Java 2D Graphics, The Stroke Interface

by Richard G. Baldwin
baldwin@austin.cc.tx.us

Java Programming, Lecture Notes # 318

March 19, 2000

Introduction

In an earlier lesson, I explained that the **Graphics2D** class extends the **Graphics** class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. Beginning with JDK 1.2, **Graphics2D** is the fundamental class for rendering two-dimensional shapes, text and images.

Understanding other classes is also required

I also explained that without understanding the behavior of other classes and interfaces such as **Shape**, **AffineTransform**, **GraphicsConfiguration**, **PathIterator**, and **Stroke**, it is not possible to fully understand the inner workings of the **Graphics2D** class.

Setter methods

I explained in an earlier lesson that the manner in which **Graphics2D** renders shapes, text, and images depends on the current values of several properties of the **Graphics2D** object. The values of these properties are controlled using standard *setter* methods of the class, such as the following.

- **setComposite()**

- **setPaint()**
- **setRenderingHint()**
- **setStroke()**
- **setTransform()**

Previous lessons have explained how to use **setPaint()** and **setTransform()**.

This lesson will show you how to use **setStroke()** to control how a **Graphics2D** object renders *strokes*.

What is a Stroke?

Here is what Sun has to say about the **Stroke** interface.

"The **Stroke** interface allows a **Graphics2D** object to obtain a **Shape** that is the decorated outline, or stylistic representation of the outline, of the specified **Shape**.

Stroking a **Shape** is like tracing its outline with a marking pen of the appropriate size and shape. The area where the pen would place ink is the area enclosed by the outline **Shape**.

The methods of the **Graphics2D** interface that use the outline **Shape** returned by a **Stroke** object include draw and any other methods that are implemented in terms of that method, such as drawLine, drawRect, drawRoundRect, drawOval, drawArc, drawPolyline, and drawPolygon."

A Shape describes a Shape

What this says to me is that the **Stroke** interface makes it possible to consider the outline of a **Shape** object to itself be considered as a **Shape** object. Once the outline is represented as a **Shape**, anything that can be done to a **Shape** object can be done to the **Shape** that represents the outline.

Like tracing with a marking pen

The geometry of the **Shape** that represents the outline is similar to what you would produce by tracing the original **Shape** with a marking pen. Different marking pens would yield different colors and pen widths.

Could also produce dot-dash patterns

In addition, if you had sufficient artistic talent,

- You could lift and lower the pen to produce various dot-dash patterns.
- You could provide some specific treatment for the ends of the lines.
- You could provide some specific treatment for the vertices where line segments produce an angle.

Of course, you could do other things as well.

Stroke interface method

The Stroke interface declares a single method named **createStrokedShape()**. Here is what Sun has to say about this method.

Returns an outline **Shape**, which encloses the area that should be painted when the **Shape** is stroked according to the rules defined by the object implementing the **Stroke** interface.

Parameters:
p - a **Shape** to be stroked

Returns:
the stroked outline **Shape**.

BasicStroke implements the Stroke interface

As of JDK 1.2.2, there is only one class in the API that implements the **Stroke** interface. The name of the class is **BasicStroke**.

Of course, if you need to do so, you can define your own class that implements the **Stroke** interface.

What is a BasicStroke?

Here is what Sun has to say about the **BasicStroke** class.

“The **BasicStroke** class defines a basic set of rendering attributes for the outlines of graphics primitives. These attributes describe the shape of the mark made by a pen drawn along the outline of a **Shape** object and the decorations applied at the ends and joins of path segments of the **Shape** object. These attributes include:

- **width:** The pen width, measured perpendicularly to the pen trajectory.
- **end caps:** The decoration applied to the ends of unclosed subpaths or dash segments.
- **line joins:** The decoration applied where two path segments are joined.
- **dash attributes:** The definition of how to make a dash pattern by alternating between opaque and transparent sections.”

Matches what I said earlier

You will recognize the attributes described here as matching the things that I said you could accomplish with the marking pen in an earlier paragraph.

The setStroke() Method

Here is what Sun has to say about the **setStroke()** method.

Sets the **Stroke** for the **Graphics2D** context.

Parameters:

s - the **Stroke** object to be used to stroke a **Shape** during the rendering process

The bottom line

Here is the bottom line regarding *stroke*. If you have a **Shape** object inside an overridden **paint()** method, and you would like to render the outline of that object onto an output device, you can do so by

- Invoking **setStroke()** on your **Graphics2D** object, passing a reference to an object of a class that implements the **Stroke** interface as a parameter.
- Invoking the **draw()** method (or some other method that can be used to render the **Shape** object) passing the **Shape** object as a parameter.

You can define your own class

You can define your own class that implements the **Stroke** interface. If you do so, you must implement the method named **createStrokedShape()**, returning a **Shape** object that represents the outline of the **Shape** object that you are rendering.

Or, you can use the BasicStroke class

Or, if the capabilities of the **BasicStroke** class will serve your needs, you don't need to define your own class to implement the **Stroke** interface. You can simply instantiate an object of the **BasicStroke** class and pass a reference to that object to the **setStroke()** method.

Rendering attributes

The **BasicStroke** class has several overloaded constructors. By passing specific values as parameters to the constructor, you control several attributes of the rendering process including:

- **width:** The pen width, measured perpendicularly to the pen trajectory.
- **end caps:** The decoration applied to the ends of unclosed subpaths or dash segments.
- **line joins:** The decoration applied where two path segments are joined.
- **dash attributes:** The definition of how to make a dash pattern by alternating between opaque and transparent sections.

The BasicStroke constructor

Before getting into the details of the sample program, let's take a look at the most complex of the **BasicStroke** constructors. Here is what Sun has to say about that constructor.

```
public BasicStroke(
  float width,
  int cap,
  int join,
  float miterlimit,
  float[] dash,
  float dash_phase)
```

Constructs a new **BasicStroke** with the specified attributes.

Parameters:

- **width** - the width of the BasicStroke
- **cap** - the decoration of the ends of a BasicStroke
- **join** - the decoration applied where path segments meet
- **miterlimit** - the limit to trim the miter join
- **dash** - the array representing the dashing pattern
- **dash_phase** - the offset to start the dashing pattern

With the exception of **miterlimit**, these are fairly self-explanatory. Later, I will tell you what another author has to say about **miterlimit**.

Sample Program

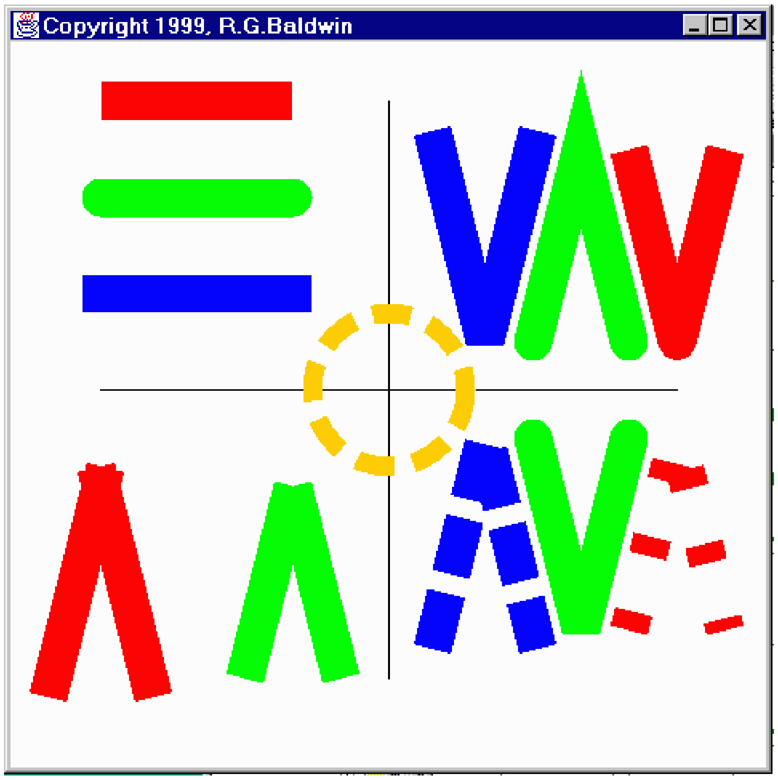
This program, named **Stroke01**, illustrates all of the attributes controlled by the constructor parameters listed above except for **dash_phase**. I will explain that attribute later.

Compile and run the program

You should compile and run this program now so that you can see the screen output while reading the remaining material in this lesson. If you can't see the screen output, the text probably won't mean a lot to you.

You can copy the program from the end of this lesson into a Java source file, compile it using JDK 1.2 or later, and then execute it. That should produce a **Frame** object on your screen.

In case you can't do that, here is a screen shot of the output. However, this image has been reduced considerably, so much of the detail, including the X and Y axes, is missing.



<span style="color:red">**The GUI is a Frame object**</span>

The program draws a four-inch by four-inch **Frame** on the screen. It translates the origin to the center of the **Frame**. Then it draws a pair of X and Y-axes centered on the new origin.

This discussion of dimensions in inches on the screen depends on the method named **getScreenResolution()** returning the correct value. However, the **getScreenResolution()** method always seems to return 120 on my computer regardless of the actual screen resolution settings.

Will discuss in fragments

I will discuss this program in fragments. The controlling class and the constructor for the GUI class are essentially the same as you have seen in several previous lessons, so, I won't repeat that discussion here. You can view that material in the complete listing of the program at the end of the lesson.

All of the interesting action takes place in the overridden **paint()** method, so I will begin the discussion there.

Overridden paint() method

The beginning portions of the overridden **paint()** method should be familiar to you by now as well. So, I am going to let the comments in Figure 1 speak for themselves.

```
publicvoid paint(Graphics g){
  //Downcast the Graphics object to a
  // Graphics2D object
  Graphics2D g2 = (Graphics2D)g;

  //Scale device space to produce inches
  // on the screen based on actual screen
  // resolution.
  g2.scale((double)res/72,(double)res/72);

  //Translate origin to center of Frame
  g2.translate((hSize/2)*ds,(vSize/2)*ds);

  //Draw x-axis
  g2.draw(new Line2D.Double(
                           -1.5*ds,0.0,1.5*ds,0.0));
  //Draw y-axis
  g2.draw(new Line2D.Double(
                           0.0,-1.5*ds,0.0,1.5*ds));
```

Figure 1

Now things get interesting

The output from this program is somewhat cluttered, and can be confusing. I will be discussing the material that you see on your screen in five steps. First, I will discuss the material in each of the four quadrants. Each quadrant illustrates a different stroking concept implemented by the **BasicStroke** class.

Then I will discuss the circle in the center.

The upper-left quadrant

Let me draw your attention to the upper-left quadrant that contains three horizontal lines. Each line is rendered in a different color. Within the set of three, the top line should be red, the middle line should be green, and the bottom line should be blue.

As you can see, each of these lines has a width of about 0.2 inches. Although it isn't obvious, each of the lines has the same specified length. The difference in the actual length of the three lines has to do with how the end of the line is treated. The exposed ends of each line are *decorated* with a specific *end cap*.

End Caps

The **BasicStroke** class provides three styles of end caps. Here is a list of the three styles along with the horizontal line to which they were applied:

- CAP_BUTT – red line
- CAP_ROUND – green line
- CAP_SQUARE – blue line

What does Flannagan have to say?

Here is what Java Foundation Classes in a Nutshell, by David Flanagan, has to say about the three styles. (Flanagan is referring to constants in the **BasicStroke** class.)

"The **BasicStroke.CAP_BUTT** constant specifies that the line should have no end cap.

The **CAP_SQUARE** constant specifies a rectangular end cap that projects beyond the end point of the line by a distance equal to half the line width; this is the default value for the end-cap attribute.

CAP_ROUND specifies a semicircular end cap, with a radius equal to half of the line width."

The wider the line, the longer the projection

As you can see from this description, the wider the line, the longer will be the projection at the end of the line for the latter two end-cap styles.

As you can see from your screen, the actual length of the green and blue lines (including the end-cap projection) is the same, with each of these lines being longer than the red line by half the line width on each end.

CAP_SQUARE can lead to confusion

The fact that **CAP_SQUARE** is the default can lead to confusion if you don't understand what is going on.

This means that by default, every line that you draw (whose width is greater than one pixel) will actually be rendered with a greater length than you specified when you instantiated the **Line2D** object.

By default, the total rendered length of the line will be the specified length plus the width.

Red CAP_BUTT

Figure 2 prepares the **Graphics2D** object to render lines with a width of 0.2 inches (see previous caveat about dimensions in inches) and a **CAP_BUTT** end cap, (which is no end cap at all).

```
  Stroke stroke = new BasicStroke(
          0.2f*ds,//width
          BasicStroke.CAP_BUTT,
          BasicStroke.JOIN_BEVEL);//don't care
   g2.setStroke(stroke);
```

Figure 2

Figure 2 instantiates a new **BasicStroke** object with these two parameter specifications, and then passes a reference to that object to the **setStroke()** method of the **Graphics2D** object.

(Figure 2 also sets a **JOIN_BEVEL** but that has no impact on the three lines in the upper-left quadrant. I will discuss the join styles later.)

Render the red horizontal line

Figure 3 renders a new **Line2D.Double** with a width of 0.2 inches. **Graphics2D** uses the width and end cap specification established by passing the **BasicStroke** object to the **setStroke()** method. Since there are no intersecting line segments, the JOIN_BEVEL has no effect.

```
  g2.setPaint(Color.red);
  g2.draw(new Line2D.Double(
           -1.5*ds,-1.5*ds,-0.5*ds,-1.5*ds));
```

Figure 3

Green line with round end caps

Figure 4 is very similar to the previous one. In this fragment, the **CAP_ROUND** constant is passed to the **BasicStroke** constructor, resulting in a horizontal line with round caps on each end.

```
  stroke = new BasicStroke(
          0.2f*ds,//width
          BasicStroke.CAP_ROUND,
          BasicStroke.JOIN_BEVEL);//don't care
  g2.setStroke(stroke);
  g2.setPaint(Color.green);
  g2.draw(new Line2D.Double(
           -1.5*ds,-1.0*ds,-0.5*ds,-1.0*ds));
```

Figure 4

Blue line with square end caps

Similarly, Figure 5 produces a blue horizontal line with square end caps.

```
stroke = new BasicStroke(
       0.2f*ds,//width
       BasicStroke.CAP_SQUARE,
       BasicStroke.JOIN_BEVEL);//don't care
g2.setStroke(stroke);
g2.setPaint(Color.blue);
g2.draw(new Line2D.Double(
         -1.5*ds,-0.5*ds,-0.5*ds,-0.5*ds));
```

Figure 5

Line Joins

One issue that arises when line widths are greater than one pixel has to do with how to treat the points where the lines intersect to produce vertices. Java 2D deals with this issue using a technique known as *line joins*.

Look at the lower-left quadrant

Please concentrate on the lower-left quadrant. This quadrant illustrates the kind of problem that can arise without the availability of *line joins*.

The lower left quadrant contains four lines. There are two red lines and two green lines.

An intersection with square end caps

The two red lines begin at the same point and proceed downward at a slight angle to one another. If your screen display looks like mine, the intersection of those two lines appears to have ears on the sides with a slight depression on the top.

This is the result of drawing two independent lines (they are not segments of the same **Shape**) that begin at the same point and are decorated on the ends with a **CAP_SQUARE** end cap. This is shown in Figure 6.

```
stroke = new BasicStroke(
        0.2f*ds,//width
        BasicStroke.CAP_SQUARE,
        BasicStroke.JOIN_BEVEL);//don't care
g2.setStroke(stroke);
g2.setPaint(Color.red);
```

```
  g2.draw(new Line2D.Double(
        -1.75*ds,1.5*ds,-1.50*ds,0.5*ds));
  g2.draw(new Line2D.Double(
        -1.50*ds,0.5*ds,-1.25*ds,1.5*ds));
```

Figure 6

When two lines are joined in this fashion, the appearance of the joint is not pleasing.

An intersection with no end caps

The two green lines show the intersection of two lines with no end caps. While not quite as ugly, this joint is still not particularly pleasing. These two lines were produced by the code in Figure 7, which is not much different from the previous fragment except for the specification of a different end-cap style and a different color.

```
  stroke = new BasicStroke(
       0.2f*ds,//width
       BasicStroke.CAP_BUTT,
       BasicStroke.JOIN_BEVEL);//don't care
  g2.setStroke(stroke);
  g2.setPaint(Color.green);
  g2.draw(new Line2D.Double(
         -0.75*ds,1.5*ds,-0.5*ds,0.5*ds));
  g2.draw(new Line2D.Double(
         -0.5*ds,0.5*ds,-0.25*ds,1.5*ds));
```

Figure 7

You will note that the two green lines appear to be shorter than the two red lines, even though all four lines were specified to be the same length in their **Line2D.Double** constructors.

So, what's the answer to ugly intersections

Now please concentrate on the upper-right quadrant.

As we saw earlier, the constructor for **BasicStroke** allows you specify a decoration style for the intersection (or join) of two line segments of the same **Shape** object.

It is important to note that this capability does not apply to independent lines whose ends just happen to intersect as in the previous example.

Types of Line Joins

The **BasicStroke** class provides three styles of joins. Here is a list of the three styles along with the lines to which they were applied in the upper-right quadrant:

- JOIN_BEVEL – blue lines
- JOIN_MITER – green lines
- JOIN_ROUND – red lines

Here is what David Flanagan, has to say about the three styles.

"The default join style is a mitered join, represented by the **Basic.Stroke.JOIN_MITER** constant. This value specifies that lines are joined by extending their outer edges until they meet.

The **JOIN_BEVEL** constant specifies that lines are joined by drawing a straight line between the outside corners of the two lines, while **JOIN_ROUND** specifies that the vertex formed by the two lines should be rounded, with a radius of half the line with."

Note that exposed ends of line segments are subject to the application of end caps while the intersections are subject to the application of line joins.

The miterlimit parameter

When the angle between two line segments is small, the **JOIN_MITER** style can produce an undesirable result consisting of a very long pointed decoration at the joint. For that reason, the constructor for **BasicStroke** allows you to specify a parameter value to prevent this. Here is what Flanagan has to say about the **miterlimit** parameter.

"**BasicStroke** includes another attribute known as the miter limit. If the miter would be longer than this value times half of the line width, it is truncated. The default for miterlimit is 10.0."

I will show you an example of the use of the **miterlimit** parameter in the bottom-right quadrant. For the moment, however, please continue to concentrate on the upper-right quadrant.

Blue JOIN_BEVEL with CAP_SQUARE end caps

Figure 8 prepares the **Graphics2D** object to apply a **JOIN_BEVEL** to the intersection of any line segments (belonging to the same **Shape** object) that are rendered.

```
   stroke = new BasicStroke(
            0.2f*ds,//width
            BasicStroke.CAP_SQUARE,
            BasicStroke.JOIN_BEVEL);
    g2.setStroke(stroke);
```

Figure 8

Apply the JOIN_BEVEL

Figure 9:

- Instantiates a **Shape** object consisting of two line segments that intersect at an angle
- Sets the **paint** property to the color blue
- Draws the outline of the **Shape** object.

As you can see on your screen output, the join decoration at the intersection is a straight line as described by Flanagan above.

```
   GeneralPath gp1 = new GeneralPath();
   gp1.moveTo(0.25f*ds,-1.25f*ds);
   gp1.lineTo(0.50f*ds,-0.25f*ds);
   gp1.lineTo(0.75f*ds,-1.25f*ds);

   g2.setPaint(Color.blue);
   g2.draw(gp1);
```

Figure 9

Green JOIN_MITER with CAP_ROUND end caps

Figure 10 is very similar, except that it uses round end caps and a miter join.

```
   stroke = new BasicStroke(
            0.2f*ds,//width
            BasicStroke.CAP_ROUND,
            BasicStroke.JOIN_MITER);
   g2.setStroke(stroke);

   GeneralPath gp2 = new GeneralPath();
   gp2.moveTo(0.75f*ds,-0.25f*ds);
   gp2.lineTo(1.00f*ds,-1.25f*ds);
   gp2.lineTo(1.25f*ds,-0.25f*ds);

   g2.setPaint(Color.green);
```

```
    g2.draw(gp2);

Figure 10
```

In this case, you can see that the miter join produces a pointed decoration at the point where the green line segments intersect.

Red JOIN_ROUND with no end caps

The two red line segments in the upper-right quadrant exhibit a round decoration at the intersection with no end caps, as created by the code in Figure 11.

```
    stroke = new BasicStroke(
             0.2f*ds,//width
             BasicStroke.CAP_BUTT,
             BasicStroke.JOIN_ROUND);
    g2.setStroke(stroke);

    GeneralPath gp3 = new GeneralPath();
    gp3.moveTo(1.25f*ds,-1.25f*ds);
    gp3.lineTo(1.50f*ds,-0.25f*ds);
    gp3.lineTo(1.75f*ds,-1.25f*ds);

    g2.setPaint(Color.red);
    g2.draw(gp3);

Figure 11
```

Dash Patterns and the Miter Limit

Now please concentrate on the bottom-right quadrant for an illustration of *dash patterns* and the *miter limit*.

The blue and red line segments illustrate the use of dash patterns. The green line segments illustrates the use of a miter limit with no dash pattern.

Note that the three geometric figures in the lower-right quadrant are mirror images of the geometric figures immediately above them, but with the additional *dash pattern* and *miter limit* decorations applied.

Dash patterns

To review, here is how Sun describes dash patterns

dash attributes: The definition of

how to make a dash pattern by alternating between opaque and transparent sections.

Interaction between dash patterns and end caps

What this doesn't say is that the end cap is applied to each end of each opaque section. As you will see shortly, this can lead to some confusion due to the fact that the end cap extends the length of the opaque sections.

The blue and red geometric figures in the bottom right-hand quadrant illustrates this situation. Each of these two figures has the same dash pattern applied. However, on the blue figure, the extension of the opaque sections by a **CAP_SQUARE** tends to fill the transparent sections.

The red figure, on the other hand, doesn't apply an end cap so the transparent sections are much longer than on the blue figure. Obviously, this is something that you can cope with as long as you understand what is going on.

Interaction between dash patterns and join decoration

The dash pattern can also interact in unpredictable ways with the decoration at the join between two line segments. This is also illustrated by the blue and red figures in the lower-right quadrant. In this case, a transparent section occurred coincident with the join decoration causing the decoration to simply disappear.

Applying the dash pattern

In order to apply a dash pattern, you provide a reference to an array of type **float** as the fifth parameter to the constructor for a **BasicStroke** object.

This array can contain any number of **float** elements (although it probably makes more sense to specify them in pairs). Each element specifies the length of one section in a series of alternating opaque and transparent sections.

One pair was used in this program

In this program, the specification was for repetition of a single pair of sections. The length of the opaque section was specified to be 0.1 inches. The length of the transparent section is specified to be 0.3 inches.

That is pretty close to what I see on my screen, for the red figure, but due to the end-cap extension problem mentioned earlier, that is not what I see for the blue figure. The opaque sections are much longer than the transparent sections for the blue figure.

Figure 12 shows the code that produced the blue figure.

```
  stroke = new BasicStroke(
           0.2f*ds,//width
           BasicStroke.CAP_SQUARE,
           BasicStroke.JOIN_BEVEL,
           0.0f,//miterlimit doesn't matter
           //Dash pattern
           new float[] {0.1f*ds,0.3f*ds},
           0.0f);//Dash phase
  g2.setStroke(stroke);
  GeneralPath gp4 = new GeneralPath();
  gp4.moveTo(0.25f*ds,1.25f*ds);
  gp4.lineTo(0.50f*ds,0.25f*ds);
  gp4.lineTo(0.75f*ds,1.25f*ds);
  g2.setPaint(Color.blue);
  g2.draw(gp4);
```

Figure 12

The only thing new in this fragment is the specification of the dash pattern in the array, and the value for the *Dash Phase*.

What is a *Dash Phase*?

In case you don't want the pattern to begin with the first element in the array, you can provide a **float** value for the sixth parameter. This value specifies a distance into the theoretical pattern that is used as the starting point for the actual pattern. In this fragment, that distance was specified as 0.0.

The miterlimit

As you can see, the miter for the green figure in the lower-right quadrant was truncated (relative to that shown for the green figure in the upper-right quadrant).

This truncation resulted from specifying a miterlimit value in the fourth parameter to the constructor shown in Figure 13. I had a little difficulty coming up with a value for this parameter that would do the job. I'm not certain that this is the appropriate value for all screen resolutions, and you may need to experiment with the value for your screen resolution. (If the green figure in the bottom-right quadrant has a pointed join style, it isn't working.)

```
  stroke = new BasicStroke(
           0.2f*ds,//width
           BasicStroke.CAP_ROUND,
           BasicStroke.JOIN_MITER,
           .057f*ds);//miterlimit
  g2.setStroke(stroke);

  GeneralPath gp5 = new GeneralPath();
  gp5.moveTo(0.75f*ds,0.25f*ds);
  gp5.lineTo(1.00f*ds,1.25f*ds);
  gp5.lineTo(1.25f*ds,0.25f*ds);

  g2.setPaint(Color.green);
  g2.draw(gp5);
```

Figure 13

The specification of the miterlimit value is the only thing new in this fragment.

The red figure

The code in Figure 14 produced the red figure in the lower-right quadrant. As mentioned earlier, the dash pattern for this figure is the same as for the blue figure discussed earlier, even though they look considerably different on the screen.

```
  stroke = new BasicStroke(
            0.2f*ds,//width
            BasicStroke.CAP_BUTT,
            BasicStroke.JOIN_ROUND,
            0.0f,//miterlimit doesn't matter
            //Dash Pattern
            new float[] {0.1f*ds,0.3f*ds},
            0.0f);//Dash phase
   g2.setStroke(stroke);

   GeneralPath gp6 = new GeneralPath();
   gp6.moveTo(1.25f*ds,1.25f*ds);
   gp6.lineTo(1.50f*ds,0.25f*ds);
   gp6.lineTo(1.75f*ds,1.25f*ds);

   g2.setPaint(Color.red);
   g2.draw(gp6);
```

Figure 14

The red figure provides a much closer representation of what you would expect, considering only the values in the **float** array and not taking end-cap extension into account.

The Circle in the Center

Finally, that brings us to the circle in the center of the **Frame** object. The circle was provided as a capstone for the other things discussed earlier in the lesson.

Figure 15 produces an orange outline of a circle centered on the origin. There are no end-cap extensions. The dash pattern has an opaque section of 0.2 inches, followed by a transparent section of 0.1 inches.

```
  stroke = new BasicStroke(
        0.1f*ds,//width
        BasicStroke.CAP_BUTT,
        BasicStroke.JOIN_ROUND,//don't care
        0.0f,//miterlimit doesn't matter
        //Dash pattern
        newfloat[] {0.2f*ds,0.1f*ds},
        0.0f);//Dash phase
  g2.setStroke(stroke);

  Ellipse2D.Double theCircle =
         new Ellipse2D.Double(
            -0.4*ds,-0.4*ds,0.8*ds,0.8*ds);

  g2.setPaint(Color.orange);
  g2.draw(theCircle);
```

Figure 15

Note that these section lengths didn't come out even on the circumference of the circle, so there is an extra long segment on the right-hand side of the circle. At least, that is the case on my machine.

The width of the **Shape** that represents the outline is 0.1 inches.

Summary

In this lesson, I have shown you how to use the following attributes of the **BasicStroke** class to produce a **Shape** object that represents the outline of another **Shape** object

- width
- end caps
- line joins
- dash patterns

Although I didn't illustrate it, I also explained how to use the *dash phase* in conjunction with *dash patterns*.

I also provided an illustration that shows why we need *line joins* to deal with the situation involving vertices produced by intersecting line segments.

Complete Program Listing

A complete listing of the program is provided in Figure 16.

```
/*Stroke01.java 12/12/99
 Copyright 1999, R.G.Baldwin
```

```
Illustrates use of the Stroke interface.

Draws a 4-inch by 4-inch Frame on the screen.

Translates the origin to the center of the Frame.

Draws a pair of X and Y-axes
centered on the new origin.

Illustrates three types of end caps in upper-left
quadrant.

Illustrates connecting lines that
are not segments of a
Shape in lower-left quadrant

Illustrates three types of line joins, along with end
caps in upper-right quadrant.

Illustrates dash pattern and miterlimit in lower-right
quadrant.

Illustrates application of several
attributes to a circle
centered on the origin.

Whether the dimensions in inches come out right
or not depends on whether the method
getScreenResolution() returns the correct
resolution for your screen.

Tested using JDK 1.2.2,WinNT Workstation 4.0
****************************************/
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class Stroke01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Stroke01

class GUI extends Frame{
  int res;//store screen resolution here
  staticfinalint ds = 72;//default scale, 72 units/inch
  staticfinalint hSize = 4;//horizonal size = 4 inches
  staticfinalint vSize = 4;//vertical size = 4 inches

  GUI(){//constructor
    //Get screen resolution
    res = Toolkit.getDefaultToolkit().
                                getScreenResolution();
    //Set Frame size
    this.setSize(hSize*res,vSize*res);
    this.setVisible(true);
    this.setTitle("Copyright 1999, R.G.Baldwin");

    //Window listener to terminate program.
    this.addWindowListener(new WindowAdapter(){
      publicvoid windowClosing(WindowEvent e){
        System.exit(0);}});
  }//end constructor
  //----------------------------------------------------//

  //Override the paint() method
  publicvoid paint(Graphics g){
    //Downcast the Graphics object to a
    // Graphics2D object
    Graphics2D g2 = (Graphics2D)g;

    //Scale device space to produce inches on the
    // screen based on actual screen resolution.
    g2.scale((double)res/72,(double)res/72);

    //Translate origin to center of Frame
    g2.translate((hSize/2)*ds,(vSize/2)*ds);

    //Draw x-axis
    g2.draw(new Line2D.Double(
                              -1.5*ds,0.0,1.5*ds,0.0));
    //Draw y-axis
    g2.draw(new Line2D.Double(
                              0.0,-1.5*ds,0.0,1.5*ds));


    //Display all three end cap types in upper-left
    // quadrant Display red CAP_BUTT
    Stroke stroke = new BasicStroke(
          0.2f*ds,//width
          BasicStroke.CAP_BUTT,
          BasicStroke.JOIN_BEVEL);//don't care
    g2.setStroke(stroke);
    g2.setPaint(Color.red);
    g2.draw(new Line2D.Double(
                   -1.5*ds,-1.5*ds,-0.5*ds,-1.5*ds));

    //Display green CAP_ROUND
    stroke = new BasicStroke(
          0.2f*ds,//width
          BasicStroke.CAP_ROUND,
          BasicStroke.JOIN_BEVEL);//don't care
    g2.setStroke(stroke);
    g2.setPaint(Color.green);
    g2.draw(new Line2D.Double(
                   -1.5*ds,-1.0*ds,-0.5*ds,-1.0*ds));

    //Display blue CAP_SQUARE
    stroke = new BasicStroke(
          0.2f*ds,//width
          BasicStroke.CAP_SQUARE,
          BasicStroke.JOIN_BEVEL);//don't care
    g2.setStroke(stroke);
    g2.setPaint(Color.blue);
    g2.draw(new Line2D.Double(
                    -1.5*ds,-0.5*ds,-0.5*ds,-0.5*ds));

    //Display two lines that connect, but are not
    // segments of a Shape in the lower left
    // quadrant.  Illustrates the problems of creating
    // geometric figures with connecting lines that
    // have width.

    //This illustrates the problem with
    // CAP_SQUARE -- red
    stroke = new BasicStroke(
          0.2f*ds,//width
          BasicStroke.CAP_SQUARE,
          BasicStroke.JOIN_BEVEL);//don't care
    g2.setStroke(stroke);
    g2.setPaint(Color.red);
    g2.draw(new Line2D.Double(
                   -1.75*ds,1.5*ds,-1.50*ds,0.5*ds));
    g2.draw(new Line2D.Double(
                   -1.50*ds,0.5*ds,-1.25*ds,1.5*ds));

    //This illustrates the problem with
    // CAP_BUTT -- green
```

```
stroke = new BasicStroke(
        0.2f*ds,//width
        BasicStroke.CAP_BUTT,
        BasicStroke.JOIN_BEVEL);//don't care
g2.setStroke(stroke);
g2.setPaint(Color.green);
g2.draw(new Line2D.Double(
                -0.75*ds,1.5*ds,-0.5*ds,0.5*ds));
g2.draw(new Line2D.Double(
                -0.5*ds,0.5*ds,-0.25*ds,1.5*ds));

//Display all three join types in upper-right
// quadrant

//Display blue JOIN_BEVEL with CAP_SQUARE
stroke = new BasicStroke(
        0.2f*ds,//width
        BasicStroke.CAP_SQUARE,
        BasicStroke.JOIN_BEVEL);
g2.setStroke(stroke);
GeneralPath gp1 = new GeneralPath();
gp1.moveTo(0.25f*ds,-1.25f*ds);
gp1.lineTo(0.50f*ds,-0.25f*ds);
gp1.lineTo(0.75f*ds,-1.25f*ds);
g2.setPaint(Color.blue);
g2.draw(gp1);

//Display green JOIN_MITER with CAP_ROUND
stroke = new BasicStroke(
        0.2f*ds,//width
        BasicStroke.CAP_ROUND,
        BasicStroke.JOIN_MITER);
g2.setStroke(stroke);

GeneralPath gp2 = new GeneralPath();
gp2.moveTo(0.75f*ds,-0.25f*ds);
gp2.lineTo(1.00f*ds,-1.25f*ds);
gp2.lineTo(1.25f*ds,-0.25f*ds);

g2.setPaint(Color.green);
g2.draw(gp2);

//Display red JOIN_ROUND with CAP_BUTT
stroke = new BasicStroke(
        0.2f*ds,//width
        BasicStroke.CAP_BUTT,
        BasicStroke.JOIN_ROUND);
g2.setStroke(stroke);

GeneralPath gp3 = new GeneralPath();
gp3.moveTo(1.25f*ds,-1.25f*ds);
gp3.lineTo(1.50f*ds,-0.25f*ds);
gp3.lineTo(1.75f*ds,-1.25f*ds);

g2.setPaint(Color.red);
g2.draw(gp3);

//Display dash pattern and miterlimit in
// bottom-right quadrant

//Display blue JOIN_BEVEL with CAP_SQUARE
//Dash pattern is one on, three off, but this is not
// what it looks like with CAP_SQUARE
stroke = new BasicStroke(
      0.2f*ds,//width
      BasicStroke.CAP_SQUARE,
      BasicStroke.JOIN_BEVEL,
      0.0f,//miterlimit doesn't matter
      newfloat[] {0.1f*ds,0.3f*ds},//Dash pattern
      0.0f);//Dash phase
```

```
    g2.setStroke(stroke);
    GeneralPath gp4 = new GeneralPath();
    gp4.moveTo(0.25f*ds,1.25f*ds);
    gp4.lineTo(0.50f*ds,0.25f*ds);
    gp4.lineTo(0.75f*ds,1.25f*ds);
    g2.setPaint(Color.blue);
    g2.draw(gp4);

    //Display green JOIN_MITER with CAP_ROUND
    // and miter limit. No dash pattern.
    stroke = new BasicStroke(
              0.2f*ds,//width
              BasicStroke.CAP_ROUND,
              BasicStroke.JOIN_MITER,
              .057f*ds);//miterlimit
    g2.setStroke(stroke);

    GeneralPath gp5 = new GeneralPath();
    gp5.moveTo(0.75f*ds,0.25f*ds);
    gp5.lineTo(1.00f*ds,1.25f*ds);
    gp5.lineTo(1.25f*ds,0.25f*ds);

    g2.setPaint(Color.green);
    g2.draw(gp5);

    //Display red JOIN_ROUND with CAP_BUTT
    //Dash pattern is one on, three off again.  Looks
    // like it with CAP_BUTT
    stroke = new BasicStroke(
           0.2f*ds,//width
           BasicStroke.CAP_BUTT,
           BasicStroke.JOIN_ROUND,
           0.0f,//miterlimit doesn't matter
           newfloat[] {0.1f*ds,0.3f*ds},//Dash pattern
           0.0f);//Dash phase
    g2.setStroke(stroke);

    GeneralPath gp6 = new GeneralPath();
    gp6.moveTo(1.25f*ds,1.25f*ds);
    gp6.lineTo(1.50f*ds,0.25f*ds);
    gp6.lineTo(1.75f*ds,1.25f*ds);

    g2.setPaint(Color.red);
    g2.draw(gp6);

    //Draw a circle with an orange outline centered
    // on the origin with a dash pattern and
    // CAP_BUTT end caps.
    stroke = new BasicStroke(
           0.1f*ds,//width
           BasicStroke.CAP_BUTT,
           BasicStroke.JOIN_ROUND,//don't care
           0.0f,//miterlimit doesn't matter
           newfloat[] {0.2f*ds,0.1f*ds},//Dash pattern
           0.0f);//Dash phase
    g2.setStroke(stroke);

    Ellipse2D.Double theCircle =
                                 new Ellipse2D.Double(
                        -0.4*ds,-0.4*ds,0.8*ds,0.8*ds);

    g2.setPaint(Color.orange);
    g2.draw(theCircle);

  }//end overridden paint()

}//end class GUI
//==============================//
```

Figure 16

About the author

__Richard Baldwin__ is a college professor and private consultant whose primary focus is a combination of Java and XML. In addition to the many platform-independent benefits of Java applications, he believes that a combination of Java and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects involving Java, XML, or a combination of the two. He frequently provides onsite Java and/or XML training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Java Programming Tutorials, which has gained a worldwide following among experienced and aspiring Java programmers. He has also published articles on Java Programming in Java Pro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

-end-