontFamilyNames();
```

or

```
env.getAllFonts();  // Much slower!
```

Despite a misleading description in the API, trying to use an available local font without first looking up the fonts, as above, gives the same result as asking for an unavailable font: a default font instead of the actual one.

Core Warning

Trying to use a local font without first looking up the fonts results in a default font being used instead of the actual one. You only need to incur this overhead the first time that you need to create a new `Font` object.

Note that `getAllFonts` returns an array of real `Font` objects that you can use like any other `Font` but is much slower. If all you need to do is tell Java to make all local fonts available, always use `getAvailableFontFamilyNames`.

The best approach is to loop down `getAvailableFontFamilyNames`, checking for the preferred font name and having several backup names to use if the first choice is not available. If you pass an unavailable family name to the `Font` constructor, a default font (SansSerif) is used. Listing 10.8 provides the basic code for listing all available fonts on the platform.

Listing 10.8 ListFonts.java

```
import java.awt.*;

/** Lists the names of all available fonts. */

public class ListFonts {
  public static void main(String[] args) {
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    String[] fontNames = env.getAvailableFontFamilyNames();
    System.out.println("Available Fonts:");
    for(int i=0; i<fontNames.length; i++)
      System.out.println("  " + fontNames[i]);
  }
}
```

Listing 10.9 gives a simple example of first looking up the available fonts on the system and then setting the style to Goudy Handtooled BT prior to drawing the String "Java 2D". The result is shown in Figure 10–5. On platforms without Goudy Handtooled BT, the text is drawn in the default font, SansSerif.

Listing 10.9 FontExample.java

```
import java.awt.*;

/** An example of using local fonts to perform drawing in
 *  Java 2D.
 */

public class FontExample extends GradientPaintExample {
  public FontExample() {
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    env.getAvailableFontFamilyNames();
    setFont(new Font("Goudy Handtooled BT", Font.PLAIN, 100));
  }

  protected void drawBigString(Graphics2D g2d) {
    g2d.setPaint(Color.black);
    g2d.drawString("Java 2D", 25, 215);
  }

  public void paintComponent(Graphics g) {
    clear(g);
```

(continued)

Listing 10.9 `FontExample.java` *(continued)*

```
    Graphics2D g2d = (Graphics2D)g;
    drawGradientCircle(g2d);
    drawBigString(g2d);
  }

  public static void main(String[] args) {
    WindowUtilities.openInJFrame(new FontExample(), 380, 400);
  }
}
```

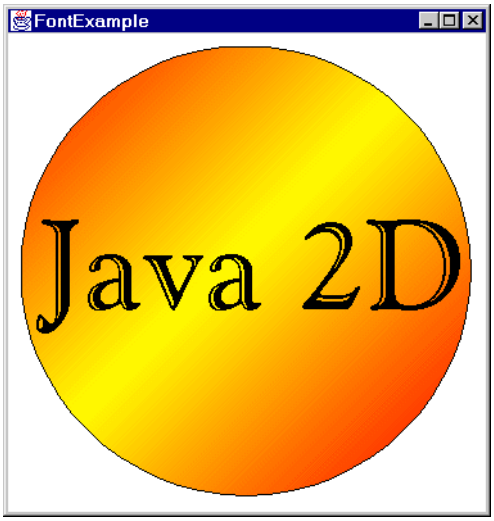

Figure 10–5 In Java 2D, writing text in any local font installed on the platform is possible.

10.6 Stroke Styles

In the AWT, the `drawXxx` methods of `Graphics` resulted in solid, 1-pixel-wide lines. Furthermore, drawing commands that consisted of multiple-line segments (e.g., `drawRect` and `drawPolygon`) had a predefined way of joining the line segments and terminating segments that did not join to others. Java 2D gives you much more flexibility. In addition to setting the pen color or pattern (through `setPaint`, as discussed in the previous section), with Java 2D you can set the pen thickness and dashing pattern and specify the way in which line segments end and are joined together. To control how lines are drawn, first create a `BasicStroke` object, then use the `setStroke` method to tell the `Graphics2D` object to use the `BasicStroke` object.

Stroke Attributes

Arguments to `setStroke` must implement the `Stroke` interface, and the `BasicStroke` class is the sole built-in class that implements `Stroke`. Here are the `BasicStroke` constructors.

public BasicStroke()
This constructor creates a `BasicStroke` with a pen width of 1.0, the default cap style of `CAP_SQUARE`, and the default join style of `JOIN_MITER`. See the following examples of pen widths and cap/join styles.

public BasicStroke(float penWidth)
This constructor creates a `BasicStroke` with the specified pen width and the default cap/join styles (`CAP_SQUARE` and `JOIN_MITER`).

public BasicStroke(float penWidth, int capStyle, int joinStyle)
This constructor creates a `BasicStroke` with the specified pen width, cap style, and join style. The cap style can be one of `CAP_SQUARE` (make a square cap that extends past the end point by half the pen width—the default), `CAP_BUTT` (cut off segment exactly at end point—use this one for dashed lines), or `CAP_ROUND` (make a circular cap centered on the end point, with a diameter of the pen width). The join style can be one of `JOIN_MITER` (extend outside edges of lines until they meet—the default), `JOIN_BEVEL` (connect outside corners of outlines with straight line), or `JOIN_ROUND` (round off corner with circle with diameter equal to the pen width).

public BasicStroke(float penWidth, int capStyle, int joinStyle, float miterLimit)
This constructor is the same as above, but you can limit how far up the line the miter join can proceed (default is 10.0). A `miterLimit` of 10.0 is a reasonable default, so you rarely need this constructor.

public BasicStroke(float penWidth, int capStyle, int joinStyle, float miterLimit, float[] dashPattern, float dashOffset)
This constructor lets you make dashed lines by specifying an array of opaque (entries at even array indices) and transparent (odd indices) segments. The offset, which is often 0.0, specifies where to start in the dashing pattern.

Two examples of controlling the pen attribute follow. The first example, Listing 10.10, sets the pen width to 8 pixels before drawing an outlined circle, and the second example, Listing 10.11, creates a dashed line. The dash pattern is

```
float[] dashPattern { 30, 10, 10, 10 };
```

where the values alternate between the dash length and the gap length. The result is an opaque dash for 30 units, a transparent dash for 10 units, another opaque dash for 10 units, and, finally, a transparent dash for the last 10 units. The pattern is repeated along the line segment. The results for the two examples are shown in Figure 10–6 and Figure 10–7, respectively.

Listing 10.10 StrokeThicknessExample.java

```
import java.awt.*;

/** An example of controlling the Stroke (pen) widths when
 *  drawing.
 */

public class StrokeThicknessExample extends FontExample {
  public void paintComponent(Graphics g) {
    clear(g);
    Graphics2D g2d = (Graphics2D)g;
    drawGradientCircle(g2d);
    drawBigString(g2d);
    drawThickCircleOutline(g2d);
  }

  protected void drawThickCircleOutline(Graphics2D g2d) {
    g2d.setPaint(Color.blue);
    g2d.setStroke(new BasicStroke(8)); // 8-pixel wide pen
    g2d.draw(getCircle());
  }

  public static void main(String[] args) {
    WindowUtilities.openInJFrame(new StrokeThicknessExample(),
                                 380, 400);
  }
}
```

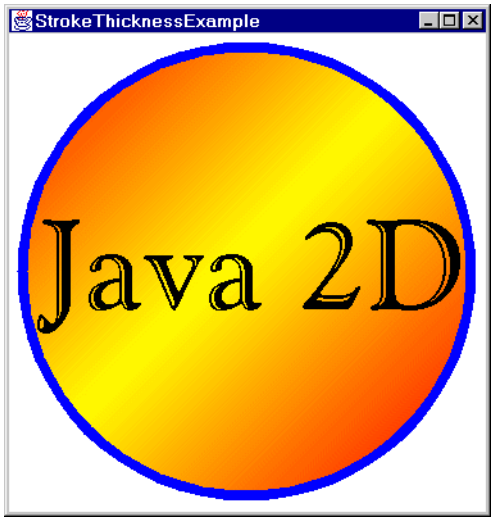

Figure 10–6 The outline of a circle drawn with a pen width of 8 pixels.

Listing 10.11 `DashedStrokeExample.java`

```
import java.awt.*;

/** An example of creating a custom dashed line for drawing.
 */

public class DashedStrokeExample extends FontExample {
  public void paintComponent(Graphics g) {
    clear(g);
    Graphics2D g2d = (Graphics2D)g;
    drawGradientCircle(g2d);
    drawBigString(g2d);
    drawDashedCircleOutline(g2d);
  }

  protected void drawDashedCircleOutline(Graphics2D g2d) {
    g2d.setPaint(Color.blue);
    // 30-pixel line, 10-pixel gap, 10-pixel line, 10-pixel gap
    float[] dashPattern = { 30, 10, 10, 10 };
    g2d.setStroke(new BasicStroke(8, BasicStroke.CAP_BUTT,
                                  BasicStroke.JOIN_MITER, 10,
                                  dashPattern, 0));
    g2d.draw(getCircle());
  }

  public static void main(String[] args) {
    WindowUtilities.openInJFrame(new DashedStrokeExample(),
                                 380, 400);
  }
}
```

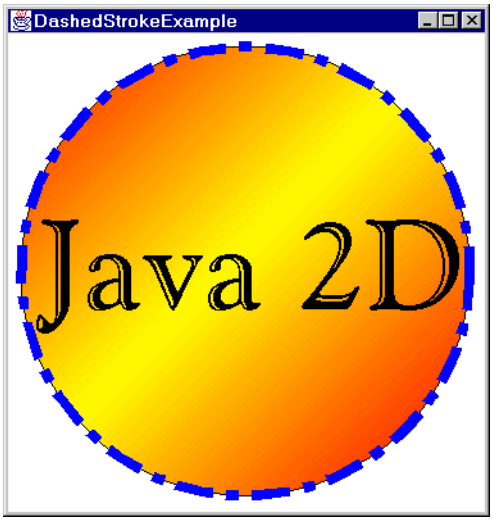

Figure 10–7 The outline of a circle drawn with a dashed line segment.

As a final example of pen styles, Listing 10.12 demonstrates the effect of different styles for *joining* line segments, and different styles for creating line end points (*cap* settings). Figure 10–8 clearly illustrates the differences of the three joining styles (`JOIN_MITER`, `JOIN_BEVEL`, and `JOIN_ROUND`), as well as the differences of the three cap styles (`CAP_SQUARE`, `CAP_BUTT`, and `CAP_ROUND`).

Listing 10.12 `LineStyles.java`

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

/** A demonstration of different controls when joining two line
 *  segments. The style of the line end point is controlled
 *  through the capStyle parameter.
 */

public class LineStyles extends JPanel {
  private GeneralPath path;
  private static int x = 30, deltaX = 150, y = 300,
                     deltaY = 250, thickness = 40;
  private Circle p1Large, p1Small, p2Large, p2Small,
                 p3Large, p3Small;
  private int compositeType = AlphaComposite.SRC_OVER;
```

(continued)

Listing 10.12 LineStyles.java *(continued)*

```
private AlphaComposite transparentComposite =
  AlphaComposite.getInstance(compositeType, 0.4F);
private int[] caps =
  { BasicStroke.CAP_SQUARE, BasicStroke.CAP_BUTT,
    BasicStroke.CAP_ROUND };
private String[] capNames =
  { "CAP_SQUARE", "CAP_BUTT", "CAP_ROUND" };
private int[] joins =
  { BasicStroke.JOIN_MITER, BasicStroke.JOIN_BEVEL,
    BasicStroke.JOIN_ROUND };
private String[] joinNames =
  { "JOIN_MITER", "JOIN_BEVEL", "JOIN_ROUND" };

public LineStyles() {
  path = new GeneralPath();
  path.moveTo(x, y);
  p1Large = new Circle(x, y, thickness/2);
  p1Small = new Circle(x, y, 2);
  path.lineTo(x + deltaX, y - deltaY);
  p2Large = new Circle(x + deltaX, y - deltaY, thickness/2);
  p2Small = new Circle(x + deltaX, y - deltaY, 2);
  path.lineTo(x + 2*deltaX, y);
  p3Large = new Circle(x + 2*deltaX, y, thickness/2);
  p3Small = new Circle(x + 2*deltaX, y, 2);
  setFont(new Font("SansSerif", Font.BOLD, 20));
}

public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setColor(Color.lightGray);
  for(int i=0; i<caps.length; i++) {
    BasicStroke stroke =
      new BasicStroke(thickness, caps[i], joins[i]);
    g2d.setStroke(stroke);
    g2d.draw(path);
    labelEndPoints(g2d, capNames[i], joinNames[i]);
    g2d.translate(3*x + 2*deltaX, 0);
  }
}

// Draw translucent circles to illustrate actual end points.
// Include text labels for cap/join style.
private void labelEndPoints(Graphics2D g2d, String capLabel,
                            String joinLabel) {
```

(continued)

Listing 10.12 `LineStyles.java` *(continued)*

```
    Paint origPaint = g2d.getPaint();
    Composite origComposite = g2d.getComposite();
    g2d.setPaint(Color.black);
    g2d.setComposite(transparentComposite);
    g2d.fill(p1Large);
    g2d.fill(p2Large);
    g2d.fill(p3Large);
    g2d.setPaint(Color.yellow);
    g2d.setComposite(origComposite);
    g2d.fill(p1Small);
    g2d.fill(p2Small);
    g2d.fill(p3Small);
    g2d.setPaint(Color.black);
    g2d.drawString(capLabel, x + thickness - 5, y + 5);
    g2d.drawString(joinLabel, x + deltaX + thickness - 5,
                   y - deltaY);
    g2d.setPaint(origPaint);
  }

  public static void main(String[] args) {
    WindowUtilities.openInJFrame(new LineStyles(),
                                 9*x + 6*deltaX, y + 60);
  }
}

class Circle extends Ellipse2D.Double {
  public Circle(double centerX, double centerY, double radius) {
    super(centerX - radius, centerY - radius, 2.0*radius,
          2.0*radius);
  }
}
```

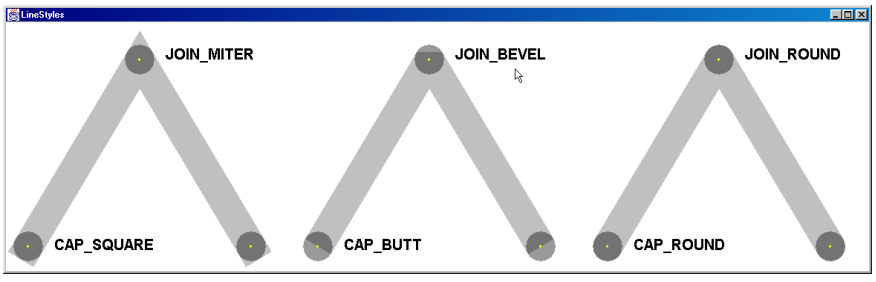

Figure 10–8 A demonstration of the different styles for joining line segments, and styles for ending a line segment.

10.7 Coordinate Transformations

Java 2D allows you to easily translate, rotate, scale, or shear the coordinate system. This capability is very convenient: moving the coordinate system is often much easier than calculating new coordinates for each of your points. Besides, for some data structures like ellipses and strings, the only way to create a rotated or stretched version is through a transformation. The meanings of translate, rotate, and scale are clear: to move, to spin, or to stretch/shrink evenly in the x and/or y direction. Shear means to stretch unevenly: an x shear moves points to the right, based on how far they are from the y-axis; a y shear moves points down, based on how far they are from the x-axis.

The easiest way to picture what is happening in a transformation is to imagine that the person doing the drawing has a picture frame that he lays down on top of a sheet of paper. The drawer always sits at the bottom of the frame. To apply a translation, you move the frame (also moving the drawer) and do the drawing in the new location. You then move the frame back to its original location, and what you now see is the final result. Similarly, for a rotation, you spin the frame (and the drawer), draw, then spin back to see the result. Similarly for scaling and shears: modify the frame without touching the underlying sheet of paper, draw, then reverse the process to see the final result.

An outside observer watching this process would see the frame move in the direction specified by the transformation but see the sheet of paper stay fixed. On the other hand, to the person doing the drawing, it would appear that the sheet of paper moved in the opposite way from that specified in the transformation but that he didn't move at all.

You can also perform complex transformations by directly manipulating the underlying arrays that control the transformations. This type of manipulation is a bit more complicated to envision than the basic translation, rotation, scaling, and shear transformations. The idea is that a new point (x_2, y_2) can be derived from an original point (x_1, y_1) as follows:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00}x_1 + m_{01}y_1 + m_{02} \\ m_{10}x_1 + m_{11}y_1 + m_{12} \\ 1 \end{bmatrix}$$

Note that you can only supply six of the nine values in the transformation array (the m_{xx} values). The coefficients m_{02} and m_{12} provide x and y *translation* of the coordinate system. The other four transformation coefficients (m_{00}, m_{01}, m_{10}, m_{11}) provide *rotation* of the system. For the transformation to preserve orthogonality ("straightness" and "parallelness" of lines), the Jacobian (determinant) of the transformation matrix must equal 1. The bottom row is fixed at [0 0 1] to guarantee that

the transformations does not rotate the shape out of the *x*-*y* plane (produce components along the *z*-axis). There are several ways to supply this array to the `AffineTransform` constructor; see the `AffineTransform` API for details.

You use transformations in two basic ways—by creating an `AffineTransform` object or by calling basic transformation methods. In the first approach, you can create an `AffineTransform` object, set the parameters for the object, assign the `AffineTransform` to the `Graphics2D` object through `setTransform`, and then draw a `Shape`. In addition, you can use the `AffineTransform` object on a `Shape` to create a newly transformed `Shape` object. Simply call the `AffineTransform` method, `createTransformedShape`, to create a new transformed `Shape`. For complex transformations, creating an `AffineTransform` object is an excellent approach because you can explicitly define the transformation matrix.

Core Note

You can apply a transformation to a `Shape` before drawing it. The `AffineTransform` method `createTransformedShape` creates a new `Shape` that has undergone the transformation defined by the `AffineTransform` object.

In the second approach, you can call `translate`, `rotate`, `scale`, and `shear` directly on the `Graphics2D` object to perform basic transformations. The transformations applied to `Graphics2D` object are cumulative; each transform method is applied to the already transformed `Graphics2D` context. For example, calling `rotate(Math.PI/2)` followed by another call to `rotate(Math.PI/2)` is equivalent to `rotate(Math.PI)`. If you need to return to a previously existing transformation state, save the `Graphics2D` context by calling `getTransform` beforehand, perform your transformation operations, and then return to the original `Graphics2D` context by calling `setTransform`. For example,

```
// Save current graphics context.
AffineTransform transform = g2d.getTransform();
// Perform incremental transformations.
translate(...);
rotate(...);
...
// Return the graphics context to the original state.
g2d.setTransform(transform);
```

Listing 10.13 illustrates a beautiful example of continuously rotating the coordinate system while periodically writing the word "Java." The result is shown in Figure 10–9.

Listing 10.13 `RotationExample.java`

```
import java.awt.*;

/** An example of translating and rotating the coordinate
 *  system before each drawing.
 */

public class RotationExample extends StrokeThicknessExample {
  private Color[] colors = { Color.white, Color.black };

  public void paintComponent(Graphics g) {
    clear(g);
    Graphics2D g2d = (Graphics2D)g;
    drawGradientCircle(g2d);
    drawThickCircleOutline(g2d);
    // Move the origin to the center of the circle.
    g2d.translate(185.0, 185.0);
    for (int i=0; i<16; i++) {
      // Rotate the coordinate system around current
      // origin, which is at the center of the circle.
      g2d.rotate(Math.PI/8.0);
      g2d.setPaint(colors[i%2]);
      g2d.drawString("Java", 0, 0);
    }
  }

public static void main(String[] args) {
    WindowUtilities.openInJFrame(new RotationExample(), 380, 400);
  }
}
```

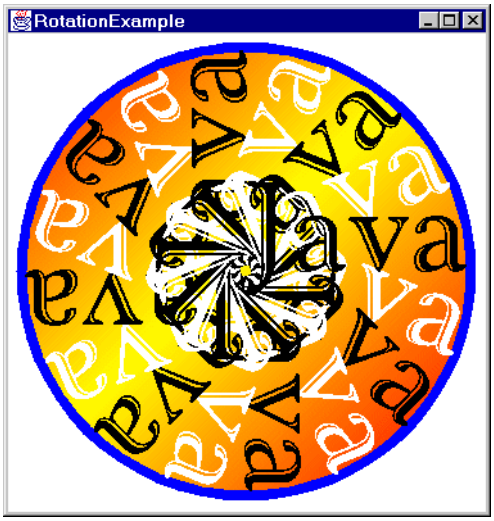

Figure 10–9 A example of translating and rotating the coordinate system before drawing text.

Shear Transformations

In a shear transformation, the coordinate system is stretched parallel to one axis. If you specify a nonzero *x* shear, then *x* values will be more and more shifted to the right the farther they are from the ***y***-axis. For example, an *x* shear of 0.1 means that the *x* value will be shifted 10% of the distance the point is moved from the *y*-axis. A *y* shear is similar: points are shifted down in proportion to the distance they are from the ***x***-axis. In addition, both the *x*- and *y*-axis can be sheared at the same time.

Probably the best way to visualize shear is in an example. The results for Listing 10.14 are shown in Figure 10–10. Here, the *x* shear is increased from a value of 0.0 for the first square to a value of +0.8 for the fifth square. The *y* values remain unaltered.

Listing 10.14 `ShearExample.java`

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

/** An example of shear transformations on a rectangle. */

public class ShearExample extends JPanel {
  private static int gap=10, width=100;
  private Rectangle rect = new Rectangle(gap, gap, 100, 100);

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    for (int i=0; i<5; i++) {
      g2d.setPaint(Color.red);
      g2d.fill(rect);
      // Each new square gets 0.2 more x shear.
      g2d.shear(0.2, 0.0);
      g2d.translate(2*gap + width, 0);
    }
  }

  public static void main(String[] args) {
    String title =
      "Shear: x shear ranges from 0.0 for the leftmost" +
      "'square' to 0.8 for the rightmost one.";
    WindowUtilities.openInJFrame(new ShearExample(),
                                 20*gap + 5*width,
                                 5*gap + width,
                                 title);
  }
}
```

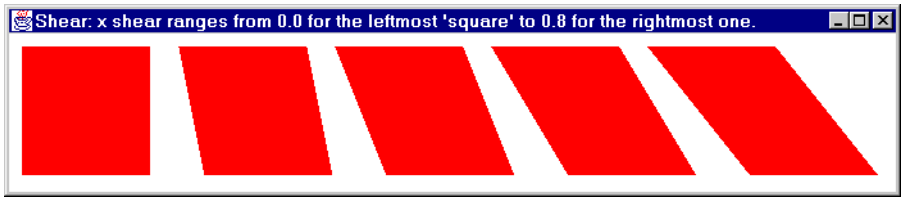

Figure 10–10 A positive *x* shear increases the shift in the *x* coordinate axis as *y* increases. Remember, the positive *y*-axis goes from the upper-left corner to the lower-left corner.

10.8 Other Capabilities of Java 2D

Since Java 2D already does a lot of calculations compared to the old AWT, by default, many of the optional features to improve performance are turned off. However, for crisper drawings, especially for rotated text, the optional features should be turned on.

The two most important settings are to turn on antialiasing (smooth jagged lines by blending colors) and to request the highest-quality graphics rendering. This approach is illustrated below:

```
RenderingHints renderHints =
  new RenderingHints(RenderingHints.KEY_ANTIALIASING,
                     RenderingHints.VALUE_ANTIALIAS_ON);
renderHints.put(RenderingHints.KEY_RENDERING,
                RenderingHints.VALUE_RENDER_QUALITY);
...

public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHints(renderHints);
  ...
}
```

Core Approach

For the highest-quality graphics, use `RenderingHints` *and turn antialiasing on,* `VALUE_ANTIALIAS_ON`*, and set the presentation for quality, not speed, with* `VALUE_RENDER_QUALITY`*.*

Thus far, we've only presented a small fraction of the power behind Java 2D. Once you've gained experience with the basics, you'll want to tackle advanced Java 2D techniques that allow you to:

- Create custom color mixing (implement `Composite` and `CompositeContext` interfaces).
- Perform bounds/hit testing (see `contains` and `intersects` methods of `Shape`).
- Create new fonts by transforming old ones (use `Font.deriveFont`).
- Draw multifont or multicolor strings (use the `draw` method of `TextLayout`).
- Draw outlines of fonts, or fill fonts with images or gradient colors (use the `getOutline` method of `TextLayout`).
- Perform low-level image processing and color model manipulation.
- Produce high-quality printing for Swing components. See Section 15.5 (Swing Component Printing) for details.

10.9 Summary

Java 2D enables the artistic imagination of any programmer to produce high-quality, professional graphics. Java 2D opens the door to numerous possibilities; you can

- Draw or fill any `Shape`. Simply call the `Graphics2D`'s `draw` or `fill` methods with the shape as an argument.
- Take advantage of the `setPaint` method in the `Graphics2D` class to paint shapes in solid colors (`Color`), gradient fills (`GradientPaint`), or with tiled images (`TexturePaint`).
- Explore transparent shapes and change mixing rules for joining shapes. Numerous Porter-Duff mixing rules in the `AlhpaComposite` class define how shapes are combined with the background.
- Break the "one pixel wide" pen boundary and create a `BasicStroke` to control the width of the pen, create dashing patterns, and define how line segments are joined.
- Create an `AffineTransform` object and call `setTransform` on the `Graphics2D` object to translate, rotate, scale, and shear those shapes before drawing.
- Control the quality of image through the `RenderingHints`. In addition, the `RenderingHints` can control antialiasing of colors at shape boundaries for a smoother, more appealing presentation.

Remember that Java 2D is a part of the Java Foundation Classes and only available with the Java 2 platform. Swing, a robust set of lightweight components, is also a fundamental component of the Java Foundation Classes. Swing is covered in Chapter 14 (Basic Swing) and Chapter 15 (Advanced Swing).

Drawing fancy shapes, text, and images is nice, but for a complete user interface, you need to be able to react to actions taken by the user, create other types of windows, and insert user interface controls such as buttons, textfields, and the like. These topics are discussed in the next three chapters.