Java 2D Graphics, The Composite Interface and Transparency

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

Java Programming, Lecture Notes # 320

March 20, 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.

Must also understand other classes

I also explained that without understanding the behavior of other classes and interfaces, it is not possible to fully understand the inner workings of the **Graphics2D** class.

Throughout this series of lessons, I have been providing you with information and sample programs designed to help you understand the various classes and interfaces that are necessary for an understanding of the **Graphics2D** class.

Two ways to achieve transpanency

There are at least two different ways to achieve transparency in Java 2D. One way is to use new constructors for the **Color** class that allow you to create solid colors with a specified degree of transparency. I will discuss that approach in a subsequent lesson.

A more generl approach

A second, and possibly more general approach, is to make use of an object that implement the **Composite** interface, passing a reference to that object to the **setComposite()** method of the **Graphics2D** class.

The Composite interface

This lesson and the next are designed to give you an understanding of the **Composite** interface, with particular emphasis on transparency.

What is the CompositeContext Interface?

I mention the **CompositeContext** interface here only because it is referred to in the following discussion of the **Composite** interface.

According to Sun,

"The **CompositeContext** interface defines the encapsulated and optimized environment for a compositing operation. **CompositeContext** objects maintain state for compositing operations. In a multi-threaded environment, several contexts can exist simultaneously for a single **Composite** object."

What is the Composite Interface?

Our primary objective in this lesson is to develop an understanding of how to use objects of the **Composite** interface as parameters to the **setComposite**() method of the **Graphics2D** class. This is part of what Sun has to say about the **Composite** interface.

"The **Composite** interface, along with **CompositeContext**, defines the methods to compose a draw primitive with the underlying graphics area.

After the **Composite** is set in the **Graphics2D** context, it combines a shape, text, or an image being rendered with the colors that have already been rendered according to pre-defined rules."

We will set the composite property

In other words, before drawing onto a **Graphics2D** object, we will invoke the **setComposite()** method to set the **composite** property of the **Graphics2D** object.

We will pass a reference to an object that implements the **Composite** interface as a parameter.

This **Composite** object will control the way in which the colors of overlapping pixels are rendered.

What is the AlphaComposite Class?

There is only one class in JDK 1.2.2 that implements the **Composite** interface. That class is named **AlphaComposite**. Here is what Sun has to say about this class.

"This **AlphaComposite** class implements the basic alpha compositing rules for combining source and destination pixels to achieve blending and transparency effects with graphics and images.

The rules implemented by this class are a subset of the Porter-Duff rules described in T. Porter and T. Duff, "Compositing Digital Images", SIGGRAPH 84, 253-259.

If any input does not have an alpha channel, an alpha value of 1.0, which is completely opaque, is assumed for all pixels. A constant alpha value can also be specified to be multiplied with the alpha value of the source pixels.

The following abbreviations are used in the description of the rules:

- Cs = one of the color components of the source pixel.
- Cd = one of the color components of the destination pixel.
- As = alpha component of the source pixel.

- Ad = alpha component of the destination pixel.
- Fs = fraction of the source pixel that contributes to the output.
- Fd = fraction of the input destination pixel that contributes to the output.

The color and alpha components produced by the compositing operation are calculated as follows:

$$Cd = Cs*Fs + Cd*Fd$$

$$Ad = As*Fs + Ad*Fd$$

where Fs and Fd are specified by each rule. The above equations assume that both source and destination pixels have the color components premultiplied by the alpha component. Similarly, the equations expressed in the definitions of compositing rules below assume premultiplied alpha.

For performance reasons, it is preferable that Rasters passed to the compose method of a CompositeContext object created by the AlphaComposite class have premultiplied data. If either source or destination Rasters are not premultiplied, however, appropriate conversions are performed before and after the compositing operation.

The alpha resulting from the compositing operation is stored in the destination if the destination has an alpha channel. Otherwise, the resulting color is divided by the resulting alpha before being stored in the destination and the alpha is discarded. If the alpha value is 0.0, the color values are set to 0.0."

A fairly complex topic

This can be a fairly complex topic. An object of the **AlphaComposite** class can be used to implement any one of about eight different compositing rules.

As you can see from the above description, the manner in which the color components of the destination pixel are determined depend on the rule being applied (a single **AlphaComposite** object can apply only one rule).

Flanagan explains the rules

You can read about the different rules in Java Foundation Classes in a Nutshell, by David Flanagan.

In these lessons, I will illustrate only one of the compositing rules: the rule known as **SRC_OVER**. Here is what Flanagan has to say about this rule.

"By far the most commonly used compositing rule. It draws the source on top of the destination. The source and destination are combined based on the transparency of the source. Where the source is opaque, it replaces the destination. Where the source is transparent, the destination is unchanged. Where the source is translucent, the source and destination colors are combined so that some of the destination color shows through the translucent source."

SRC_OVER as per Sun

Here is Sun's formal definition of this rule from the JDK 1.2.2 documentation.

SRC_OVER

public static final int SRC_OVER

Porter-Duff Source Over Destination rule. The source is composited over the destination.

```
Fs = 1 and Fd = (1-As), thus:

        Cd = Cs + Cd*(1-As)

        Ad = As + Ad*(1-As)
```

SRC_OVER as per Flanagan

However, Sun's explanation doesn't agree with Flanagan's explanation. According to Flanagan, the equations for this rule should be written as follows:

```
Fs = As and Fd = (1-As), thus:

        Cd = Cs*As + Cd*(1-As)
```

In these lessons, the thing that will be most obvious will be the color resulting from overlapping two or more geometric figures with varying degrees of transparency.

Which is correct?

Although I'm not qualified to tell you which of the two explanations is the correct one, experimental results seem to favor Flanagan's explanation over Sun's explanation.

If As is set to zero...

For example, if **As** is set to zero, Sun's explanation would cause the destination color, **Cd**, to contain equal contributions of the source color and the destination color.

Flanagan's explanation for this case would eliminate all of the source color from the final destination color, which agrees with experimental results.

May be a matter of interpretation

However, these apparent discrepancies may simply be a matter of interpretation of the various terms used in the equations and the explanations.

How Do I Get an AlphaComposite Object?

You cannot directly instantiate an object of the **AlphaComposite** class. Rather, you get an **AlphaComposite** object by invoking the following factory method of the **AlphaComposite** class.

```
public static AlphaComposite
```

getInstance(int rule, float alpha)
Creates an **AlphaComposite** object with the specified rule and the constant alpha to multiply with the alpha of the source. The source is multiplied with the specified alpha before being composited with the destination.

Parameters:

- rule - the compositing rule
- alpha - the constant alpha to be multiplied with the alpha of the source. alpha must be a floating point number in the inclusive range [0.0, 1.0].

How do I specify the rule?

You specify the rule by passing an **int** value given by one of the symbolic constants of the **AlphaComposite** class, such as **SRC_OVER** described earlier.

What is the setComposite() Method?

We need to look at one more definition before embarking on our sample program. Here is part of what Sun has to say about the **setComposite()** method of the **Graphics2D** class.

public abstract void
setComposite(Composite comp)

Sets the **Composite** for the **Graphics2D** context. The **Composite** is used in all drawing methods such as drawImage, drawString, draw, and fill. It specifies how new pixels are to be combined with the existing pixels on the graphics device during the rendering process.

Parameters:

- comp - the **Composite** object to be used for rendering

You could define your own class

The required parameter is a reference to any object that implements the **Composite** interface, meaning that you could define your own class to implement this interface.

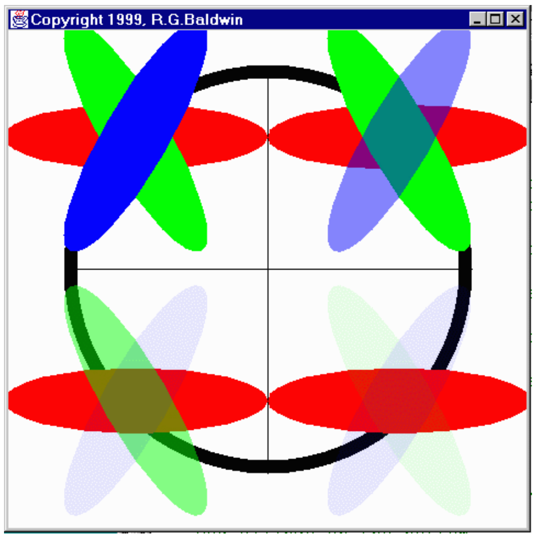
In this lesson, I elected to make use of the existing **AlphaComposite** class described above.

Sample Program

This program is named **Composite01**. You will need to compile and execute the program so that you can view its output while reading the discussion. Without being able to view the output, the discussion will probably mean very little to you.

A screen shot of the output

In case you are unable to compile and execute the program, a screen shot of the output follows. Note, however that this screen shot has been reduced to about 70-percent of its original size in pixels, so some of the quality has been lost in the process.



The GUI is 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.

A large circle

After drawing the X and Y-axes, the program draws a circle with a thick border centered on the origin. This circle is used later to provide visual cues relative to transparency.

Transparent ellipses

After the large circle is drawn, three ellipses are drawn on top of one another in each quadrant.

Each ellipse has a common center, and is rotated by sixty degrees relative to the ellipse beneath it.

The red ellipse is on the bottom of the stack, the green ellipse is in the center, and the blue ellipse is on the top of the stack.

The different ellipses are given various transparency values in the different quadrants to illustrate the effect of the alpha parameter to the **setComposite()** method.

Transparency by quadrant

Here is the transparency given to each of the ellipses in the different quadrants.

```
TRANSPARENCY
Upper-left quadrant
No transparency

Upper-right quadrant
Red is not transparent
Green is not transparent
Blue is 50-percent transparent

Lower-left quadrant
Red is not transparent
Green is 50-percent transparent
Blue is 90-percent transparent

Lower-right quadrant
Red is not transparent
Green is 90-percent transparent
Blue is 90-percent transparent
```

As you can see from the information given above, the red ellipse is opaque in all four quadrants. As a result, the large black circle doesn’t show through the red ellipse in any of the quadrants.

Upper-left quadrant

All three ellipses are opaque in the upper-left quadrant, so nothing shows through, and the stacking order of the ellipses is pretty obvious.

Green and blue ellipses become transparent

The green and blue ellipses are made progressively more transparent as you move through the other three quadrants. As a result, you can "see through" the green and blue ellipses and see the geometric figures that lie beneath them (the other ellipses and the large black circle).

Upper-right quadrant

In this quadrant, the blue ellipse is transparent, but the green ellipse and the red ellipse are opaque. Neither the large circle nor the red ellipse can be seen through the green ellipse.

However, both the green and red ellipses show through the blue ellipse in the upper-right quadrant.

Lower-left quadrant

In the lower-left quadrant, both the green and blue ellipses are transparent to some degree, with the blue ellipse being the more transparent of the two.

Both the green and the red ellipses show through the blue ellipse, which is on the top of the stack.

Both the red ellipse and the large circle show through the green ellipse.

The large circle would also show through the blue ellipse as well except that the red ellipse, which is opaque, hides the circle in the area of the blue ellipse.

Lower-right quadrant

In the lower-right quadrant, both the green and blue ellipses are ninety-percent transparent, and the large circle shows through the blue ellipse. Again, the red ellipse, which is opaque, hides the circle.

The lower-right quadrant also produces an optical illusion. On my screen, it looks like the red ellipse is partially transparent with the green and blue ellipses showing through the red ellipse. However, that is not the case. The red ellipse is opaque in this quadrant, as evidenced by the fact that the large circle does not show through the red ellipse. The opaque red ellipse is still on the bottom of the stack, and the transparent blue ellipse is still on the top of the stack.

Illustrates rotation and translation

Although this isn't the primary purpose of this lesson, the lesson also provides a good illustration of the benefits of rotation and translation.

As we will see when we examine the code, the task of rotating the ellipses relative to each other and the task of translating them into the various quadrants was made much easier (even possible) through the use of the **AffineTransform** to rotate and translate the ellipses.

The normal caveat regarding inches

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.

```
//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));
```

Figure 1

The large circle

Figure 2 draws the large circle with a border width of 0.1 inches. There is nothing new here, so I won't provide an explanation.

```
//Draw a big circle underneath all of the ellipses
g2.setStroke(new BasicStroke(0.1f*ds));
Ellipse2D.Double bigCircle =
            new Ellipse2D.Double(
              -1.5*ds,-1.5*ds,3.0*ds,3.0*ds);
g2.draw(bigCircle);
```

Figure 2

An ellipse reference variable

Figure 3 simply declares a reference variable of the class **Ellipse2D.Double**. This reference variable will be used repeatedly in subsequent code for the instantiation of ellipse objects.

```
   Ellipse2D.Double theEllipse;
```

Figure 3

Now things get interesting

At this point, things get interesting. I need to draw the three filled ellipses in the upper-left quadrant. One way to do this (the hard way) would be to calculate the coordinates of the ellipses in the quadrant and define them according to those coordinates.

The easier way is to translate the origin to the center of what was previously the upper-left quadrant, and to define the bounding rectangle for the ellipses centered on the new origin.

Figure 4 translates the origin to the center of what was previously the upper-left quadrant. After this statement is executed, any geometric figure that is drawn centered on the origin will actually be rendered in the center of what was earlier the upper-left quadrant.

```
   g2.translate(-1.0*ds,-1.0*ds);
```

Figure 4

Opaque red ellipse

The next several fragments draw and fill a red opaque ellipse centered on the new origin.

The code in Figure 5 has been covered in previous lessons, so I won't discuss it further in this lesson.

```
   theEllipse = new Ellipse2D.Double(
            -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
   g2.setPaint(Color.red);
```

Figure 5

setComposite() and AlphaComposite

Figure 6 shows the use of the **setComposite()** method and the **AlphaComposite** class.

```
   g2.setComposite(
      AlphaComposite.getInstance(
           AlphaComposite.SRC_OVER,1.0f));
```

Figure 6

An understanding of this single statement is pretty much the heart of this entire lesson.

As mentioned earlier, the parameter to the **setComposite()** method must be a reference to an object that implements the **Composite** interface. This requirement can be satisfied by passing a reference to an object of the **AlphaComposite** class.

Also, as mentioned earlier, such an object can be obtained only by invoking the factory method, named **getInstance()** of the **AlphaComposite** class.

Parameters to getInstance()

The **getInstance** method requires two parameters. The first parameter is an **int** whose value specifies the compositing rule that will be used. The second is the alpha value that will be used with that rule to establish how compositing will be accomplished.

This fragment specifies the **SRC_OVER** rule by passing that symbolic constant from the **AlphaComposite** class. You might want to use this program to experiment with the other rules of the **AlphaComposite** class and view the results.

Specifying transparency

This fragment also specifies that the source object will be opaque.

For this compositing rule, an alpha parameter value of **1.0f** specifies opaque while a value of **0.0f** specifies total transparency.

It is primarily the value of the alpha parameter that will be varied throughout the remainder of this program to achieve the desired results.

Rendering the red ellipse

Figure 7 simply renders the ellipse according to the **paint** and **composite** properties previously established. There is nothing new here.

```
   g2.fill(theEllipse);
```

Figure 7

Green opaque ellipse at sixty degrees

To some extent, the use of translation above was for convenience. I could have placed the red ellipse in the upper-left quadrant by specifying a bounding rectangle at that location.

However, the code in Figure 8 will render a green opaque ellipse rotated by an angle of sixty degrees. This rotational transformation is not simply for convenience. I don't know of any other way to draw an ellipse whose major axis is neither horizontal nor vertical, except for use of the **GeneralPath** class to construct the ellipse piecemeal. I certainly don't want to create the ellipse using **GeneralPath**.

```
   theEllipse = new Ellipse2D.Double(
             -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.rotate(Math.PI/3.0);//rotate 60 degrees
    g2.setPaint(Color.green);
    g2.setComposite(
            AlphaComposite.getInstance(
              AlphaComposite.SRC_OVER,1.0f));
    g2.fill(theEllipse);
```

Figure 8

The code in Figure 8

- Defines a new ellipse centered on the new origin with a horizontal major axis.
- Specifies that it will be rendered by rotating it sixty degrees.
- Sets its fill color to green.
- Sets its transparency to opaque.
- Renders it in device space (the screen).

Blue opaque ellipse at 120 degrees

Recall from the earlier lesson on the use of the **AffineTransform** that successive calls to the **rotate()** method produce cumulative angles of rotation. Therefore, in order to rotate the blue ellipse by a total of 120 degrees, the following fragment invokes another sixty-degree rotation.

Otherwise, the code in Figure 9 is straightforward, producing an opaque blue ellipse in the upper-left quadrant rotated by a total of 120 degrees relative to the red ellipse.

```
  theEllipse = new Ellipse2D.Double(
             -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
   g2.rotate(Math.PI/3.0);//rotate 60 degrees
   g2.setPaint(Color.green);
   g2.setComposite(
          AlphaComposite.getInstance(
             AlphaComposite.SRC_OVER,1.0f));
   g2.fill(theEllipse);
```

Figure 9

As mentioned earlier, the blue ellipse is on the top of the stack, the red ellipse is on the bottom of the stack, and the green ellipse is in the middle.

Rotation can produce complexity

The combination of rotation and translation can produce very complicated results. The objective of the next few fragments is to produce ellipses having different transparency values in the upper-right quadrant.

Translate origin to upper-right quadrant

To avoid the complexity mentioned above, before translating the origin to the center of the original upper-right quadrant, the code in Figure 10 reverses the rotation previously imposed by the code discussed above.

```
  //undo previous rotation
  g2.rotate(-2*(Math.PI/3.0));
  g2.translate(2.0*ds,0.0*ds);
```

Figure 10

Note that the angle of rotation is negative, and amount of rotation is sixty degrees multiplied by two.

Following reversal of the previous rotation, the origin is translated. In this case, the new origin is translated two inches to the right of the previous origin, but at the same vertical position. This places the new origin in the center of what was originally the upper-right quadrant.

Repetitive code

From this point forward, the code becomes very repetitive. Therefore, I am going to show only that code that distinguishes the rendering of the ellipses in one quadrant from the rendering in the other quadrants, with respect to transparency.

You can view the entire program at the end of the lesson.

Upper-right quadrant

Figure 11 shows the invocation of the **setComposite()** methods for the red, green, and blue ellipses respectively.

```
//...
//Red is not transparent
g2.setComposite(
    AlphaComposite.getInstance(
       AlphaComposite.SRC_OVER,1.0f));
//...
g2.setComposite(
    AlphaComposite.getInstance(
       AlphaComposite.SRC_OVER,1.0f));
//...
//Blue is 50-percent transparent
g2.setComposite(
    AlphaComposite.getInstance(
       AlphaComposite.SRC_OVER,0.5f));
//...
```

Figure 11

As you can see, the first two are opaque (alpha equals **1.0f**) while the blue ellipse is fifty-percent transparent (alpha equals **0.5f**).

Lower-left quadrant

Figure 12 shows the invocation of the **setComposite()** methods for the red, green, and blue ellipses respectively in the lower-left quadrant.

```
//...
//Red is not transparent
g2.setComposite(
    AlphaComposite.getInstance(
       AlphaComposite.SRC_OVER,1.0f));
//...
//Green is 50 percent transparent
g2.setComposite(
    AlphaComposite.getInstance(
       AlphaComposite.SRC_OVER,0.5f));
//...
//Blue is 90-percent transparent
g2.setComposite(
    AlphaComposite.getInstance(
       AlphaComposite.SRC_OVER,0.1f));
```

Figure 12

The red ellipse is opaque (alpha equals **1.0f**). The green ellipse is fifty-percent transparent (alpha equals **0.5f**), while the blue ellipse is ninety-percent transparent (alpha equals **0.1f**).

Lower-right quadrant

Figure 13 shows that the red ellipse is opaque (alpha equals **1.0f**). The green and blue ellipses are ninety-percent transparent (alpha equals **0.1f**).

```
  //...
  //Red is not transparent
  g2.setComposite(
      AlphaComposite.getInstance(
         AlphaComposite.SRC_OVER,1.0f));
  //...
  //Green is 90-percent transparent
  g2.setComposite(
      AlphaComposite.getInstance(
          AlphaComposite.SRC_OVER,0.1f));
  //...
  //Blue is 90-percent transparent
  g2.setComposite(
      AlphaComposite.getInstance(
          AlphaComposite.SRC_OVER,0.1f));
```

Figure 13

Summary

In this lesson, I have shown you how to use the **setComposite()** method of the **Grapics2D** class along with the **AlphaComposite** class to control the manner in which new pixel values are composited with existing pixel values. The sample programs in this lesson concentrate on transparency as a way to demonstrate compositing pixels.

In addition, you have seen some additional uses for the **translate** and **rotate** transforms.

Complete Program Listing

A complete listing of the program is provided in Figure 14.

```
/*Composite01.java 12/12/99
 Copyright 1999, R.G.Baldwin

 Illustrates use of the AlphaComposite class to
 achieve transparency with solid-fill colors.

 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.

Draw a big circle centered on the origin underneath
all of the ellipses.

Uses rotation and translation to fill three ellipses in
each of the four quadrants.  The ellipses intersect
at their center.  Each is rotated by 60 degrees
relative to the one below it.  The order is:
  Red ellipse on the bottom
  Green ellipse in the middle
  Blue ellipse on the top

TRANSPARENCY
Upper-left quadrant
No transparency

Upper-right quadrant
Red is not transparent
Green is not transparent
Blue is 50-percent transparent

Lower-left quadrant
Red is not transparent
Green is 50-percent transparent
Blue is 90-percent transparent

Lower-right quadrant
Red is not transparent
Green is 90-percent transparent
Blue is 90-percent transparent


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 under WinNT Workstation 4.0
******************************************/
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class Composite01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Composite01

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));

  //Draw a big circle underneath all of the ellipses.
  g2.setStroke(new BasicStroke(0.1f*ds));
  Ellipse2D.Double bigCircle =
             new Ellipse2D.Double(
                  -1.5*ds,-1.5*ds,3.0*ds,3.0*ds);
  g2.draw(bigCircle);

  Ellipse2D.Double theEllipse;


  //Translate origin to upper-left quadrant
  g2.translate(-1.0*ds,-1.0*ds);

  //Red horizontal ellipse
  theEllipse = new Ellipse2D.Double(
                -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
  g2.setPaint(Color.red);
  //Red is not transparent
  g2.setComposite(AlphaComposite.getInstance(
                  AlphaComposite.SRC_OVER,1.0f));
  g2.fill(theEllipse);

  //Green ellipse at 60 degrees
  theEllipse = new Ellipse2D.Double(
                 -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
  g2.rotate(Math.PI/3.0);//rotate 60 degrees
  g2.setPaint(Color.green);
  //Green is not transparent
  g2.setComposite(AlphaComposite.getInstance(
                 AlphaComposite.SRC_OVER,1.0f));
  g2.fill(theEllipse);

  //Blue ellipse at 120 degrees
  theEllipse = new Ellipse2D.Double(
               -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
  g2.rotate((Math.PI/3.0));//rotate 60 more deg
  g2.setPaint(Color.blue);
  //Blue is not transparent
  g2.setComposite(AlphaComposite.getInstance(
                  AlphaComposite.SRC_OVER,1.0f));
  g2.fill(theEllipse);


  //Translate origin to upper-right quadrant
```

```
    g2.rotate(-2*(Math.PI/3.0));//undo prev rotation
    g2.translate(2.0*ds,0.0*ds);

    //Red horizontal ellipse
    theEllipse = new Ellipse2D.Double(
                -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.setPaint(Color.red);
    //Red is not transparent
    g2.setComposite(AlphaComposite.getInstance(
                 AlphaComposite.SRC_OVER,1.0f));
    g2.fill(theEllipse);

    //Green ellipse at 60 degrees
    theEllipse = new Ellipse2D.Double(
                -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.rotate(Math.PI/3.0);//rotate 60 degrees
    g2.setPaint(Color.green);
    //Green is not transparent
    g2.setComposite(AlphaComposite.getInstance(
                 AlphaComposite.SRC_OVER,1.0f));
    g2.fill(theEllipse);

    //Blue ellipse at 120 degrees
    theEllipse = new Ellipse2D.Double(
                -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.rotate((Math.PI/3.0));//rotate 60 more deg
    g2.setPaint(Color.blue);
    //Blue is 50-percent transparent
    g2.setComposite(AlphaComposite.getInstance(
                 AlphaComposite.SRC_OVER,0.5f));
    g2.fill(theEllipse);


    //Translate origin to lower-left quadrant
    g2.rotate(-2*(Math.PI/3.0));//undo prev rotation
    g2.translate(-2.0*ds,2.0*ds);

    //Red horizontal ellipse
    theEllipse = new Ellipse2D.Double(
                -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.setPaint(Color.red);
    //Red is not transparent
    g2.setComposite(AlphaComposite.getInstance(
                 AlphaComposite.SRC_OVER,1.0f));
    g2.fill(theEllipse);

    //Green ellipse at 60 degrees
    theEllipse = new Ellipse2D.Double(
                -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.rotate(Math.PI/3.0);//rotate 60 degrees
    g2.setPaint(Color.green);
    //Green is 50 percent transparent
    g2.setComposite(AlphaComposite.getInstance(
                 AlphaComposite.SRC_OVER,0.5f));
    g2.fill(theEllipse);

    //Blue ellipse at 120 degrees
    theEllipse = new Ellipse2D.Double(
                -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.rotate((Math.PI/3.0));//rotate 60 more deg
    g2.setPaint(Color.blue);
    //Blue is 90-percent transparent
    g2.setComposite(AlphaComposite.getInstance(
                 AlphaComposite.SRC_OVER,0.1f));
    g2.fill(theEllipse);


    //Translate origin to lower-right quadrant
    g2.rotate(-2*(Math.PI/3.0));//undo prev rotation
    g2.translate(2.0*ds,0.0*ds);
    //Red horizontal ellipse
    theEllipse = new Ellipse2D.Double(
               -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.setPaint(Color.red);
    //Red is not transparent
    g2.setComposite(AlphaComposite.getInstance(
                AlphaComposite.SRC_OVER,1.0f));
    g2.fill(theEllipse);

    //Green ellipse at 60 degrees
    theEllipse = new Ellipse2D.Double(
               -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.rotate(Math.PI/3.0);//rotate 60 degrees
    g2.setPaint(Color.green);
    //Green is 90-percent transparent
    g2.setComposite(AlphaComposite.getInstance(
                AlphaComposite.SRC_OVER,0.1f));
    g2.fill(theEllipse);

    //Blue ellipse at 120 degrees
    theEllipse = new Ellipse2D.Double(
               -1.0*ds,-0.25*ds,2.0*ds,0.5*ds);
    g2.rotate((Math.PI/3.0));//rotate 60 more deg
    g2.setPaint(Color.blue);
    //Blue is 90-percent transparent
    g2.setComposite(AlphaComposite.getInstance(
                AlphaComposite.SRC_OVER,0.1f));
    g2.fill(theEllipse);

  }//end overridden paint()

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

Figure 14

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-