March 6, 2000

Java 2D Graphics, The Shape Interface, Part 2

Java Programming, Lecture Notes # 310

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

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.

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.

This and an earlier lesson are intended to give you the necessary understanding of the **Shape** interface and the **PathIterator** class.

What is a Shape?

As I reported in the earlier lesson, according to Sun:

> "The **Shape** interface provides definitions for objects that represent some form of geometric shape. The **Shape** is described by a **PathIterator** object, which can express the outline of the **Shape** as well as a rule for determining how the outline divides the

2D plane into interior and exterior points. Each **Shape** object provides callbacks to get the bounding box of the geometry, determine whether points or rectangles lie partly or entirely within the interior of the **Shape**, and retrieve a **PathIterator** object that describes the trajectory path of the Shape outline."

What is a PathIterator Object?

The **Shape** interface provides four groups of overloaded methods that make it possible to perform the following tasks:

- Get a bounding box that is guaranteed to enclose a specified **Shape** object.
- Determine if a specified **Shape** object completely contains a specified point or rectangle.
- Determine if a specified **Shape** intersects a specified rectangle
- Get a **PathIterator** object that can be used to obtain the path that makes up the boundary of a **Shape** object.

I discussed the first three capabilities in the earlier lesson. I told you that I was going to defer a discussion of the **PathIterator** to this lesson. This lesson is dedicated the **PathIterator** interface, but in order to understand the behavior of that class, it will be instructive to provide a brief discussion and illustration of the **GeneralPath** class. You might consider the **PathIterator** and the **GeneralPath** to be opposite sides of the same coin.

This is part of what Sun has to say about **PathIterator**.

The **PathIterator** interface provides the mechanism for objects that implement the **Shape** interface to return the geometry of their boundary by allowing a caller to retrieve the path of that boundary a segment at a time.

In other words, **PathIterator** makes it possible for code in a program to obtain information about the geometric outline of an object that implements the **Shape** interface.

What is a GeneralPath Object?

Here is part of what Sun has to say about **GeneralPath**.

"The **GeneralPath** class represents a

geometric path constructed from straight lines, and quadratic and cubic (Bezier) curves. It can contain multiple subpaths.

The winding rule specifies how the interior of a path is determined. There are two types of winding rules: EVEN_ODD and NON_ZERO.

An EVEN_ODD winding rule..."

I plan to discuss the **GeneralPath** class in much more detail in a subsequent lesson. In this lesson, I will simply show you how to create a geometric shape using **GeneralPath**. Then I will show you how to use **PathIterator** to analyze that shape and to replicate it with an offset.

Let's listen to David Flanagan

Here is what Java Foundation Classes in a Nutshell, by David Flanagan, has to say about **GeneralPath**.

"This class represents an arbitrary path or shape that consists of any number of line segments and quadratic and cubic Bezier curves. After creating a **GeneralPath** object, you must define a current point by calling **moveTo()**. Once an initial current point is established, you can create the path by calling **lineTo()**, **quadTo()**, and **curveTo()**. These methods draw line segments, quadratic curves, and cubic curves from the current point to a new point (which becomes the new current point).

The shape defined by a **GeneralPath** need not be closed, although you may close it by calling the **closePath()** method, which appends a line segment between the current point and the initial point... The **append()** method allows you to add a **Shape** or **PathIterator** to a **GeneralPath**, optionally connecting it to the current point with a straight line."

Since my purpose in using **GeneralPath** in this lesson is to provide background for understanding **PathIterator**, I will keep it simple and deal only with the following methods of **GeneralPath**:

- moveTo()
- lineTo()
- closePath()

What does Jonathan Knudsen have to say?

This is what Jonathan Knudsen has to say on the subject in his excellent book entitled Java 2D Graphics, from O'Reilly.

"Lurking behind the **Shape** interface, there's a handy toolbox of shapes in the **java.awt.geom** package – rectangles, ellipses, and so on. I'll discuss these soon. If you want to draw pentagons, decagons, stars, or something completely different, you'll have to describe the path yourself using a **java.awt.geom.GeneralPath**. This class implements the **Shape** interface and allows you to build a path, segment by segment."

So, what's the bottom line?

The bottom line is that a **Shape** object is constructed from segments consisting of moves (pen up), lines, curves, and an optional closure. **GeneralPath** can be used to construct such a **Shape**, one segment at a time.

The **Shape** object remembers how it was originally constructed in terms of the types of segments, their coordinate values, etc. The object can tell us how it was originally constructed. The **PathIterator** is used to extract that information from the **Shape** object.

The two following programs show how to construct a simple **Shape** consisting of moves, lines, and a closure, and how to extract the necessary information from the **Shape** to replicate it with an offset. I didn't use Bezier curves. The topic of Bezier curves is sufficiently complex as to merit a lesson of its own. I will be discussing the use of Bezier curves in a subsequent lesson.

Sample Program GeneralPath01

This first sample program illustrates the use of the **GeneralPath** class to construct a simple **Shape** object.

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

Then the program uses **GeneralPath** to create a diamond **Shape** and draw it on the **Frame**. The vertices of the diamond shape are at plus and minus one-half inch on each of the axes. When viewed on the screen, the vertices are at the North, South, East, and West positions and the diamond is centered on the origin.

The program was tested using JDK 1.2.2 under WinNT Workstation 4.0.

Will discuss in fragments

As is usually the case, 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 bore you by repeating that discussion here. You can view this 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 simply going to let the comments in Figure 1 speak for themselves.

```
public void 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

The interesting part

Now we have arrived at the interesting part. Figure 2 begins by instantiating a new object of the **GeneralPath** class.

```
GeneralPath thePath = new GeneralPath();

thePath.moveTo(0.5f*ds,0.0f*ds);
```

Figure 2

Then it invokes the **moveTo()** method to establish the *current point* (as described above by Flanagan).

An analogy

As a practical matter (see the caveat below), if you think of the creation and population of a **GeneralPath** object as analogous to creating a line drawing using a pen and paper, invocation of the **moveTo()** method corresponds to moving the drawing pen without the tip of the pen touching the paper.

Create some lines

Continuing with our pen and paper analogy, the program invokes the **lineTo()** method three times in succession to draw three straight lines on the paper. Given the coordinates shown in Figure 3, these lines form three sides of a diamond shape.

```
thePath.lineTo(0.0f*ds,0.5f*ds);
thePath.lineTo(-0.5f*ds,0.0f*ds);
thePath.lineTo(0.0f*ds,-0.5f*ds);
```

Figure 3

Close the path

Finally, the program invokes the **closePath()** method to draw a single straight line from the current point back to the current point established by the original **moveTo()** method call. This line forms the fourth side of a diamond shape. See Figure 4 for this code fragment.

```
  thePath.closePath();
```

Figure 4

The analogy is a problem!

The problem with using the pen and paper analogy to describe the process is that the process implemented in the above fragments really doesn't draw anything (not anything that a human can see, anyway). Rather, it creates and populates an object that implements the **Shape** interface that exist only in the computer's memory.

This is a very important point that we must keep in mind. Creating a **Shape** object in the computer's memory is not the same thing as rendering that object on a graphics display device. The object doesn't become visible until it is rendered. However, the fact that it isn't visible doesn't mean that it doesn't exist. That just means that it hasn't been rendered.

The **Shape** object can be rendered onto an output device using the **draw(Shape)** method shown in Figure 5.

```
  g2.draw(thePath);
```

Figure 5

When the virtual object created and populated by the previous fragments is rendered, it will look like a diamond shape, centered on the origin, with the vertices at the North, South, East, and West positions. The shape of the object and its location is controlled by the coordinate information passed to the **moveTo()**, and **lineTo()** methods in the above fragments, and the attributes of the **AffineTransform** object currently associated with the **Graphics2D** object.

Now that we know how to create and populate a **GeneralPath** object, the next program will illustrate how to extract the necessary information from that object to replicate it.

Sample Program PathIterator01

The purpose of this program is to illustrate the use of the **getPathIterator()** method of the **Shape** interface, and the **PathIterator** object returned by that method. The program uses that object to learn all that there is to know about an object that implements the **Shape** interface.

The program creates and populates a **GeneralPath** object identical to that in the previous program.

Then the program invokes **getPathIterator()** on that **GeneralPath** object to obtain a **PathIterator** object that provides access to information about the **GeneralPath** object.

Then the program invokes various methods on the **PathIterator** object to extract the pertinent information about the **GeneralPath** object that it represents. This information is used to

replicate the original object, offset by one inch in both directions. The segment information is also displayed on the command-line screen so that you can compare it with what you know to be true about the original object.

Then the program draws the new object in red.

Discuss in fragments

I will discuss this program in fragments. This program is identical to the previous program down to the point where the original **GeneralPath** object has been created, populated, and displayed on a **Frame** object. Therefore, I won't repeat the discussion of that material. You can view the code that accomplishes this in the complete listing of the program at the end of the lesson.

For continuity, this fragment picks up at the point in the overridden **paint()** method where the original object, referred to by **thePath**, is being rendered on the **Frame**. Once the following fragment has been executed, the original object has been rendered on the screen. See Figure 6 for the code fragment.

```
// black
 g2.draw(thePath);
```

Figure 6

Get a PathIterator object

Figure 7 invokes the **getPathIterator()** method on the original object to get a **PathIterator** object that represents it.

```
PathIterator theIterator =
        thePath.getPathIterator(null);
```

Figure 7

Note that this program passes **null** to the **getPathIterator()** method. It is also possible to pass a reference to an **AffineTransform** object to the method. In that case, the **PathIterator** object will represent a transformed version of the target object with the nature of the transformation being determined by the **AffineTransform** object.

Since my objective here is to reproduce the original object with an offset in both the horizontal and vertical directions, I could have passed a translation transform object as a parameter. Then the **PathIterator** that is returned would represent the original object translated to a different location in the 2D space. However, I elected to perform the translation numerically when populating a new **GeneralPath** object instead. Therefore, I passed **null** as a parameter to **getPathIterator()**.

A flattened PathIterator object

The **PathIterator** object that is returned could contain segments consisting of Bezier curves. That may not be what you want. Another overloaded version of **getPathIterator(AffineTransform at, double flatness)** takes two parameters and returns a **PathIterator** object in which all Bezier curves have been replaced by a series of straight-line segments.

The **flatness** parameter defines how well the straight lines represent the curve. In particular, this parameter defines the maximum distance that any point on the curve can deviate from the straight line that represents the curve at that point. Thus, in general, the smaller the **flatness** parameter, the more straight line segments will be generated to represent the curve, and the better will be the straight line approximation of the curve.

Data storage

Later on, I am going to need some place to store two kinds of information about the segments that represent the **GeneralPath** object. I will need to store the *type* of segment, which is of type **int**. I will also need to store *coordinate information* about the segment.

Figure 8 declares two local variables that will be used later to store this information.

```
//Use this array to store segment
// coordinate data
float[] theData = new float[6];
int theType;//store type of segment here
```

Figure 8

Further explanation of these variables

I will be invoking the **currentSegment()** method on the **PathIterator** object to get information about the segment. This method returns the type, which I will store in **theType**.

I will pass the array named **theData** to the **currentSegment()** method. The method will populate the elements in the array with the coordinate information describing the segment. For the case where the segment can be a Bezier curve, the size of this array needs to be six elements, as described in the following information from Sun regarding the **currentSegment()** method (note that there is also a version of this method that deals in coordinate data of type **double**).

"Returns the coordinates and type of the current path segment in the iteration. The return value is the path segment type: SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, or SEG_CLOSE.

A **float** array of length 6 must be passed in and can be used to store the

coordinates of the point(s). Each point is stored as a pair of **float** x, y coordinates.

SEG_MOVETO and SEG_LINETO types returns one point,
SEG_QUADTO returns two points,
SEG_CUBICTO returns 3 points and
SEG_CLOSE does not return any points."

Since I was the one who created the **Shape** object that this **PathIterator** object will represent, and since I know that it doesn't contain any **SEG_QUADTO** or **SEG_CUBICTO** segments, I could have gotten by with a two-element array. However, to be more general, I used a six-element array.

Instantiate a new GeneralPath object

My objective is to extract the necessary information from an existing **Shape** object to allow me to replicate that object with a one-inch offset in the horizontal and vertical dimensions. For that, I need a new **GeneralPath** object that I can populate with the information that I extract from the existing object.

Figure 9 instantiates, but does not populate a new **GeneralPath** object.

```
GeneralPath newPath =
                    new GeneralPath();
```

Figure 9

Iterate and populate

At the risk of causing total confusion, I am going to do something unusual here. In particular, I am going to show a code fragment that would ordinarily be a fairly long fragment, but I am going to delete some of code interior to the fragment to make it more manageable. I will explain the code that remains in the fragment after the deletion. Then I will come back in a subsequent fragment and explain the code that I deleted from the fragment.

You can refer to the original code, fully intact, in the complete listing of the program at the end of the lesson if this is confusing.

Use a while loop

Figure 10 uses a while loop to iterate on, and extract information from the existing **Shape** object. This information is used to populate the new **GeneralPath** object inside a **switch** statement, which was deleted from the fragment for brevity.

```
  while(!theIterator.isDone()){
  //while not done
    theType = theIterator.currentSegment(
                                    theData);

    //Populate a segment of the new
    // GeneralPath object.
    switch(theType){
      //... code deleted for brevity ...
    }//end switch

    theIterator.next();
  }//end while loop
```

Figure 10

The three key methods used in this fragment are:

- isDone()
- currentSegment()
- next()

Descriptions of the methods

The invocation of the **isDone()** method is used to provide the loop control parameter. This method tests to determine if the iteration is complete. It returns **true** if all the segments have been read and returns **false** otherwise.

Note that a **not** operator was used to cause the iteration to continue while the iteration is ***not*** *done*.

A description of the **currentSegment()** method was given above. Briefly, it uses a return value and an array parameter to return information about the current segment of the **Shape** object on which the iteration is being performed. This information is used to populate a segment in the new **Shape** object during each iteration.

According to Sun, the **next()** method

> "Moves the iterator to the next segment of the path forwards along the primary direction of traversal as long as there are more points in that direction."

All in all, this is a fairly standard iteration process, not unlike the use of the **Enumeration** interface that I discussed in detail in an earlier lesson. If enumeration, or iteration is new to you, you might want to go back and review the material in that lesson.

Populate the new Shape object

Now it's time to go back and discuss the **switch** statement that was deleted from the previous fragment. Remember, this statement occurs inside the **while** loop of the previous fragment. In that fragment, the **currentSegment()** method is used to extract information from the original **Shape** object. That information is used in the following **switch** statement to populate the new object that was instantiated outside the **while** loop.

Figure 11 begins with a repeat of the invocation of the **currentSegment()** method from the previous fragment just to get you oriented. Then it picks up with the beginning of the **switch** statement.

```
    theType = theIterator.currentSegment(
                                    theData);

    switch(theType){
      case PathIterator.SEG_MOVETO :
        System.out.println("SEG_MOVETO");
        newPath.moveTo(theData[0]+1.0f*ds,
                       theData[1]+1.0f*ds);
        break;
```

Figure 11

The **switch** statement compares the segment type returned by the **currentSegment()** method against the five possible segment types. The statement uses the information returned in the array by the **currentSegment()** method to populate the new **Shape** object whenever a match is found.

The code also displays the type of segment on the command-line screen. I will show you the complete output on the command-line screen following the next segment.

Populate a SEG_MOVETO segment

The code in Figure 11 will be executed whenever the type of segment returned by the **currentSegment()** method is **SEG_MOVETO**. By this, I mean when the **int** value returned by the method matches the symbolic constant **SEG_MOVETO** defined in the **PathIterator** interface.

This code invokes the **moveTo()** method on the new **GeneralPath** object to store a segment of that type in the path. The coordinate values passed to the **moveTo()** method are the coordinate values extracted from the original **Shape** object with a one-inch offset in both the horizontal and vertical directions.

The code also displays the type of the segment on the command-line screen, but this is for information only, and is not critical to the process.

Command-line screen output

The execution of the **switch** statement inside the execution of the **while** loop produces the following output on the command line screen.

```
SEG_MOVETO
SEG_LINETO
SEG_LINETO
SEG_LINETO
SEG_CLOSE
```

You will note that, as expected, this is an exact match for the types of segments that were created when the original **Shape** object was created using the methods of the **GeneralPath** class. Thus, as mentioned earlier, a **Shape** object knows about its segment types and coordinate values. It can provide that information to us for whatever purpose we may need it.

Populate a SEG_LINETO segment

Figure 12 does essentially the same thing as the previous fragment. This code is executed when there is a match for **SEG_LINETO**.

```
      case PathIterator.SEG_LINETO :
        System.out.println("SEG_LINETO");
        newPath.lineTo(theData[0]+1.0f*ds,
                       theData[1]+1.0f*ds);
        break;
```

Figure 12

What about SEG_QUADTO and SEG_CUBICTO

Since I was the person who created the original **Shape** object that is being replicated here, and since I knew that the object being replicated did not contain any Bezier curves, I didn't provide the ability to support those segment types. However, I did include those types in the **switch** statement to make it general in nature, as shown in Figure 13.

```
      case PathIterator.SEG_QUADTO :
        System.out.println(
                       "Not supported here");
        break;
      case PathIterator.SEG_CUBICTO :
        System.out.println(
                       "Not supported here");
        break;
```

Figure 13

Populate a SEG_CLOSE segment

Finally, Figure 14 creates a SEG_CLOSE segment in the new **Shape** object whenever a matching segment type is found in the object being replicated.

```
      case PathIterator.SEG_CLOSE :
        System.out.println("SEG_CLOSE");
        newPath.closePath();
```

```
        break;
    }//end switch
```

Figure 14

And that is the end of the **switch** statement.

Render the new object in red

Figure 15 sets the drawing color to red, and draws the new object on the **Frame** object. When you run this program, you should see the original **Shape** object appearing as a diamond, drawn in black, and centered on the origin. The new **Shape** object, which is an offset replica of the original object, is drawn in red, one inch down and one inch to the right of the original object. (Your output may not match the dimensions in inches, depending on actual screen resolution.)

```
  g2.setColor(Color.red);
  g2.draw(newPath);
```

Figure 15

Summary

That brings us to the end of this lesson. In this lesson, I have shown you how to use the **GeneralPath** class to create a new **Shape** object consisting of lines and spaces. I did not show you how to include Bezier curves in your new object. I plan to do that in a subsequent lesson.

The **GeneralPath** class was used in this lesson to support the primary objective of the lesson – learning how to get and use a **PathIterator** object that represents another object that implements the **Shape** interface.

In this lesson, I have shown you

- How to create a simple **Shape** object with the **GeneralPath** class, and
- How to extract information from, and replicate that object using the **getPathIterator()** method and the **PathIterator** object that the method returns.

Complete Program Listings

Complete listings of both programs are provided in Figure 16 and Figure 17.

```
/*GeneralPath01.java 12/12/99
Copyright 1999, R.G.Baldwin

Illustrates use of the GeneralPath class.

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

Translates the orgin to the center of the Frame.
```

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

Draws a GeneralPath object on the Frame. The
object is
a diamond whose vertices are at plus and minus
one-half
inch on each of the axes.  The vertices are
located at
the N, S, E, and W positions.

Tested using JDK 1.2.2 under WinNT Wkstn 4.0
**********************************/
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;

class GeneralPath01{
  public static void main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class GeneralPath01

class GUI extends Frame{
  int res;//store screen resolution here
  //default scale, 72 units/inch
  static final int ds = 72;
  //horizonal size = 4 inches
  static final int hSize = 4;
  //vertical size = 4 inches
  static final int vSize = 4;
  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(){
      public void windowClosing(WindowEvent e){
        System.exit(0);}});
  }//end constructor

  //Override the paint() method
  public void 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));

    //Use the GeneralPath class to instantiate a
    // diamond
    // object whose vertices are at plus and minus
    // one-half inch on each of the axes.  The
```

```
    // vertices in the N, S, E, and W positions,
    // centered
    // about the origin.
    GeneralPath thePath = new GeneralPath();
    thePath.moveTo(0.5f*ds,0.0f*ds);
    thePath.lineTo(0.0f*ds,0.5f*ds);
    thePath.lineTo(-0.5f*ds,0.0f*ds);
    thePath.lineTo(0.0f*ds,-0.5f*ds);
    thePath.closePath();

    //Now draw the diamond on the screen
    g2.draw(thePath);

  }//end overridden paint()

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

Figure 16

```
/*PathIterator01.java 12/12/99
Copyright 1999, R.G.Baldwin

Illustrates use of the GeneralPath class and the
PathIterator class.

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

Translates the orgin to the center of the Frame.

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

Draws a GeneralPath object on the Frame. The object is
a diamond whose vertices are at plus and minus one-half
inch on each of the axes.  The vertices are located at
the N, S, E, and W positions.

Uses getPathIterator() to get a PathIterator on the
GeneralPath (diamond) object.

Extracts information from the PathIterator to populate
another GeneralPath object that is offset by one inch
from the original in both the X and Y directions.

Draws the new GeneralPath object in red.

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

class PathIterator01{
  public static void main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class PathIterator01

class GUI extends Frame{
  int res;//store screen resolution here
  static final int ds = 72;//default scale, 72 units/inch
  static final int hSize = 4;//horizonal size = 4 inches
  static final int 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(){
    public void windowClosing(WindowEvent e){
      System.exit(0);}});
}//end constructor

//Override the paint() method
public void 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 the origin to the center of the 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));

  //Use the GeneralPath class to instantiate a diamond
  // object whose vertices are at plus and minus
  // one-half inch on each of the axes. The
  // vertices in the N, S, E, and W positions, centered
  // about the origin.
  GeneralPath thePath = new GeneralPath();
  thePath.moveTo(0.5f*ds,0.0f*ds);
  thePath.lineTo(0.0f*ds,0.5f*ds);
  thePath.lineTo(-0.5f*ds,0.0f*ds);
  thePath.lineTo(0.0f*ds,-0.5f*ds);
  thePath.closePath();

  //Now draw the diamond on the screen in black
  g2.draw(thePath);

  //Get a PathIterator object on the diamond
  PathIterator theIterator =
                        thePath.getPathIterator(null);
  //Use this array to store segment coordinate data
  float[] theData = new float[6];
  int theType;//store type of segment here

  //Get a new GeneralPath object.  Populate it based on
  // coordinates and segment types extracted from the
  // original GeneralPath object but offset the
  // coordinate values by one inch in both X and Y.
  GeneralPath newPath = new GeneralPath();

  //Iterate on the original GeneralPath object and
  // get information to populate the new
  // GeneralPath object.
  while(!theIterator.isDone()){//while not done
    //Get type of segment and coordinate values
    // for the current segment
    theType = theIterator.currentSegment(theData);

    //Process the current segment to populate a new
    // segment of the new GeneralPath object.
```

```
    switch(theType){
      case PathIterator.SEG_MOVETO :
        System.out.println("SEG_MOVETO");
        newPath.moveTo(theData[0]+1.0f*ds,
                       theData[1]+1.0f*ds);
        break;
      case PathIterator.SEG_LINETO :
        System.out.println("SEG_LINETO");
        newPath.lineTo(theData[0]+1.0f*ds,
                       theData[1]+1.0f*ds);
        break;
      case PathIterator.SEG_QUADTO :
        System.out.println("Not supported here");
        //Will illustrate SEG_QUADTO in later lesson
        break;
      case PathIterator.SEG_CUBICTO :
        System.out.println("Not supported here");
        //Will illustrate SEG_CUBICTO in later lesson
        break;
      case PathIterator.SEG_CLOSE :
        System.out.println("SEG_CLOSE");
        newPath.closePath();
        break;
    }//end switch

    //Get the next segment
    theIterator.next();
  }//end while loop

  g2.setColor(Color.red);
  g2.draw(newPath);

 }//end overridden paint()

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

Figure 17

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.

baldwin@austin.cc.tx.us

-end-