Java OOP: GradientPaint and other Java2D Classes*

Richard Baldwin

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License 3.0†

Abstract

Learn to use a variety of Java2D classes including GradientPaint.

1 Table of Contents

- Preface (p. 1)
  - Viewing tip (p. 1)
    - Figures (p. 2)
    - Listings (p. 2)
- Preview (p. 2)
- Discussion and sample code (p. 4)
- Run the program (p. 13)
- Summary (p. 13)
- What's next? (p. 13)
- Online video link (p. 14)
- Miscellaneous (p. 14)
- Complete program listing (p. 14)

2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [1] .

2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

*Version 1.3: Nov 14, 2012 12:18 pm +0000

†http://creativecommons.org/licenses/by/3.0/

[1] http://cnx.org/content/m44148/latest/

2.1.1 Figures

- Figure 1 (p. 3) . Required graphic output.
- Figure 2 (p. 4) . Required text output.
- Figure 3 (p. 8) . A drawing of an ellipse.

2.1.2 Listings

- Listing 1 (p. 4) . The driver class named Prob03.
- Listing 2 (p. 4) . Beginning of the class named Prob03Runner.
- Listing 3 (p. 5) . Beginning of the method named process.
- Listing 4 (p. 6) . Translate the origin to the center of the image.
- Listing 5 (p. 6) . Draw the black horizontal and vertical axes.
- Listing 6 (p. 9) . Draw the solid green filled ellipse in the upper-left quadrant.
- Listing 7 (p. 10) . Draw a circle with a gradient fill in the upper-right quadrant.
- Listing 8 (p. 12) . Code for the remaining two quadrants..
- Listing 9 (p. 14) . Complete program listing.

3 Preview

In this module you will learn to use the **GradientPaint** class along with a variety of other Java2D classes.

Program specifications

Write a program named **Prob03** that uses the class definition shown in Listing 1 (p. 4) and Ericson's media library along with the image file named **Prob03.jpg** to produce the graphic output image shown in Figure 1 (p. 3) . *(Note that the image in the file named Prob03.jpg is a blank white image. You could also create this blank image using one of the constructors for the* ***Picture*** *class.)*

Required graphic output.

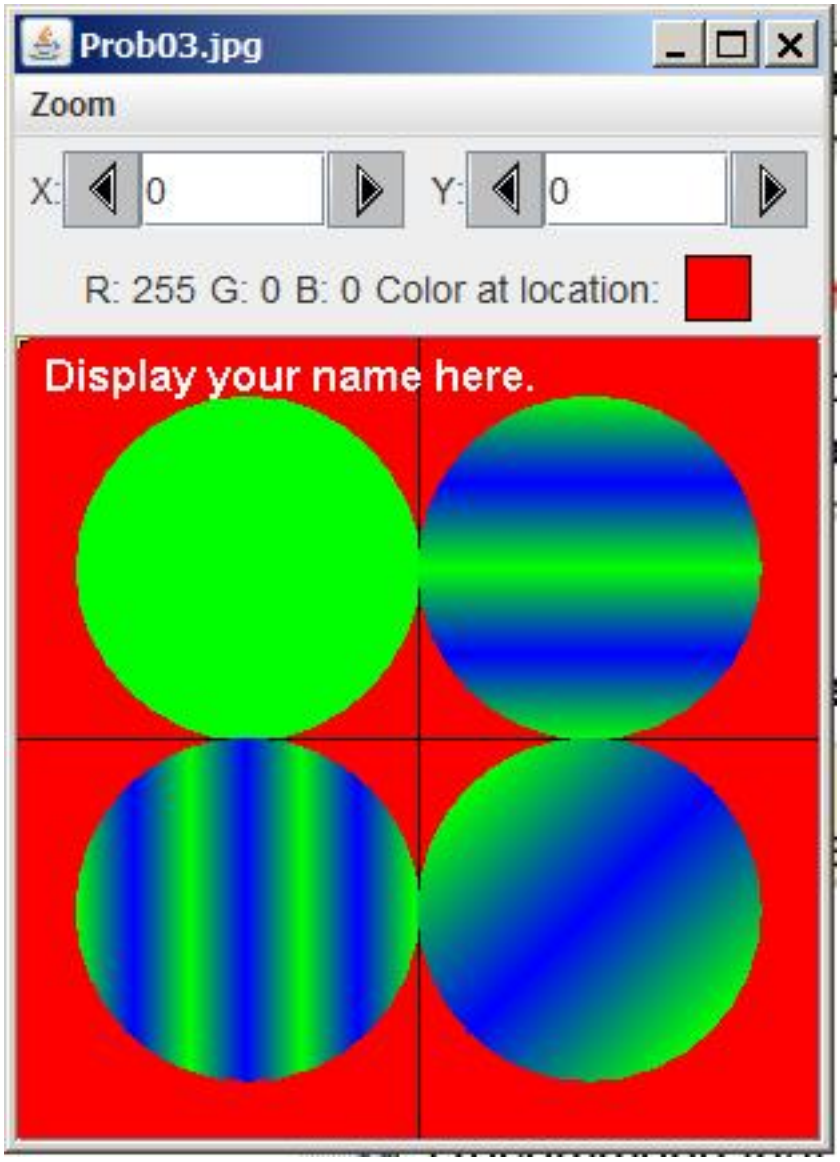

Figure 1: Required graphic output.

Circles with gradient paint

The program draws four circles in the quadrants of a Cartesian coordinate system. One is filled with solid green. The other three are filled with cyclic gradient paint from green to blue.

The number of cycles varies in each circle, as do the axes along which the gradient occurs.

The background is set to Color.RED.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob03** shown in Listing 1 (p. 4) .

Required text output

In addition to the output image mentioned above, your program must display your name and the other line of text shown in Figure 2 (p. 4) .

Required text output.

```
    Display your name here.
Picture, filename Prob03.jpg height 300 width 300
```

Figure 2: Required text output.

---

4 Discussion and sample code

Will discuss in fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 9 (p. 14) near the end of the module.

The driver class named Prob03

The driver class containing the main method is shown in Listing 1 (p. 4) .

Listing 1: The driver class named Prob03.

```
    public class Prob03{
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03
```

If you have been studying the earlier modules in this series, no explanation of Listing 1 (p. 4) should be required.

Beginning of the class named Prob03Runner

The class named **Prob03Runner** begins in Listing 2 (p. 4) .

Listing 2: Beginning of the class named Prob03Runner.

```
    class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//
```

```
  public void run(){
    Picture pic = new Picture("Prob03.jpg");
//  Picture pic = new Picture(300,300);

    pic.setAllPixelsToAColor(Color.RED);

    process(pic);

    //Add your name and display the output picture.
    pic.addMessage("Display your name here.",10,20);
    pic.explore();
    System.out.println(pic);
  }//end run
```

Avoiding use of a blank input image file

This program was originally written using an early version of Ericson's class library that didn't support the second statement in the **run** method. *(That statement was disabled by turning it into a comment in Listing 2* (p. 4) *)* As a result, with that library, it was necessary to read an image file containing a blank white image to create a **Picture** object with a blank white image.

No longer a problem

That problem was rectified with an update to her library and the disabled statement can be substituted for the statement immediately above it. If you do that, you won't need the input image file.

Except for that, and the call to the method named **process** , you should already understand all of the code in Listing 2 (p. 4) .

Beginning of the method named process

The method named **process** begins in Listing 3 (p. 5) .

Listing 3: Beginning of the method named process.

```
    private void process(Picture pic){

    Graphics2D g2 = (Graphics2D)(pic.getGraphics());

    int width = pic.getWidth();
    int height = pic.getHeight();
```

Java2D graphics

This is basically a module on the use of classes from the Java2D section of the standard class library. *(See my lessons 300 through 324 on Java2D graphics here* [2] *.)* Classes from Ericson's library are used mainly to support the display aspects of the program.

What are Java2D graphics?

Although the capabilities provided by Java2D graphics are wide and varied, in one way or another, they generally have to do with the creation of images by drawing.

A graphics context is required

In order to use the methods that will be using, it is necessary to gain access to the graphics context of an object as type **Graphics2D** . The first statement in Listing 3 (p. 5) calls Ericson's **getGraphics** method to gain access to the graphics context of a **Picture** object.

A cast to type Graphics2D is required

[2] http://www.dickbaldwin.com/tocadv.htm

However, the **getGraphics** method returns a reference to the graphics context as type **Graphics** . In order for us to use it to do what we want to do, we must cast it to type **Graphics2D** . This gives us access to many more methods that would be the case without the cast.

Save as type Graphics2D in a variable named g2

The graphics context for the **Picture** object is saved in Listing 3 (p. 5) as type **Graphics2D** . The reference is saved in the reference variable named **g2** .

Save the width and height of the Picture object

The last two statements in Listing 3 (p. 5) get and save the **width** and the **height** of the image encapsulated in the **Picture** object.

Translate the origin to the center of the image

By default, the origin *(with coordinates of 0,0)* is in the upper-left corner of the image. However, we would like to be able to work with a coordinate system in which the origin is at the center.

Listing 4 (p. 6) calls the **translate** method to move the origin to the center of the image.

Listing 4: Translate the origin to the center of the image.

```
g2.translate(width/2,height/2);
```

(Fortunately, you already understand affine transforms. Otherwise, you might not be able to understand the documentation for the ***translate*** *method.)*

From this point forward...

From this point forward, we can think of the coordinates of the pixel at the very center of the object as having values of 0,0. Locations to the left of center have negative X coordinates and locations above the center have negative Y coordinates.

Draw the black horizontal and vertical axes

The next thing we want to do is to draw the black horizontal and vertical axes that you see in the center of the image in Figure 1 (p. 3) . This is accomplished by the code in Listing 5 (p. 6) .

Listing 5: Draw the black horizontal and vertical axes.

```
    //Set the drawing color to black
g2.setColor(Color.BLACK);

//Draw x-axis
g2.draw(new Line2D.Double(-width/2, 0.0,
                          width/2, 0.0));

//Draw y-axis
g2.draw(new Line2D.Double(0.0, -width/2,
                         0.0,  height/2));
```

Set the drawing color to black

Listing 5 (p. 6) begins by calling the **setColor** method to set the drawing color to Color.BLACK.

(BLACK is a static constant in the ***Color*** *class that represents the color black.)*

A new Line2D.Double object

The fourth line of code in Listing 5 (p. 6) instantiates a new object of the **Line2D.Double** class. This object represents a line extending between two points specified by coordinate values passed as parameters to the constructor.

A black horizontal line

The first pair of coordinate values specifies the left end of the black horizontal line in Figure 1 (p. 3) . The second pair of coordinate values specifies the right end of the black horizontal line in Figure 1 (p. 3) .

(See my Lesson Number 300 [3] *for an explanation of the somewhat unusual name of a class consisting of two words separated by a period:* ***Line2D.Double*** *.)*

Pass the line object to the Draw method

The new object's reference is passed to the **draw** method, which is responsible for causing the line to be drawn on the graphics context.

A black vertical axis

The last statement in Listing 5 (p. 6) draws the black vertical line shown in Figure 1 (p. 3) .

Coordinates relative to the origin at the center

Note that in both cases, the end points of the line are specified using coordinate values that are relative to the origin, which is positioned at the center of the drawing context.

Draw the solid green filled ellipse in the upper-left quadrant

In case you are unfamiliar with the term, an *ellipse* is the 2D shape shown in the upper-left quadrant of Figure 3 (p. 8) . A circle is an ellipse with the major and minor axes having the same lengths.

[3] http://www.dickbaldwin.com/java/Java300.htm

A drawing of an ellipse.

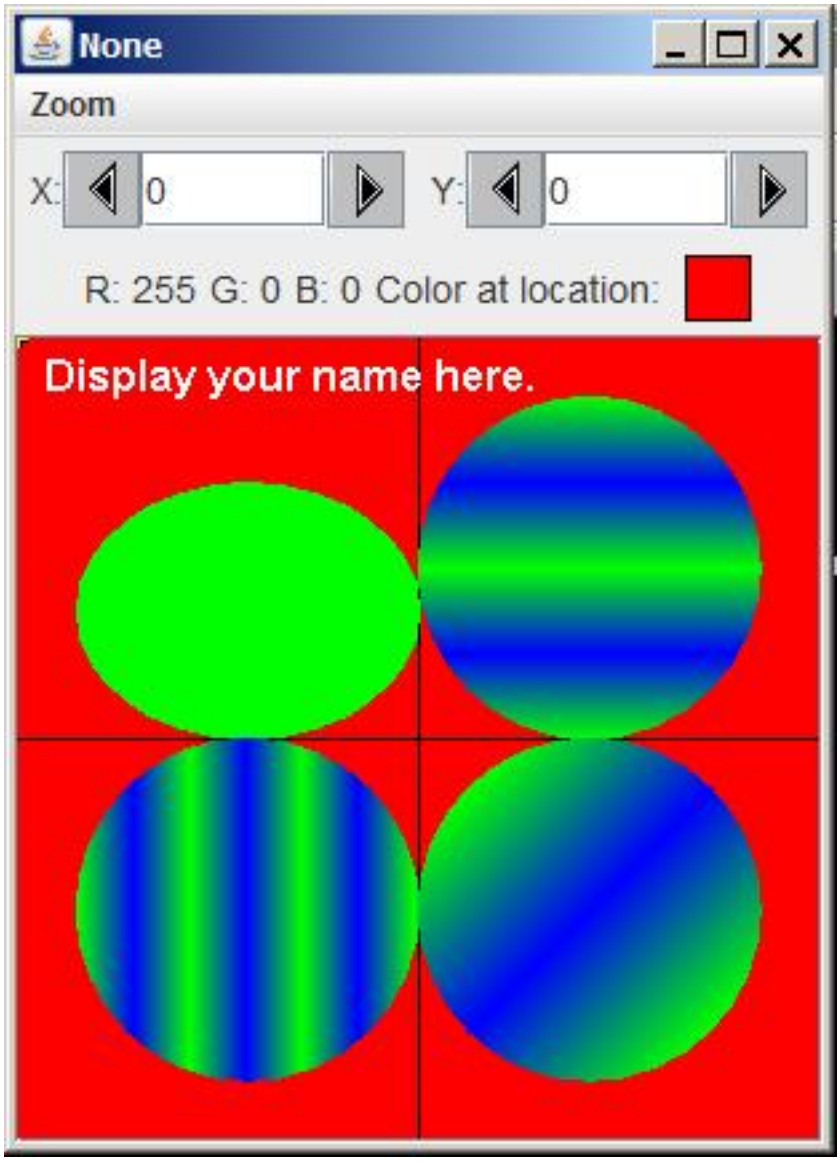

Figure 3: A drawing of an ellipse.

There is no circle class

Java does not provide a **Circle2D.Double** class but it does provide an **Ellipse2D.Double** class, which you can use to draw circles.

*(The **Graphics** class also provides a **drawOval** method that can be used to draw circles but those circles won't suffice for what we will be doing in this module.)*

Four constructor parameters

The constructor for the **Ellipse2D.Double** class requires four parameters of type **double** . The first two parameters are the X and Y coordinates of the upper-left corner of an imaginary rectangle.

The next two parameters are the **width** and the **height** of the imaginary rectangle.

The sides of the imaginary rectangle are parallel to the X and Y axes, but can be rotated using affine transforms.

An ellipse inside an imaginary rectangle

The ellipse is constructed inside the imaginary rectangle such that it is symmetrical about its horizontal and vertical axes and it touches all four sides of the rectangle. *(The documentation refers to the rectangle as a framing rectangle.)*

How do you construct a circle?

If the rectangle is actually a square, then the ellipse becomes a circle.

Construct an ellipse inside an imaginary square

The first statement in Listing 6 (p. 9) constructs an object of type **Ellipse2D.Double** inside a 128x128 square that just fits in the upper left quadrant of our Cartesian coordinate system shown in Figure 1 (p. 3) . The circle object's reference is saved in the reference variable named **circle1** .

Listing 6: Draw the solid green filled ellipse in the upper-left quadrant.

```
    //Upper left quadrant
Ellipse2D.Double circle1 =
         new Ellipse2D.Double(-128,-128,128,128);

//Solid GREEN fill
g2.setPaint(Color.GREEN);

g2.fill(circle1);

g2.draw(circle1);
```

Set the painting color

The statement near the middle of Listing 6 (p. 9) sets the painting color to **Color.GREEN** .

*(Note that Listing 6 (p. 9) calls **setPaint** whereas Listing 5 (p. 6) calls **setColor** . I will leave it as an exercise for the student to study the documentation in order to understand the difference between **setColor** and **setPaint** .)*

Fill the circle referred to by circle1

The second statement from the bottom in Listing 6 (p. 9) calls the **fill** method of the **Graphics2D** class passing the circle object's reference as a parameter. Although this can get quite complex, in this simple case, it causes the circle object to be filled with the paint color *(green)* .

Draw the filled circle object

Finally, the last statement in Listing 6 (p. 9) calls the **draw** method of the **Graphics2D** class to cause the filled circle to be drawn inside the framing rectangle that was specified when the circle was constructed. This results in the filled green circle in the upper-left quadrant in Figure 1 (p. 3) .

Draw a circle with a gradient fill in the upper-right quadrant

This is where things tend to get a little complicated.

If you compare the circles in the upper-left and upper-right quadrants in Figure 1 (p. 3) , you will see that they look considerably different.

Difference caused by parameter to the setPaint method

If you compare the code in Listing 6 (p. 9) with the code in Listing 7 (p. 10) , you will see that except for the coordinate values that position the circle, the only difference is the parameter that is passed to the **setPaint** method.

Listing 7: Draw a circle with a gradient fill in the upper-right quadrant.

```
    //Upper right quadrant
//Gradient GREEN to BLUE, cyclic along horizontal
// axis.
Ellipse2D.Double circle2 =
   new Ellipse2D.Double(0.0,-128,128,128);
g2.setPaint(new GradientPaint(
                64,0,Color.GREEN,
                64,-32,Color.BLUE,true));
g2.fill(circle2);
g2.draw(circle2);
```

A simple color paint versus a gradient paint

Listing 6 (p. 9) passes an object of the simple **Color** class representing the color green to the **setPaint** method.

Listing 7 (p. 10) passes an object of the **GradientPaint** class to the **setPaint** method.

Therefore, we need to understand the behavior of an object of the **GradientPaint** class.

Constructor parameters

There are four overloaded constructors for the **GradientPaint** class. The constructor used in Listing 7 is one of the most complicated. It requires the following parameters:

- x1 - x coordinate of the first specified Point in user space
- y1 - y coordinate of the first specified Point in user space
- color1 - Color at the first specified Point
- x2 - x coordinate of the second specified Point in user space
- y2 - y coordinate of the second specified Point in user space
- color2 - Color at the second specified Point
- cyclic - true if the gradient pattern should cycle repeatedly between the two colors; false otherwise

Two points and two colors

Basically, the class allows you to specify two points and two colors *(plus one additional boolean parameter)* when you construct an object of the class.

One of the colors is associated with each point.

An imaginary line segment

Think of the two points as being at the ends of an imaginary line segment, which can be at any angle relative to the horizontal.

Colors at the ends of the line segment

When a shape is drawn using a gradient fill, one of the colors will appear at one end of the line segment and the other color will appear at the other end of the line segment.

The color will change

The color will change gradually from one color to the other along the imaginary line segment connecting the two points.

Perpendicular color bands on the sides

The colors extend out to the sides of the imaginary line segment in bands that are perpendicular to the line segment.

The cyclic parameter

If the last *(cyclic)* parameter is set to false, the color will only change along the imaginary line segment. Areas beyond each end of the line segment will be the colors that are specified for the points at the ends of the line segment.

(This parameter was not set to false for any of the circles in Figure 1.)

If the cyclic parameter is true...

The pattern of color change that occurs along the line segment will extend in a cyclic fashion beyond the ends of the line segment all the way to infinity.

Compare constructor parameters with upper-right quadrant

Now consider the parameter values used in Listing 7 (p. 10) and compare them with the image in the upper-right quadrant in

A simple color paint versus a gradient paint

Listing 6 passes an object of the simple **Color** class representing the color green to the **setPaint** method.

Listing 7 passes an object of the **GradientPaint** class to the **setPaint** method.

Therefore, we need to understand the behavior of an object of the **GradientPaint** class.

Constructor parameters

There are four overloaded constructors for the **GradientPaint** class. The constructor highlighted in yellow in Listing 7 is one of the most complicated. It requires the following parameters:

- x1 - x coordinate of the first specified Point in user space
- y1 - y coordinate of the first specified Point in user space
- color1 - Color at the first specified Point
- x2 - x coordinate of the second specified Point in user space
- y2 - y coordinate of the second specified Point in user space
- color2 - Color at the second specified Point
- cyclic - true if the gradient pattern should cycle repeatedly between the two colors; false otherwise

Two points and two colors

Basically, the class allows you to specify two points and two colors *(plus one additional boolean parameter)* when you construct an object of the class.

One of the colors is associated with each point.

An imaginary line segment

Think of the two points as being at the ends of an imaginary line segment, which can be at any angle relative to the horizontal.

Colors at the ends of the line segment

When a shape is drawn using a gradient fill, one of the colors will appear at one end of the line segment and the other color will appear at the other end of the line segment.

The color will change

The color will change from one color to the other along the imaginary line segment connecting the two points.

Perpendicular color bands on the sides

The colors extend out to the sides of the imaginary line segment in bands that are perpendicular to the line segment.

The cyclic parameter

If the last *(cyclic)* parameter is set to false, the color will only change along the imaginary line segment. Areas beyond each end of the line segment will be the colors that are specified for the points at the ends of the line segment.

(This parameter was not set to false for any of the circles in Figure 1.)

If the cyclic parameter is true...

The pattern of color change that occurs along the line segment will extend in a cyclic fashion beyond the ends of the line segment all the way to infinity.

Compare constructor parameters with upper-right quadrant

Now consider the parameter values used in Listing 7 (p. 10) and compare them with the image in the upper-right quadrant in Figure 1 (p. 3) .

The location of the first point

The diameter of each circle in Figure 1 (p. 3) is 128 pixels. The first point for the upper-right quadrant in Figure 1 (p. 3) is on the X axis at the point where the circle touches the X axis. This point is specified to have a color of green.

The location of the second point

The second point is 32 pixels directly above the first point. Therefore, the imaginary line segment is perpendicular to the X axis. It extends from the X axis one-fourth of the way to the top of the circle. The second point is specified to have a color of blue.

The color change

As a result, the color changes from green to blue along the 32-pixel line segment. Color bands extend to the right and left, perpendicular to the line segment.

The cyclic parameter

The cyclic parameter is set to true, so the pattern repeats along the remaining vertical dimension of the circle. The color changes from blue back to green along the next 32-pixel vertical distance causing the circle to be green at the center.

The pattern repeats again resulting in a blue band three-fourths of the way from the bottom to the top of the circle and a green band at the top of the circle.

That's all there is to it

Now that you know the scheme, you should be able to examine the code in Listing 8 (p. 12) for the remaining two quadrants and understand how the constructor parameters resulted in the color patterns that you see in Figure 1 (p. 3) .

Listing 8: Code for the remaining two quadrants.

```
        //Lower left quadrant
    //Gradient GREEN to BLUE, cyclic along vertical axis.
    //horizontal axis
    Ellipse2D.Double circle3 =
       new Ellipse2D.Double(-128,0.0,128,128);
    g2.setPaint(
            new GradientPaint(
                    -128,-64,Color.GREEN,
                    -107,-64,Color.BLUE,true));
    g2.fill(circle3);
    g2.draw(circle3);

    //Lower right quadrant
    //Gradient GREEN to BLUE, cyclic along
    // 45 degree angle
    Ellipse2D.Double circle4 =
        new Ellipse2D.Double(0,0,128,128);
    g2.setPaint(
            new GradientPaint(
                    19,19,Color.GREEN,
                    64,64,Color.BLUE,true));

    g2.fill(circle4);
    g2.draw(circle4);

  }//end process

}//end class Prob03Runner
```

An interesting anomaly

I will point out one interesting anomaly, however. Even though we shifted the origin to the center, we did not change the direction that represents positive values on the Y axis.

Coordinates above the center are negative and coordinates below the center are positive.

Where is the line segment?

If you examine the constructor parameters for the lower-left quadrant in Listing 8 (p. 12) , you will see that the imaginary line segment isn't in the lower-left quadrant.

The line segment is 21 pixels in length, parallel to the horizontal axis. However, it is 64 pixels above the horizontal axis. That is not in the lower-left quadrant.

The effect is...

The effect is as though the color gradient fills the universe, and is based on a line segment placed anywhere in the universe.

However, we only see the color gradient through the shape that we specify as a parameter to the **fill** method. In other words, that shape is a window through which we can see a background consisting of gradient color changes.

Our window is in the lower-left quadrant

In this case, that shape is the circle in the lower-left quadrant, so we see the gradient color effect in the lower-left quadrant.

How about the bottom-right quadrant

I stated earlier that the imaginary line segment can be at any angle relative to the horizontal axis.

For the bottom right quadrant, the line segment is 64 pixels in length. It lies along a line that goes through the origin and is at 45 degrees clockwise relative to the horizontal.

Where is the line segment positioned?

One end of the line segment is at the center of the circle. The other end of the line segment is at the point where the 45-degree line intersects the edge of the circle closest to the origin.

You can take it from there

You should be able to take it from there and explain the color gradient in the circle in the lower-right quadrant. Recall that the diameter of the circle is 128 pixels. The length of the line segment is 64 pixels or one-half the diameter.

The end of the class

Listing 8 (p. 12) signals the end of the class named **Prob03Runner** and the end of the program.

5 Run the program

I encourage you to copy the code from Listing 9 (p. 14) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. For example, try setting the cyclic constructor parameter to false and observe the effect. Make certain that you can explain why your changes behave as they do.

Click Prob03.jpg [4] to download the input image file if you elect to use it.

6 Summary

In this module, you learned to use the **GradientPaint** class along with a variety of other Java2D classes.

7 What's next?

In the next module, you will Learn how to use shapes to clip images during the drawing process.

[4] http://cnx.org/content/m44242/1.3/Prob03.jpg

8 Online video link

Select the following link to view an online video lecture on the material in this module.

- ITSE 2321 Lecture 13 [5]

9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: GradientPaint and other Java2D Classes
- File: Java3026.htm
- Published: August 4, 2012
- Revised: November 14, 2012

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

10 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 9 (p. 14) below.

Listing 9: Complete program listing.

```
    /*File Prob03 Copyright 2008 R.G.Baldwin
*******************************************************/
import java.awt.geom.Line2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GradientPaint;

public class Prob03{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
```

[5] http://vimeo.com/channels/itse2321/21220418

```
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03
//=====================================================//

class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//
  public void run(){
    Picture pic = new Picture("Prob03.jpg");
    pic.setAllPixelsToAColor(Color.RED);

    process(pic);
    //Add your name and display the output picture.
    pic.addMessage("Display your name here.",10,20);
    pic.explore();
    System.out.println(pic);
  }//end run
  //-----------------------------------------------------//

  private void process(Picture pic){

    Graphics2D g2 = (Graphics2D)(pic.getGraphics());

    int width = pic.getWidth();
    int height = pic.getHeight();

    //Translate origin to center of Frame
    g2.translate(width/2,height/2);
    g2.setColor(Color.BLACK);
    //Draw x-axis
    g2.draw(new Line2D.Double(-width/2,0.0,width/2,0.0));
    //Draw y-axis
    g2.draw(new Line2D.Double(0.0,-width/2,0.0,height/2));

    //Upper left quadrant, Solid GREEN fill
    Ellipse2D.Double circle1 =
      new Ellipse2D.Double(-128,-128,128,128);
    g2.setPaint(Color.GREEN);
    g2.fill(circle1);
    g2.draw(circle1);

    //Upper right quadrant
    //Gradient GREEN to BLUE, cyclic along horizontal
    // axis.
    Ellipse2D.Double circle2 =
       new Ellipse2D.Double(0.0,-128,128,128);
    g2.setPaint(new GradientPaint(
```

```
                  64,0,Color.GREEN,
                  64,-32,Color.BLUE,true));
    g2.fill(circle2);
    g2.draw(circle2);


    //Lower left quadrant
    //Gradient GREEN to BLUE, cyclic along vertical axis.
    //horizontal axis
    Ellipse2D.Double circle3 =
       new Ellipse2D.Double(-128,0.0,128,128);
    g2.setPaint(
            new GradientPaint(
                    -128,-64,Color.GREEN,
                    -107,-64,Color.BLUE,true));
    g2.fill(circle3);
    g2.draw(circle3);

    //Lower right quadrant
    //Gradient GREEN to BLUE, cyclic along
    // 45 degree angle
    Ellipse2D.Double circle4 =
        new Ellipse2D.Double(0,0,128,128);
    g2.setPaint(
            new GradientPaint(
                    19,19,Color.GREEN,
                    64,64,Color.BLUE,true));

    g2.fill(circle4);
    g2.draw(circle4);

  }//end process

}//end class Prob03Runner
```

-end-