March 1, 2000

Java 2D Graphics, The Shape Interface, Part 1

Java Programming, Lecture Notes # 308

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 a subsequent lesson are intended to give you the necessary understanding of the **Shape** interface.

What is a Shape?

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?

A subsequent lesson will be dedicated entirely to the **PathIterator** interface, so I won't go into detail on **PathIterator** in this lesson. For the time being, 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.

Geometric Objects Implement Shape

A shape need not enclose an area

In his book, Java Foundation Classes in a Nutshell, David Flanagan tells us that the Java 2D definition of a shape does not require the shape to enclose an area. In other words, a **Shape** object can represent an open curve. According to Flanagan, if an open curve is passed to a method that requires a closed curve (such as **fill()**), the curve is automatically closed by connecting its end points with a straight line.

Graphics2D methods require Shape parameters

The following methods of the **Graphics2D** class require an object that implements the **Shape** interface as a parameter:

- draw()
- fill()
- clip()
- hit()

I have illustrated the use of the **draw()** method in previous, and will illustrate the use of the other methods listed above in subsequent lessons.

In addition, both the **Graphics2D** class and the **AffineTransform** class provide methods that allow us to *scale, rotate, translate,* and *shear* objects that implement the **Shape** interface. I have illustrated the use of these methods in previous lessons.

Shape is even used with text

The **Shape** interface even makes it way into the display of text, because the individual characters can be viewed as **Shape** objects.

Many geometric classes implement Shape

Java 2D provides a number of classes in the **java.awt.geom** package that implement the **Shape** interface, such as **Rectangle2D.Double** and **Ellipse2D.Double**. I have illustrated the use of some of these classes in previous lessons, and undoubtedly will continue to illustrate them in subsequent lessons. This lesson illustrates the use of the **Line2D.Double**, **Rectangle2D.Double** and **Ellipse2D.Double** classes.

According to Java 2D Graphics by Jonathan Knudsen,

> "Directly or indirectly, every geometric class in Java 2D implements the **Shape** interface. This means they can all be passed to **Graphic2D**'s **draw()** and **paint()** methods."

Knudsen also points out that every **Shape** object has an *interior* and an *exterior*, and that we can determine if a point or a rectangle is inside the **Shape** object.

He goes on to explain that what constitutes "inside" can be very complex for certain types of complex shapes. There are some special rules, called *winding rules* that are used to determine if a point is inside a **Shape** object. That is another topic that I will defer to a subsequent lesson.

The Area class combines Shape objects

Java 2D also provides an **Area** class that allows you to create new shapes by combining existing objects that implement the **Shape** interface. The **Area** class supports *union, intersection, subtraction,* and *exclusive OR* of **Shape** objects. I will defer a detailed discussion of those capabilities until a subsequent lesson.

GeneralPath class can create custom Shape objects

Another class, named **GeneralPath**, makes it possible for you to describe a **Shape** as a sequence of line segments and curves. I will also defer the discussion of that class until a subsequent lesson.

What Does This Lesson Cover?

So, having deferred several topics to subsequent lessons, what will I cover in this lesson?

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 will discuss the first three capabilities in this lesson, and defer a discussion of the **PathIterator** to a subsequent lesson.

Why should you care about the Shape interface?

You might be saying that you don't care about doing any of these four things, so why should you care about the **Shape** interface. Even if you don't care about bounding boxes, **PathIterator** objects, etc., you still need to know about the **Shape** interface.

As I explained earlier in this lesson, beginning with JDK 1.2, **Graphics2D** is the fundamental class for rendering two-dimensional shapes, text and images. Many of the capabilities introduced in the **Graphics2D** class are available only when working with objects that implement the **Shape** interface.

Also, as I explained in an earlier lesson, the **AffineTransform** class is used to transform graphic coordinate information between *user space* and *device space*. Similarly, most of the capabilities of the **AffineTransform** class are available only when working with objects that implement the **Shape** interface.

Shape interface is "new" to JDK 1.2

Flanagan tells us that even though the **Shape** interface was first defined in JDK 1.1,

> "The interface is so central to Java 2D and has grown so much since the Java 1.1 version, ... it should generally be considered to be new in Java 1.2."

Sample Program

This sample program, named **Shape03.java** illustrates the use of the **Shape** interface for

- Getting a bounding box that encloses a **Shape** object of the class **Ellipse2D.Double**.
- Determining if the **Shape** object completely contains a specified rectangle.
- Determining if the **Shape** intersects a specified rectangle

The **Shape** object, as well as the bounding rectangle, and three rectangles used in the tests are displayed in a **Frame** object in different colors.

Start with a Frame object

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.

After this, it draws a black circle with a diameter of one inch, centered on the new origin. Then it gets the bounding rectangle for the circle and draws it in red.

After this, it draws a green half-inch square completely inside the circle. Then it draws a blue half-inch square partially inside and partially outside the circle. Finally, it draws a magenta half-inch square completely outside the circle.

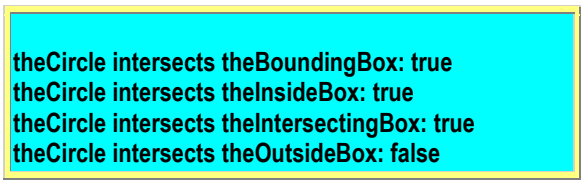
Test for contains() and intersects()

Then it tests the bounding rectangle and the three half-inch squares to determine if they are contained in the circle. It displays the results on the standard output device.

Finally, it tests the bounding rectangle and the three half-inch squares to determine if they intersect the circle, and displays the results on the standard output device.

In addition to drawing the circle and the squares in the **Frame**, the following output is presented on the command-line screen:

```
theCircle contains theBoundingBox: false
theCircle contains theInsideBox: true
theCircle contains theIntersectingBox: false
theCircle contains theOutsideBox: false
```



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

Will discuss in fragments

As is my practice, I will discuss the program in fragments. The entire program is presented intact at the end of the lesson for your review.

The controlling class and the constructor for the **GUI** class are almost identical to similar code that I have discussed in earlier lessons. Therefore, I won't discuss that code in this lesson. You might want to take a quick look at the complete listing at the end of the lesson. If you see something there that you don't understand, you should go back and review the previous lessons in this series on the Java 2D API.

Overridden paint() method

All of the interesting action in this program takes place in the overridden **paint()** method. Figure 1 shows the beginning of that method, including a reminder that it is necessary to downcast the incoming **Graphics** reference in order to gain access to the capabilities of the **Graphics2D** class.

```
public void paint(Graphics g){
  Graphics2D g2 = (Graphics2D)g;
```

Figure 1

Update the AffineTransform object

Figure 2 updates the **AffineTransform** object in the **Graphics2D** object to:

1. Compensate for the difference in actual screen resolution and assumed screen resolution of 72 units per inch.
2. Translate the origin to the center of the **Frame** object.

```
  g2.scale((double)res/72,(double)res/72);
  g2.translate((hSize/2)*ds,(vSize/2)*ds);
```

Figure 2

If you don't understand this fragment, you should go back and review an earlier lesson that discussed Affine transforms in detail.

Draw the axes

Figure 3 draws a horizontal line and a vertical line through the origin to represent the **X** and **Y**-axes in the 2D plane. This should be pretty intuitive. If it isn't clear what is happening here, you should take a look at the Sun documentation, particularly with respect to the parameters passed to the constructor for the **Line2D.Double** class.

```
//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 3

Instantiate a circle object

Figure 4 instantiates a circle object as an object of the **Ellipse2D.Double** class. Again, you might want to take a look at the Sun documentation, particularly with regard to the parameters being passed to the constructor.

```
Ellipse2D.Double theCircle = new
    Ellipse2D.Double(
            -0.5*ds, -0.5*ds, 1.0*ds, 1.0*ds);
```

Figure 4

When creating an object of this class, the parameters that are passed are actually the location, width, and height of a bounding rectangle for the ellipse. If the rectangle is a square (as in this case), the ellipse becomes a circle. If the rectangle is centered on the origin (as in this case), the circle is also centered on the origin.

Instantiates but doesn't render

Note that this fragment simply instantiates the object of the class **Ellipse2D.Double**. This fragment doesn't actually render that object onto the screen. This is an important point because we sometimes tend to overlook the difference between instantiating a **Shape** object and passing that object to a method of the **Graphics2D** class to actually render it on the output device.

Rendering the circle

Figure 5 passes the reference to the **Shape** (circle) object to the **draw()** method to have it rendered on the output device. Recall that the rendering process implements the current state of the **AffineTransform** object.

```
g2.draw(theCircle);
```

Figure 5

In this case, the transform provides scaling to convert from *user space* units to inches on the screen.

The transform also translates the origin to the center of the **Frame** object. Therefore, the circle is drawn centered in the **Frame** object with a diameter of one inch.

The circle is drawn in the default drawing color, which is black.

If you manually resize the **Frame**, the size and location of the circle doesn't change.

Get the bounding box

Figure 6 invokes the **getBounds2D()** method of the **Shape** interface to get the bounding rectangle for the circle. Since the **Shape** is a circle, the bounding rectangle is actually a square whose horizontal and vertical dimensions are the same the diameter of the circle.

```
Rectangle2D theBoundingBox =
                    theCircle.getBounds2D();
```

Figure 6

A reference to the bounding box is saved in the reference variable named **theBoundingBox** because it will be needed later.

Set the Color property

The **Graphics2D** object has several properties that are used to control certain aspects of the rendering process. One of those properties is **Color**. This property defines the actual drawing color that is used when the **draw()** method is invoked and passed a reference to a **Shape** object as a parameter.

Figure 7 uses a standard *setter* method to change the drawing color to red. Then it invokes the **draw()** method to draw **theBoundingBox** in red.

```
g2.setColor(Color.red);//change drawing color
g2.draw(theBoundingBox);
```

Figure 7

The test boxes

The next fragment instantiates three objects of the class **Rectangle2D.Double** that will be used later to illustrate the **contains()** method and the **intersects()** method of the **Shape** interface.

The contains() method

Here is part of what Sun has to say about the **contains()** method:

> "Tests if the interior of the **Shape** entirely contains the specified rectangular area. All coordinates that lie inside the rectangular area must lie within the **Shape** for the entire rectangular area to be considered contained within the **Shape**."

Sun goes on to explain some cases when this method might return false when the answer is really true due to arithmetic accuracy issues. If this is of interest to you, take a look at the Sun documentation.

The intersects() method

Here is part of what Sun has to say about the **intersects()** method:

> "Tests if the interior of the **Shape** intersects the interior of a specified rectangular area. The rectangular area is considered to intersect the **Shape** if any point is contained in both the interior of the **Shape** and the specified rectangular area."

Again, Sun provides some caveats having to do with arithmetic accuracy that may be of interest to you as well. If so, refer to the Sun documentation.

Instantiate the test boxes

Figure 8 instantiates three boxes. Given the coordinates passed as parameters to the constructor, one box is clearly contained in the circle. One box intersects the circle, and the third box is clearly outside the circle.

```
  Rectangle2D.Double theInsideBox = new
        Rectangle2D.Double(-0.25*ds, -0.25*ds,
                                   0.5*ds, 0.5*ds);
   Rectangle2D.Double theIntersectingBox = new
              Rectangle2D.Double(0.3*ds, 0.3*ds,
                                   0.5*ds, 0.5*ds);
   Rectangle2D.Double theOutsideBox = new
            Rectangle2D.Double(-1.25*ds, -1.25*ds,
```

```
                                        0.5*ds, 0.5*ds);
```

Figure 8

Draw the boxes in the Frame

Figure 9 draws the three boxes in colors of green, blue, and magenta.

```
g2.setColor(Color.green);
g2.draw(theInsideBox);//theInsideBox is green
g2.setColor(Color.blue);
g2.draw(theIntersectingBox);//blue
g2.setColor(Color.magenta);
g2.draw(theOutsideBox);//magenta
```

Figure 9

Perform the contains() test and display results

Figure 10 invokes the **contains()** method on **theCircle** to determine if **theBoundingBox** is contained in the circle. If so, the method returns **true**. Otherwise, the method returns **false**.

```
//Now perform the tests and display the results
// on the command-line screen.
System.out.println(
        "theCircle contains theBoundingBox: "
        + theCircle.contains(theBoundingBox));
```

Figure 10

The return value is formatted along with some explanatory text and displayed on the standard output device. In this case, the method returns **false** because **theBoundingBox** is not contained in **theCircle**.

I am going to skip over three other almost identical statements that test the other three boxes to see if they are contained in the circle. You can view these statements in the complete listing of the program at the end of the lesson.

Perform the intersects() test and display results

Figure 11 invokes the **intersects()** method on **theCircle** to determine if **theBoundingBox** intersects the circle.

```
System.out.println(
            "theCircle intersects theBoundingBox: "
    + theCircle.intersects(theBoundingBox));
```

Figure 11

In this case, the screen output was

```
theCircle intersects theBoundingBox: true
```

This means that the bounding box has some coordinate values in common with the circle. Because of the symmetry involved, it is probably reasonable to think that there are four points on the circle that are common with the bounding box.

Again, I am going to skip over three other almost identical statements that test the other three boxes to see if they intersect the circle. You can view those statements at the end of the lesson.

Summary

That brings us to the end of this lesson. We now know how to use three of the four primary capabilities of the **Shape** interface: **getBounds()**, **contains()**, and **intersects()**.

I will discuss the fourth primary capability, **getPathIterator()** in a subsequent lesson.

Complete Program Listing

A complete listing of the program is provided in Figure 12.

```
/*Shape03.java 12/12/99
 Copyright 1999, R.G.Baldwin

 Illustrates use of the Shape 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.

 Draws a black circle on the screen with a diameter of
 one inch centered at the origin.

 Gets the bounding box for the circle and draws it
 in red.

 Draws a one-half inch green square completely inside
 of the circle.

 Draws a one-half inch blue square partial inside and
 partially outside the circle.

 Draws a one-half magenta square completely outside the
 circle.

 Tests the bounding box and the three one-half inch
 squares to determine if they are contained in the circle.
 Displays the results on the command-line screen.
```

```
Tests the bounding box and the three one-half inch
squares to determine if they intersect the circle.
Displays the results on the command-line screen.

The program produces the following output:

theCircle contains theBoundingBox: false
theCircle contains theInsideBox: true
theCircle contains theIntersectingBox: false
theCircle contains theOutsideBox: false

theCircle intersects theBoundingBox: true
theCircle intersects theInsideBox: true
theCircle intersects theIntersectingBox: true
theCircle intersects theOutsideBox: false

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

class Shape03{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Shape03

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));

    //Define a one-inch diameter circle centered about
    // its origin.  Note that Ellipse2D implements Shape
    Ellipse2D.Double theCircle = new
```

```
      Ellipse2D.Double(-0.5*ds, -0.5*ds, 1.0*ds, 1.0*ds);

    //Draw theCircle in the Frame in the default
    // drawing color, black
    g2.draw(theCircle);

    //Get bounding box of theCircle
    Rectangle2D theBoundingBox =
                                    theCircle.getBounds2D();
    g2.setColor(Color.red);//change the drawing color
    //Draw the bounding box in the new color
    g2.draw(theBoundingBox);

    //Create boxes to test for contains and intersects
    Rectangle2D.Double theInsideBox = new
                  Rectangle2D.Double(-0.25*ds, -0.25*ds,
                                     0.5*ds, 0.5*ds);
    Rectangle2D.Double theIntersectingBox = new
                    Rectangle2D.Double(0.3*ds, 0.3*ds,
                                     0.5*ds, 0.5*ds);
    Rectangle2D.Double theOutsideBox = new
                  Rectangle2D.Double(-1.25*ds, -1.25*ds,
                                     0.5*ds, 0.5*ds);

    //Draw the test boxes in new colors
    g2.setColor(Color.green);
    g2.draw(theInsideBox);//theInsideBox is green
    g2.setColor(Color.blue);
    g2.draw(theIntersectingBox);//theIntersectingBox blue
    g2.setColor(Color.magenta);
    g2.draw(theOutsideBox);//theOutsideBox is magenta

    //Now perform the tests and display the results
    // on the command-line screen.
    System.out.println(
                  "theCircle contains theBoundingBox: "
                  + theCircle.contains(theBoundingBox));
    System.out.println("theCircle contains theInsideBox: "
                    + theCircle.contains(theInsideBox));
    System.out.println(
              "theCircle contains theIntersectingBox: "
              + theCircle.contains(theIntersectingBox));
    System.out.println("theCircle contains theOutsideBox: "
                     + theCircle.contains(theOutsideBox));
    System.out.println();//blank line
    System.out.println(
                "theCircle intersects theBoundingBox: "
                + theCircle.intersects(theBoundingBox));
    System.out.println(
                  "theCircle intersects theInsideBox: "
                  + theCircle.intersects(theInsideBox));
    System.out.println(
            "theCircle intersects theIntersectingBox: "
            + theCircle.intersects(theIntersectingBox));
    System.out.println(
                 "theCircle intersects theOutsideBox: "
                 + theCircle.intersects(theOutsideBox));

  }//end overridden paint()

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

Figure 12

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-