February 9, 2000

# Java 2D Graphics, the Graphics2D Class

Java Programming, Lecture Notes # 304

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



---

## Introduction

The **Graphics2D** class**,** which was released with JDK 1.2, extends the **Graphics** class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. Beginning with JDK 1.2, this is the fundamental class for rendering two-dimensional shapes, text and images.

Because it extends the **Graphics** class, the capabilities of the **Graphics** class that existed in earlier versions of the JDK continue to be available.

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

### Setter methods

The manner in which **Graphics2D** renders shapes, text, and images depends on the current values of several properties of the **Graphics2D** object.  The values of these properties are

controlled using standard *setter* methods of the class. This and subsequent lessons will discuss the following methods that are used to set the properties of the object.

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

This lesson provides an overview of **Graphics2D** and discusses transforms. It illustrates the use of the **setTransform()** method for *setting* an Affine transform.

Information about the other classes listed above, and additional information about **Graphics2D** will be provided in subsequent lessons.

# Coordinates

A system of device-independent coordinates (called *User Space*) is used to pass all coordinate information to the methods of a **Graphics2D** object. An **AffineTransform** object (see definition below) is contained in the **Graphics2D** object as part of its state. This **AffineTransform** object defines how to convert coordinates from user space to device-dependent coordinates in *Device Space.*

## What is an Affine transform?

According to Sun:

> The **AffineTransform** class represents a 2D Affine transform that performs a linear mapping from 2D coordinates to other 2D coordinates that preserves the "straightness" and "parallelness" of lines. Affine transformations can be constructed using sequences of translations, scales, flips, rotations, and shears.

The transformation of coordinate information from *User Space* to *Device Space* may consist either of

- Transformation to a space representing individual pixels on a device of known resolution, or
- Transformation into a graphics metafile for playback on a device of unknown physical resolution at a later time.

In the latter case, the **Graphics2D** transform is set up to transform user coordinates to a virtual device space that approximates the expected resolution of the target device. If the estimate proves to be incorrect, it might be necessary to apply further transformations at playback time.

All **Graphics2D** methods take user space coordinates.

## GraphicsConfiguration object

Every **Graphics2D** object is associated with a target that defines where rendering takes place (such as the screen or a printer), and the same rendering target is used throughout the life of a **Graphics2D** object. A **GraphicsConfiguration** object defines the characteristics of the rendering target, such as pixel format and resolution.

According to Sun:

> The **GraphicsConfiguration** class describes the characteristics of a graphics destination such as a printer or monitor. There can be many **GraphicsConfiguration** objects associated with a single graphics device. For example, on X11 windowing systems, each visual is a different **GraphicsConfiguration**. On PCs and Macintoshes, the different screen resolution/color resolution combinations would be different **GraphicsConfiguration** objects.

According to the Sun documentation, when the **Graphics2D** object is created, a default transform for the target of the **Graphics2D** (a Component or Image) is specified by the **GraphicsConfiguration**. This default transform is used to map the user space coordinate system to screen or printer device coordinates. The origin maps to the upper left hand corner of the target region of the device. Increasing X coordinates extend to the right and increasing Y coordinates extend downward.

## What, me worry?

According to Java 2D Graphics, by Jonathan Knudsen,

*"In general, you don't ever have to worry about the details of a particular device. Your applications live in User Space. As long as you remember that User Space, by default, has 72 coordinates per inch, Java 2D will ensure that everything is the right size on your output devices."*

In other words, it seems that Sun and Knudsen are saying that if I create a square whose dimensions are 72 units by 72 units, that square will be rendered on my output device as a square with dimensions of one inch on each side. The idea is to stop thinking about dimensions in terms of pixels and to start thinking of dimensions in *User Space* with 72 units per inch.

## Something isn't working!

While the above may be true of output devices such as printers, I don't find it to be true for screen output. For example, the computer that I am working on right now reports a screen resolution of 120 pixels per inch. If I draw a 72x72 square on the screen using the **Rectangle2D** class, the square is rendered at a size of about 0.6 inch on each side. In other words, the square is rendered on the screen as a square that is 72 pixels on each side, and not one inch on each side.

## Is there a workaround?

Obviously I can use the **getScreenResolution()** method of the **Toolkit** class (that returns the screen resolution in pixels per inch) to scale my coordinate values, but the requirement to do that doesn't seem to be consistent with the above discussion. (A sample program later in this tutorial lesson shows how to do that.)

At least I could do that if the **getScreenResolution()** method would return the correct value for actual screen resolution. As it turns out, this method always returns 120 on my machine no matter what the actual resolution is. This value is fairly close for a screen size of 1280 by 1024, but isn't even close for other screen settings.

I am going to assume that this is a bug, or at least an incompatibility between JDK 1.2.2, WinNT Workstation 4.0, and the driver software for my screen, and proceed through these lessons as if the method **getScreenResolution()** returns the correct value. Hopefully, someday it will.

By the way, the method **getScreenSize()** of the Toolkit class does return the correct values for total horizontal and vertical size in pixels.

Also, I can define and set a modified **AffineTransform** object that compensates for the difference in the actual screen resolution and the assumed default screen resolution of 72 pixels

per inch. Then I can forget about pixels and think in terms of *User Space* coordinates with 72 units per inch. This also doesn't seem to be consistent with the explanation given earlier because it still requires me to deal explicitly with actual screen resolution. (Another sample program later in this lesson shows how to do this as well.)

# David Flanagan to the Rescue!

Just when I was beginning to feel distraught because my experience wasn't matching what Sun and Knudsen seemed to be saying, one of my favorite authors, David Flanagan came to the rescue. In Java Foundation Classes in a Nutshell, Flanagan tells us,

*"When drawing to a screen or an off-screen image, X and Y coordinates are measured in pixels. When drawing to a printer or other high-resolution device, however, X and Y coordinates are measured in points instead of pixels (and there are 72 points in one inch)."*

Flanagan goes on to tell us

*"By default, when drawing to a screen or image, user space is the same as device space. However, the **Graphics2D** class defines methods that allow you to trivially modify the default coordinate system... By default, when drawing to the screen, one unit in user space corresponds to one pixel in device space. The **scale()** method changes this. If you scale the coordinate system by a factor of 10, one unit of user space corresponds to 10 pixels in device space..."*

## A new AffineTransform object

Although the sample programs in this lesson don't make use of the **scale()** method as described by Flanagan, one of the programs does take a similar approach by creating a new **AffineTransform** object to provide the necessary scaling from user space to device space on the screen.

On a related subject, in the same section, Flanagan discusses how **Graphics2D** provides improved graphics capability for high-resolution devices, such as printers, by allowing us to specify coordinate values as either **float** or **double**, rather than being required to specify them as integers, as has been the case prior to the release of **Graphics2D**.

Flanagan has published another excellent book in Java Foundation Classes in a Nutshell, and I highly recommend it.

# The Rendering Process

According to the Sun documentation, the rendering process consists of the following four steps that are controlled by the **Graphics2D** rendering attributes:

- Determine what to render.

- Constrain the rendering operation to the current *Clip* (see below).
- Determine what colors to render.
- Apply the colors to the destination drawing surface using the current *Composite* attribute in the **Graphics2D** context.

The *Clip* refers to a defined region outside of which nothing is drawn. Additional information from Sun about the Clip follows:

> The Clip is specified by a **Shape** in user space and is controlled by the program using the various clip manipulation methods of **Graphics** and **Graphics2D**. This user clip is transformed into device space by the current Transform and combined with the device clip, which is defined by the visibility of windows and device extents. The combination of the user clip and device clip defines the composite clip, which determines the final clipping region. The user clip is not modified by the rendering system to reflect the resulting composite clip.

The *Composite* attribute defines how the colors of the new item are combined with the existing colors in the location where the item is to be rendered. There are a variety of choices here that I will discuss in a subsequent lesson. For example, this is where you can control the *transparency* to cause the new item to completely cover an existing item, or to allow the pixels of an existing item to partially show through the new item being rendered at that location.

## What is a Shape?

I will have more to say about the **Shape** interface in a subsequent lesson. However, for the time being, the following information should suffice. 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.

The **Shape** interface is implemented by the following classes (and possibly others as well). These are classes from which geometric objects can be instantiated: **Polygon**, **RectangularShape**, **Rectangle**, **Line2D**, **CubicCurve2D**, **Area**, **GeneralPath**, and **QuadCurve2D**.

Other geometric classes, such as **Rectangle2D** extend these geometric classes, thus leading to other classes that implement the **Shape** interface indirectly. The bottom line is that the **Shape** interface completely permeates the Java 2D Graphics API.

## Types of Rendering Operations

According to Sun, there are three types of rendering operations:

- Shape operations
- Text operations
- Image operations

Each of these types will be discussed briefly in the following sections. (More detailed discussions will be provided in subsequent lessons.) However, before getting into the operations, it will be necessary to provide some information about the **Stroke** interface.

## The Stroke Interface

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

The **Stroke** interface allows a **Graphics2D** object to obtain a **Shape** that is the decorated outline, or stylistic representation of the outline, of the specified **Shape**. Stroking a **Shape** is like tracing its outline with a marking pen of the appropriate size and shape. The area where the pen would place ink is the area enclosed by the outline **Shape**.

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

drawRoundRect, drawOval, drawArc, drawPolyline, and drawPolygon.

You need to read this very carefully to make sure that you understand it. What this says to me is that the **Stroke** interface is used to produce a **Shape** that forms the outline of another **Shape**.

### A simple example

For example, consider tracing the complete outline of a star, on a piece of tracing paper, using a wide marking pen. The drawing that you produce on the tracing paper can itself be considered to be a shape, consisting of a pair of (approximately) parallel lines,

- whose color is the same as the color of the ink in the pen,
- which connect at their ends to form two closed, non-intersecting contours, and
- which are filled in between with the color of the ink in the marking pen.

At least, I think that is what this means.

## Shape Operations

Rendering operations of the **Shape** variety consist briefly of the following steps:

- If the rendering operation is a **draw(Shape)** operation, create a new **Shape** object that represents the outline of the **Shape** being rendered.
- Transform the new **Shape** from user space to device space using the current **Transform** in the **Graphics2D** context.
- Extract the outline of the **Shape** using the **getPathIterator()** method of the **Shape** interface. This returns a **PathIterator** object that iterates along the boundary of the **Shape**.
- In those cases where the **Graphics2D** object can't handle the curved segments that the **PathIterator** object returns, an alternative **getPathIterator()** method of **Shape**, which redefines the shape as a series of straight line segments, can be used.
- Query the current **Paint** in the **Graphics2D** context for a **PaintContext**, which specifies the colors to render in device space.

I will be discussing **PathIterator, Paint**, and **PaintContext** in a subsequent lesson.

## Text Operations

Rendering operations of the **Text** variety consist briefly of the following steps:

- Determine the set of *glyphs* required to render the indicated **String**. There are several possibilities here that I will discuss in subsequent lessons.

- Query the current **Font** to obtain outlines for the indicated glyphs. These outlines are treated as shapes in user space relative to the position of each glyph that was determined in step 1.
- Fill the character outlines as indicated above under **Shape** operations.
- Query the current **Paint** for a **PaintContext**, which specifies the colors to render in device space.

That brings us to the third type of rendering operation, which is an **Image** operation.

## Image Operations

Rendering operations of the **Image** variety consist briefly of the following steps:

- Get the region of interest, which is defined by the bounding box of the source **Image**. This bounding box is specified in *Image Space*, which is the Image object's local coordinate system.
- Use the **AffineTransform** that is passed to **drawImage(Image, AffineTransform, ImageObserver)** to transform the bounding box from image space to user space. If no **AffineTransform** is supplied, treat the bounding box as if it is already in user space.
- Transform the bounding box of the source Image from user space into device space using the current **Transform**.
- Use the **Image** object, sampled according to the source to destination coordinate mapping specified by the current **Transform** and the optional image transform to determine what colors to render

I will have a lot more to say about images in subsequent lessons.

## Rendering Attributes

As the author of the program, you have control over a number of rendering attributes. The default rendering attributes are given below:

- **Paint**: The color of the Component.
- **Font**: The Font of the Component.
- **Stroke**: A square pen with a line width of 1, no dashing, miter segment joins and square end caps (will provide more information in subsequent lessons).
- **Transform**: The **getDefaultTransform** for the **GraphicsConfiguration** of the **Component**.
- **Composite**: The AlphaComposite.SRC_OVER rule.
- **Clip**: No rendering Clip, the output is clipped to the Component.

I am providing this information here simply as a very brief introduction to the topic of rendering attributes. I will have much more to say about this topic in subsequent lessons.

## Additional Details and Compatibility Issues

There is a great deal more detailed, and sometimes complex information involved in a complete understanding of the **Graphics2D** class, particularly with respect to compatibility issues between **Graphics2D** and legacy code written under the earlier JDK 1.1. While I will be covering many aspects of **Graphics2D** in subsequent lessons, I won't spend much time dealing with compatibility issues and legacy code. You are encouraged to visit the Sun documentation for information on compatibility issues.

# Sample Programs

I don't have the final answers to the apparent ambiguity surrounding the automatic transformation from user space to device space with respect to screen coordinates. However, I can show you a couple of ways to write code that will deal with the required transformation. Two programs follow that contain this information.

## Program Graphics2D01.java

This program illustrates the use of the **Graphics2D** class and the **Rectangle2D** class. The screen width, screen height, and screen resolution are obtained by the program and displayed on the screen following the command line. For the particular computer that I am using at the time of this writing, the screen output is:

```
120 pixels per inch
1280 pixels wide
1024 pixels high
```

This program should run successfully on your computer also, except that you may get different values for the resolution, width, and height of the screen.

The program produces a **Frame** object on the screen that is two inches on each side. The screen resolution (in pixels per inch) is used to establish the size of the **Frame** object in inches. In addition, the screen resolution is used to draw a square that is one inch on each side, centered in the **Frame**.

This program is structured to make it as easy to follow as possible (For the most part, I have tried to avoid the use of cryptic constructs such as Inner Classes). I will break this program up and discuss it in fragments. A complete listing of the program is provided at the end of the lesson.

## The controlling class

Figure 1 shows the import directives and the entire controlling class. As you can see, the **main()** method in the controlling class does nothing but instantiate an object of another class named **GUI**. All of the action occurs in the **GUI** class.

```
/*Graphics2D01.java 12/12/99
Copyright 1999, R.G.Baldwin
```

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

class Graphics2D01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Graphics2D01
```

**Figure 1**

## The GUI class

Figure 2 shows the beginning of the **GUI** class. Note that this class extends the **Frame** class. This is necessary so that I can override the **paint()** method to make it possible to use a **Graphics2D** object to render the desired graphics on the screen.

```
class GUI extends Frame{
  int res;//store screen resolution here
  int width;//store screen width here
  int height;//store screen height here
```

**Figure 2**

This class declares three instance variables that are later used to store information about the resolution, width, and height of the computer screen.

## The constructor for the GUI class

Figure 3 shows the beginning of the constructor including the code used to get and display the resolution, width, and height of the computer on which the program is executing. Hopefully this is "old stuff" to you by now, because it really has nothing to do with the Java 2D Graphics API.

```
GUI(){//constructor
  //Get screen resolution, width, and height
  res = Toolkit.getDefaultToolkit().
                  getScreenResolution();
  width = Toolkit.getDefaultToolkit().
                  getScreenSize().width;
  height = Toolkit.getDefaultToolkit().
                           getScreenSize().height;

  //Display screen resolution,
  // width, and height
  System.out.println(res + " pixels per inch");
  System.out.println(width + " pixels wide");
  System.out.println(height + " pixels high");
```

**Figure 3**

## Using screen resolution data

Figure 4 continues the constructor and shows the use of the screen resolution information to cause the size of the **Frame** object to be two inches on each side. It also causes the **Frame** to become visible, and places a title in the title bar on the **Frame**.

```
  //Set Frame size to two-inch by two-inch
  this.setSize(2*res,2*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
```

**Figure 4**

I know that I promised to avoid cryptic code, but I couldn't resist the use of an anonymous Inner Class to create and register a Listener object to terminate the program when the user clicks the

*close* button on the top of the **Frame**. This is a very standard use of anonymous Inner Classes, and hopefully you know all about this sort of thing by this point in your Java studies.

## Finally, the Graphics2D class

Finally (thanks for your patience), I am going to talk about **Graphics2D.** The following code fragment shows the beginning of a **paint()** method that is overridden to draw a square, one inch on each side, which is centered in the **Frame** object. Note that, as was the case in JDK 1.1, the overridden **paint()** method always receives a reference to an object of the class **Graphics**. Thus, for backward compatibility, all of the methods of the **Graphics** class are still available.

## Downcast is required

However, the **Graphics2D** class extends the **Graphics** class, and in order to gain access to the new capabilities of the **Graphics2D** class, it is necessary to downcast the incoming reference to **Graphics2D** as shown in Figure 5.

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

**Figure 5**

## Invoking the draw(Shape) method

Having downcast the reference to **Graphics2D**, I can now invoke the **draw(Shape)** method of the **Graphics2D class** on that reference. Invoking this method strokes the outline of the **Shape** using the settings of the current **Graphics2D** context. The rendering attributes that are applied include the Clip, Transform, Paint, Composite and Stroke attributes. I will have more to say about these attributes in subsequent lessons.

Figure 6 instantiates and draws an object of the **Rectangle2D** class that is centered in the **Frame** object. The size of the **Rectangle2D** object is one inch on each side. The rectangle is centered in the **Frame** by placing its upper left-hand corner at a position that is one-half inch to the right and one-half inch below the upper left-hand corner of the **Frame**.

```
   g2.draw(new Rectangle2D.Double(
           res*0.5,res*0.5,res*1.0,res*1.0));
  }//end overridden paint()
}//end class GUI
```

**Figure 6**

## Actually this is a Rectangle2D.Double object

Note that the actual rectangle object is instantiated from the **Rectangle2D.Double** class. If you are not familiar with this terminology, see the first lesson in this series on the Java 2D Graphics API for an explanation of just what this means.

Note also that the screen resolution in pixels per inch is used in the expressions that convert the dimensions in inches to dimensions in pixels for drawing on the screen. Thus, although the **draw()** method is new to the 2D API, the sizing and positioning of the rectangle does not use the new features of the 2D API. This same approach can be used with JDK 1.1 to size and locate an object in screen space.

The next sample program will make more significant use of the new features of the 2D API.

## Program Graphics2D02.java

As with the previous program, this program illustrates the use of the **Graphics2D** class and the **Rectangle2D** class. However, unlike the previous program, this program also illustrates the use of the **AffineTransform** class.

## Compensating for actual screen resolution

A newly instantiated object of the **AffineTransform** class is used to compensate for the difference in actual screen resolution on my computer (120 pixels per inch) and the default screen resolution (72 pixels per inch) used the 2D API.

As before, this program uses the actual screen resolution to place a **Frame** object on the screen that is two inches on each side. It then uses the **AffineTransform** object mentioned above to draw a square that is one inch on each side, centered in the **Frame**.

The drawing of the square inside the **Frame** is based on inches scaled by the default resolution used by the 2D API of 72 pixels per inch (or 72 printer points per inch). The resulting square is scaled by the **AffineTransform** that compensates for the difference in actual screen resolution and the default resolution to produce a square that is one inch on each side.

As usual, I will discuss the program in fragments. A complete listing of the program is provided at the end of the lesson.

## The entire controlling class plus some

Figure 7 contains the entire controlling class, along with the constructor for the **GUI** class. This code is essentially the same as in the previous program, so I won't discuss it further.

```
/*Graphics2D02.java 12/12/99
Copyright 1999, R.G.Baldwin

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

class Graphics2D02{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Graphics2D02

class GUI extends Frame{
  int res;//store screen resolution here

  GUI(){//constructor
    //Get screen resolution
    res = Toolkit.getDefaultToolkit().
                  getScreenResolution();

    System.out.println(res + " pixels per inch");

    //Set Frame size to two-inches by two-inches
    this.setSize(2*res,2*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
```

**Figure 7**

## Overridden paint() method

Figure 8 shows the overridden **paint()** method in the **GUI** class. As before, this fragment downcasts the incoming reference to type **Graphics2D** to provide access to the new features of the **Graphics2D** class.

```
publicvoid paint(Graphics g){
  Graphics2D g2 = (Graphics2D)g;
```

**Figure 8**

The next fragment is where the differences begin to show up between this program and the previous program.

## Using the setTransform() method

The **setTransform()** method of the **Graphics2D** class can be used to set the transform that is used to transform from user space to device space. This method requires a reference to an object of the **AffineTransform** class.

If you know about matrix algebra, you can use any of several overloaded constructors to instantiate an **AffineTransform** objects to accomplish different kinds of linear transforms.

Even if you don't know about matrix algebra, certain specialized **AffineTransform** objects are relatively easy to produce.

## Factory methods of the AffineTransform class

The **AffineTransform** class provides several *static* factory methods that can be used to produce transform objects. Some of them are listed below. These methods each return an instance of the **AffineTransform** class designed to perform a specific type of transform (rotate, scale, shear, and translate). I will have a lot more to say about these transforms in subsequent lessons.

- getRotateInstance(double theta)
- getScaleInstance(double sx, double sy)
- getShearInstance(double shx, double shy)
- getTranslateInstance(double tx, double ty)

In this program, I am interested in scaling the coordinates of the rectangle when transforming it from user space to device (screen) space to compensate for the difference in actual screen resolution and the assumed screen resolution of 72 pixels per inch that is used in the 2D API.

## A scaling instance of the AffineTransform class

Figure 9 gets a scaling instance of the **AffineTransform** class, and passes it to the **setTransform()** method of the **Graphics2D** object.

```
g2.setTransform(AffineTransform.getScaleInstance(
                        (double)res/72,(double)res/72));
```

**Figure 9**

The scale factors that are set into the transform object are the ratio of the actual screen resolution (**res**) to the assumed screen resolution of the 2D API (72 pixels per inch). This transform will then properly convert user space coordinates in inches (that assume 72 units per inch) to actual inches on the computer screen. This should work on any computer screen regardless of the actual resolution of the screen.

## Default resolution is 72 units per inch

Figure 10 begins by declaring and initializing a local variable to the value of 72 units per inch. (A more robust approach would have been to make this a public static final instance variable of the class. Then it would have been impossible to corrupt it.)

```
  int ds = 72;//default scale = 72 units per inch

  g2.draw(new Rectangle2D.Double(
            0.5*ds, 0.5*ds, 1.0*ds, 1.0*ds));
 }//end overridden paint()
}//end class GUI
```

**Figure 10**

In any event, this variable serves as a scale factor for producing user space coordinate values with 72 units per inch.

## Drawing a one-inch square

Then the fragment draws a new **Rectangle2D.Double** object, one inch on each side and centered in the **Frame** object. As before, the centering is accomplished by locating the upper left corner of the rectangle at *0.5 inch by 0.5 inch* relative to the upper left-hand corner of the **Frame**. This is accomplished by the first two parameters to the constructor.

The size of the rectangle, *1.0 inch by 1.0 inch*, is accomplished by the second two parameters to the constructor for the rectangle. In all four cases, the multiplicative factor (**ds**) is used to account for the fact that user space is assumed to be in inches where each inch is assumed to be divided into 72 parts.

So there you have it. An introduction to the use of the **Graphics2D** class along with a number of associated other aspects of the Java 2D Graphics API.

## Complete Program Listings

Complete listings of both programs are provided in Figure 11 and Figure 12.

```
/*Graphics2D01.java 12/12/99
Copyright 1999, R.G.Baldwin

Illustrates use of the Graphics2D class and the
Rectangle2D class.

Illustrates getting and displaying screen width, height,
and resolution.

Illustrates using screen resolution to produce a Frame
that is two inches on each side containing a square that
is one inch on each side, centered in the Frame.

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

class Graphics2D01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Graphics2D01

class GUI extends Frame{
  int res;//store screen resolution here
  int width;//store screen width here
  int height;//store screen height here

  GUI(){//constructor
    //Get screen resolution, width, and height
    res = Toolkit.getDefaultToolkit().
                      getScreenResolution();
    width = Toolkit.getDefaultToolkit().
                      getScreenSize().width;
    height = Toolkit.getDefaultToolkit().
                     getScreenSize().height;

    //Display screen resolution,
    // width, and height.
    System.out.println(res + " pixels per inch");
    System.out.println(width + " pixels wide");
    System.out.println(height + " pixels high");

    //Set Frame size to two-inch by two-inch
    this.setSize(2*res,2*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 to draw a one-inch by
  // one-inch rectangle centered in the Frame.
  publicvoid paint(Graphics g){
    //Downcast the Graphics object to a Graphics2D object
    // to make the features of the Graphics2D class
    // available
    Graphics2D g2 = (Graphics2D)g;

    //Instantiate and draw an object of the
    // Rectangle2D.Double class that is centered in the
    // Frame and is one inch on each side.  Center the
    // rectangle in the Frame by placing its upper left-
    // hand corner at a position that is one-half inch to
    // the right and one-half inch below the upper left-hand
    // corner of the Frame.  Note that the screen
    // resolution in pixels per inch is used to establish
    // the location and size of the rectangle in inches.
    g2.draw(new Rectangle2D.Double(
                      res*0.5,res*0.5,res*1.0,res*1.0));
  }//end overridden paint()
}//end class GUI
//====================================//
```

**Figure 11**

```
/*Graphics2D02.java 12/12/99
Copyright 1999, R.G.Baldwin

Illustrates use of the Graphics2D class, the
Rectangle2D class, and an object of the AffineTransform
class.

The object of the AffineTransform class is used to
compensate for the difference in actual screen resolution
and the default screen resolution of 72 pixels per inch
used by the API.

Illustrates using the default screen resolution and an
AffineTransform based on the actual screen resolution to
produce a Frame that is two inches on each side containing
a square that is one inch on each side, centered in the
Frame.

The size of the Frame is based on the actual screen
resolution in pixels per inch.  The drawing of the square
inside the frame is based on inches scaled by the default
resolution of 72 pixels per inch, with the resulting
square scaled by an AffineTransform that compensates for
the difference in default screen resolution and actual
screen resolution.

Tested using JDK 1.2.2 under WinNT Workstation 4.0
**********************************************/
```

```
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;

class Graphics2D02{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Graphics2D02

class GUI extends Frame{
  int res;//store screen resolution here

  GUI(){//constructor
    //Get screen resolution
    res = Toolkit.getDefaultToolkit().
                        getScreenResolution();

    System.out.println(res + " pixels per inch");

    //Set Frame size to two-inches by two-inches
    this.setSize(2*res,2*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 to draw a one-inch by
  // one-inch square centered in the Frame.
  publicvoid paint(Graphics g){
    //Downcast the Graphics object to a Graphics2D object
    // to make the features of the Graphics2D class
    // available
    Graphics2D g2 = (Graphics2D)g;

    //Set the transform to a scaling transform that
    // compensates for the difference in actual screen
    // resolution and the default screen resolution of
    // 72 pixels per inch used in the API
    g2.setTransform(AffineTransform.getScaleInstance(
              (double)res/72,(double)res/72));

    int ds = 72;//default scale = 72 pixels per inch

    //Instantiate and draw an object of the
    // Rectangle2D.Double class that is centered in the
    // Frame and is one inch on each side.  Center the
    // rectangle in the Frame by placing its upper left-
    // hand corner at a position that is one-half inch to
    // the right and one-half inch below the upper
    // left-hand corner of the Frame.  Note that the
    // default screen resolution of 72 pixels per inch is
    // used to establish the location and size of the
    // rectangle in inches.
    g2.draw(new Rectangle2D.Double(
              0.5*ds, 0.5*ds, 1.0*ds, 1.0*ds));
  }//end overridden paint()
}//end class GUI
//===================================//
```

**Figure 12**

***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-