Wrapping Up the SimplePicture Class

Learn how to cause two or more pictures to have the same width or the same height and otherwise maintain their individual aspect ratios, how to create composite pictures containing side-by-side images, how to draw text on a picture, and how to apply the same operation to every pixel in a Picture object.

Published: March 19, 2009
By Richard G. Baldwin

Java Programming Notes # 360

Preface

General

This lesson is the next in a series *(see Resources)* designed to teach you how to write Java programs to do things like:

- Remove *redeye* from a photographic image.
- Distort the human voice.
- Display one image inside another image.
- Do edge detection, blurring, and other filtering operations on images.
- Insert animated cartoon characters into videos of live humans.

If you have ever wondered how to do these things, you've come to the right place.

What you have learned from earlier lessons

If you have studied the earlier lessons in this series, you have learned about **Turtle** objects and their ability to move around in a world or a picture and to draw lines as they are moving. You have learned all about the **World** class and are in the process of learning about the **SimplePicture** class.

The class named **SimplePicture**, *(which is the superclass of the **Picture** class)*, is a large and complex class that defines almost forty methods and several constructors. By learning about those constructors and methods, you have learned that objects of the **Picture** class are useful for much more than simply serving as living quarters for turtles. They are also useful for manipulating images in interesting and complex ways.

In the previous lesson, you learned how to apply affine transforms to pictures to achieve scaling, rotation, and translation.

What you will learn in this lesson

Near the end of the previous lesson, I told you that there remained nine methods of the **SimplePicture** class that were sufficiently interesting or complicated that you would do well to learn about them.

I will explain and illustrate the following six methods from the **SimplePicture** class in this lesson:

- Picture **getPictureWithHeight**(int height)
- Picture **getPictureWithWidth**(int width)
- Pixel[] **getPixels**()
- void **addMessage**(String message, int xPos, int yPos)
- void **drawString**(String text, int xPos, int yPos)
- boolean **loadPictureAndShowIt**(String fileName)

Unable to illustrate the file writing capability

I also attempted to illustrate the following two methods to write **Picture** objects into image files:

- boolean **write**(String fileName)
- void **writeOrFail**(String fileName)throws IOException

However, the behavior of these two methods was very unreliable when running under Windows Vista Home Premium Edition and Java v1.6. Sometimes the program was able to write the file and sometimes it wasn't. Sometimes when the file was written, it would contain the image and sometimes it would be empty. As a result, I abandoned that effort for the time being. Someday I may find the time to investigate further and determine what the problem is.

The explore method and the PictureExplorer class

Several lessons back, I told you that I would illustrate and explain all of the methods in the **SimplePicture** class. With the exception of the two methods listed above, the only remaining method that I haven't explained is the **explore** method.

The **explore** method consists of a single statement that creates an object of the **PictureExplorer** class. The **PictureExplorer** class is a relatively complex class in its own right. Therefore, I will defer an explanation of the **explore** method until a future lesson that is dedicated to an explanation of the **PictureExplorer** class.

Source code listings

A complete listing of Ericson's **Picture** class is provided in Listing 16 and a listing of Ericson's **SimplePicture** class is provided in Listing 17. A listing of Ericson's **DigitalPicture** interface is provided in Listing 18. A listing of the program that I will present and explain in this lesson is provided in Listing 19.

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.

Figures

Listings

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com.

General background information

A multimedia class library

In this series of lessons, I will present and explain many of the classes in a multimedia class library that was developed and released under a **Creative Commons Attribution 3.0 United States License** *(see Resources)* by Mark Guzdial and Barbara Ericson at Georgia Institute of Technology. In doing this, I will also present some interesting sample programs that use the library.

Software installation and testing

I explained how to download, install, and test the multimedia class library in an earlier lesson titled *Multimedia Programming with Java, Getting Started (see Resources)*.

Preview

As I mentioned earlier, I will explain and illustrate the following six methods from the **SimplePicture** class in this lesson:

- Picture **getPictureWithHeight**(int height)
- Picture **getPictureWithWidth**(int width)
- Pixel[] **getPixels**()
- void **addMessage**(String message, int xPos, int yPos)
- void **drawString**(String text, int xPos, int yPos)
- boolean **loadPictureAndShowIt**(String fileName)

I will present and explain a sample program that illustrates the use of the methods in the above list

Reducing the confusion

Because I will be switching back and forth between code fragments extracted from Ericson's **SimplePicture** class and code fragments extracted from my sample program, things can get confusing.

In an attempt to reduce the confusion, I will present code fragments from Ericson's **SimplePicture** class against the background color shown in Listing 1.

Listing 1. Background color for the SimplePicture class.

```
I will present code fragments from the
SimplePicture class
against this background color.
```

Similarly, I will present code fragments from my sample program against the background color shown in Listing 2.

Listing 2. Background color for the program named Java360a.

```
I will present code fragments from my sample
programs
with this background color.
```

Discussion and sample code

The sample program named Java360a

The purpose of this program is to illustrate the use of the following methods of the **SimplePicture** class:

- boolean **loadPictureAndShowIt**(String fileName)
- Picture **getPictureWithWidth**(int width)
- void **drawString**(String text,int xPos,int yPos)
- void **addMessage**(String message, int xPos, int yPos)
- Picture **getPictureWithHeight**(int height)

- Pixel[] **getPixels**()

Unable to illustrate image file output

An attempt was also made to illustrate the following methods to write **Picture** objects into image files:

- boolean **write**(String fileName)
- void **writeOrFail**(String fileName)throws IOException

However, the results were very unreliable under *Windows Vista Home Premium Edition* and *Java 1.6*. Sometimes the program was able to write the file and sometimes it wasn't. Sometimes when the file was written, it would contain the image and sometimes it would be empty. Therefore, I abandoned the effort to illustrate and explain these two methods.

Will explain the explore method in a future lesson

This program completes the illustrations and explanations of the methods of the **SimplePicture** class with the exception of the **explore** method. The explore method will be explained in a future lesson that is dedicated to an explanation of the **PictureExplorer** class.

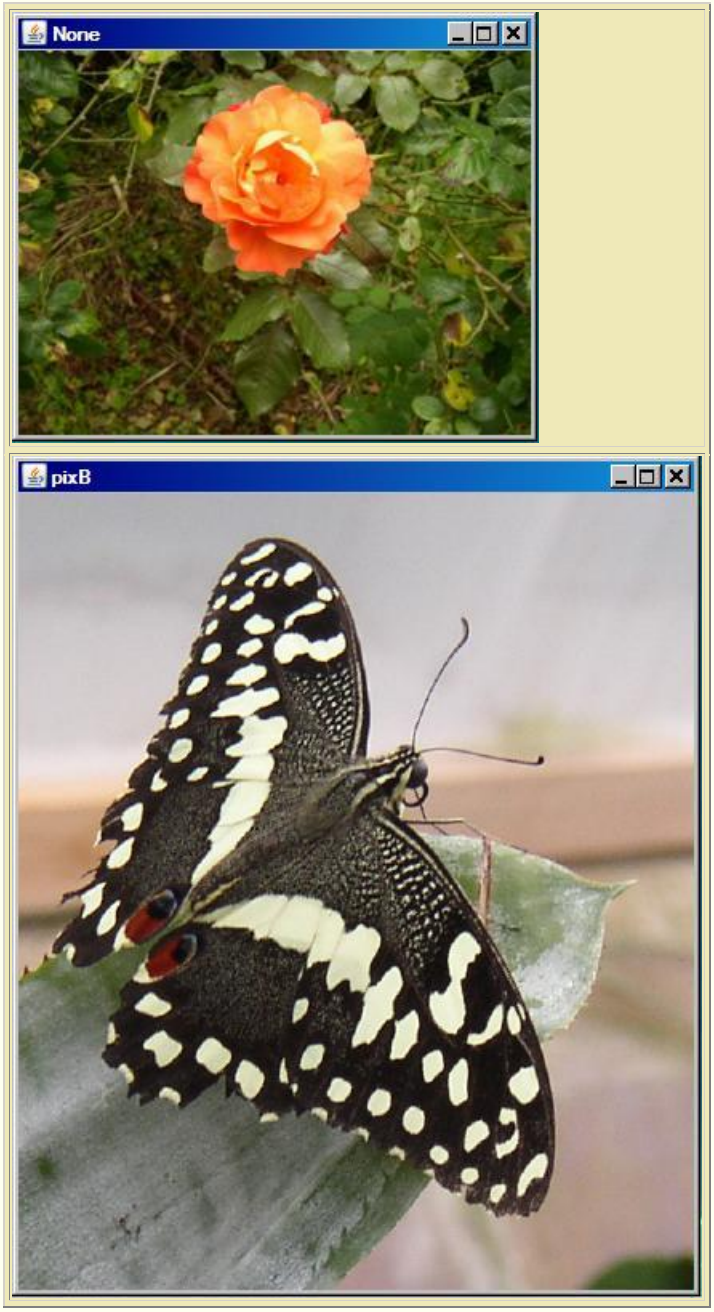
Behavior of the program

The program begins by calling the **loadPictureAndShowIt** method to load and show a picture of a rose. The title shown in the **JFrame** object is "None." *(The method doesn't set the filename as a title on the **JFrame** object.)*

A large butterfly picture

Then the program reads an image file to create a picture of a butterfly, which is much larger than the picture of the rose. These first two pictures are shown in the top and bottom of Figure 1 respectively.

Figure 1. Initial pictures of the rose and the butterfly.



Call getPictureWithWidth

Then the program calls the **getPictureWithWidth** method to create a new **Picture** object containing the butterfly image with the width set to match the width of the picture of the rose. *(Note that the aspect ratio of the butterfly picture is preserved when the width is adjusted.)*

Call the drawString method

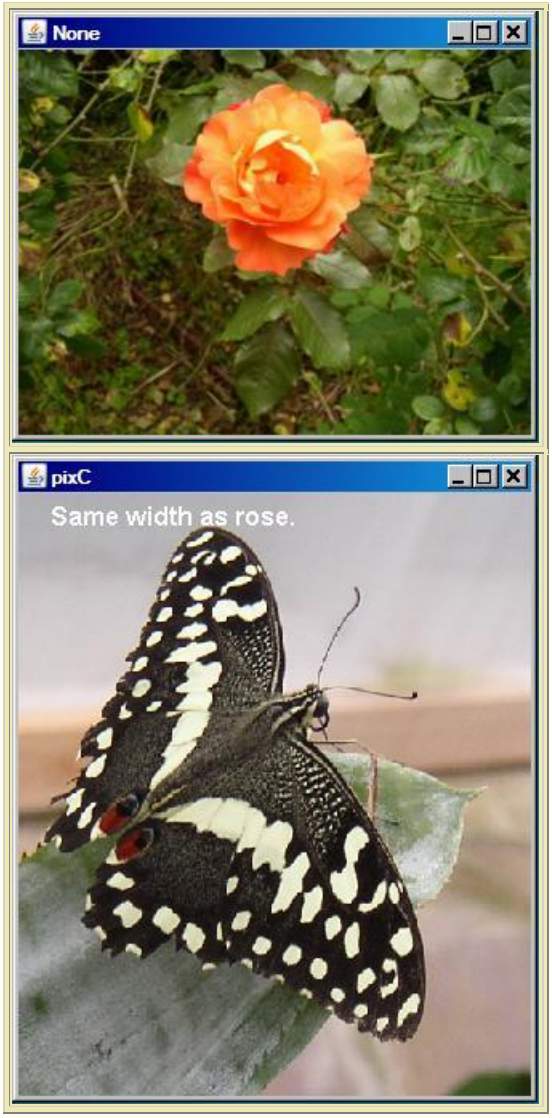
After that, the program calls the **drawString** method to draw a white text string on the picture of the butterfly. The **drawString** method calls the **addMessage** method to actually draw the text on the image. *(The color white is fixed and cannot be changed without modifying the method.)*

Pictures of rose and width-adjusted butterfly

Figure 2 shows the original picture of the rose at the top along with the new picture of the butterfly. This is the butterfly picture for which:

- The width has been adjusted to match the width of the rose.
- The white text has been drawn on the butterfly picture.

Figure 2. Pictures of rose and width-adjusted butterfly.



Call the getPictureWithHeight method

Following that, the program calls the **getPictureWithHeight** method twice to create two new pictures of the rose and the butterfly with their heights adjusted to be the same. Again, the original aspect ratio of each image is preserved.

Compose side-by-side images

Then the program calls my method named **translatePicture** from the previous lesson *(see Resources)* to copy the picture of the rose into the right side of another new **Picture** object. *(You can view the source code for the **translatePicture** method in Listing 19.)*

The program also calls the **copyPicture** method, *(which I also explained in the previous lesson)* to copy the picture of the butterfly into the left side of the same picture.

The pictures used as input to this operation are the pictures of the rose and the butterfly with the same height. This procedure results in a new **Picture** object containing side-by-side images of the butterfly and the rose as shown in Figure 3.

Figure 3. Picture with side-by-side images of rose and butterfly.

An important difference

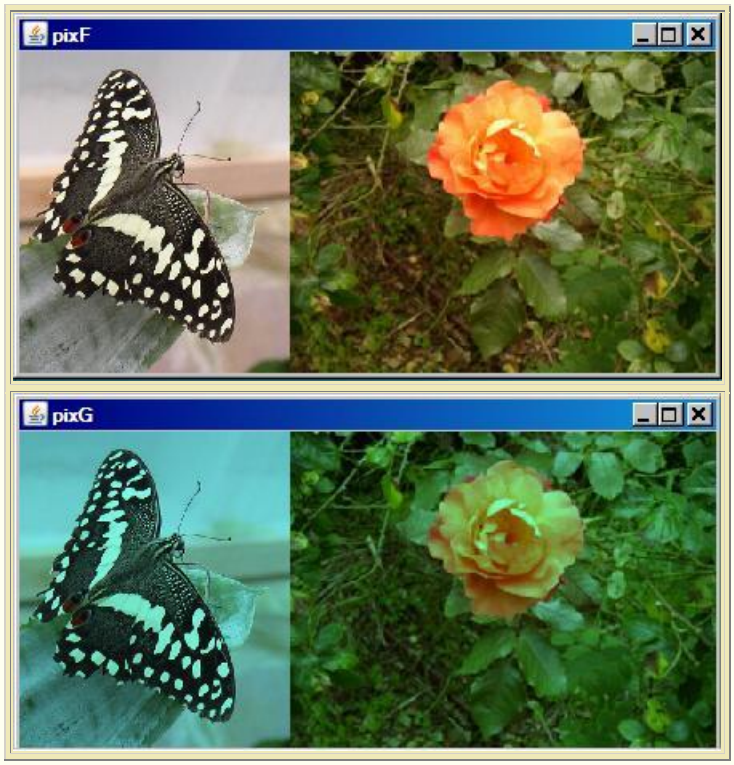
It is important to understand that there is a major difference between Figure 3 on one hand and Figure 2 on the other. The pairing of two pictures in Figure 2 was accomplished at the HTML level using two screen-shot image files. In other words, in the case of Figure 2, there were actually two **Picture** objects that produced two screen output images, which I captured with screen capture software. Each image in Figure 2 is in a different image file.

However, the images of the butterfly and the rose in Figure 3 were placed in the same **Picture** object by program code. The screen display of that **Picture** object was captured into a single image file, which was inserted into this document to produce Figure 3.

Call the getPixels method

Finally, the program uses the **getPixels** method to create a new picture of the butterfly and the rose side-by-side with the value of the red color component reduced by a factor of two as shown by the bottom picture in Figure 4. *(The top picture in Figure 4 is the same image file shown in Figure 3.)*

Figure 4. Picture with red color component reduced by a factor of two.



The **getPixels** method is very useful when you want to perform the same operation on every pixel in a **Picture** object.

Will explain the code in fragments

As is my custom, I will explain the program code in fragments. *(A complete listing of the program named Java360a is provided in Listing 19 near the end of the lesson.)*

The first such code fragment, which shows the beginning of the program named Java360a, is shown in Listing 3. *(Remember, the background color in Listing 3 indicates that the code fragment was extracted from my sample program named Java360a.)*

Listing 3. Beginning of the program named Java360a.

```
import java.awt.Graphics2D;
import java.awt.Color;
import java.awt.geom.AffineTransform;

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

Listing 3 simply defines an object of a new class named **Runner** and calls the **run** method on that object. When the **run** method returns, the **main** method will terminate causing the program to terminate.

Beginning of the run method of the Runner class

The **run** method of the **Runner** class begins in Listing 4.

Listing 4. Beginning of the run method of the Runner class.

```
class Runner{
  void run(){

    //The following code will load and show
the rose with
    // a title of "None"
    Picture pixA = new Picture(1,1);
    pixA.loadPictureAndShowIt("rose.jpg");
```

The loadPictureAndShowIt method

Listing 4 begins by creating a new **Picture** object with 1x1 pixels. Then it calls the **loadPictureAndShowIt** method to load an image from a specific image file into the **Picture** object and to display the resulting **Picture** object on the screen.

The **loadPictureAndShowIt** method of the **SimplePicture** class is shown in its entirety in Listing 5. *(Remember, the background color in Listing 5 indicates that the code fragment was extracted from Ericson's **SimplePicture** class.)*

Listing 5. The loadPictureAndShowIt method of the SimplePicture class.

```
 /***
  * Method to load a picture from a file name
and show it
  * in a picture frame
  * @param fileName the file name to load the
picture
  * from
  * @return true if success else false
```

```
  */
 public boolean loadPictureAndShowIt(String
fileName){
   boolean result = true;// the default is
that it worked

   // try to load the picture into the
buffered image from
   // the file name
   result = load(fileName);

   // show the picture in a picture frame
   show();

   return result;
 }//end loadPictureAndShowIt method
```

Call the load method

The method in Listing 5 begins by calling the **load** method to load the image from the image file into the current **Picture** object. *(I explained the **load** method in the earlier lesson titled The DigitalPicture Interface: Multimedia Programming with Java. See Resources).*

Call the show method

Then the method in Listing 5 calls the **show** method to display the picture on the screen. *(I explained the **show** method in the earlier lesson titled The show Method and the PictureFrame Class: Multimedia Programming with Java. See Resources).*

No further explanation needed

Since I have already explained the methods that are called in Listing 5, no further explanation of the **loadPictureAndShowIt** method of the **SimplePicture** class should be needed.

The onscreen image produced by the call to the **loadPictureAndShowIt** method in Listing 4 is shown as the top picture in Figure 1.

Create a picture with a butterfly image

Listing 6 uses code that you have seen in numerous previous programs to create a new **Picture** object containing an image of a butterfly.

Listing 6. Create a picture with a butterfly image.

```
    //Create a picture of a butterfly, which
is much
    // larger than the picture of the rose.
```

```
    Picture pixB = new
Picture("butterfly1.jpg");
    pixB.setTitle("pixB");
    pixB.show();
```

I purposely chose an image that was larger than the picture of the rose in order to demonstrate size control in some of the later code. The onscreen image produced by Listing 6 is shown as the bottom picture in Figure 1.

The getPictureWithWidth method

Listing 7 begins by calling the **getPictureWithWidth** method to create a new **Picture** object containing the butterfly image. The width of the new picture is set to match the width of the picture of the rose. Note that the aspect ratio of the butterfly image is preserved throughout this process.

Listing 7. Create a new width-controlled picture of the butterfly.

```
    Picture pixC =

pixB.getPictureWithWidth(pixA.getWidth());
    pixC.setTitle("pixC");

    //Draw white text on the picture of the
butterfly.
    pixC.drawString("Same width as
rose.",20,20);
    pixC.show();
```

Source code for the getPictureWithWidth method

The **getPictureWithWidth** method is shown in its entirety in Listing 8.

Listing 8. The getPictureWithWidth method of the SimplePicture class.

```
  /**
   * Method to create a new picture of the
passed width.
   * The aspect ratio of the width and height
will stay
   * the same.
   * @param width the desired width
   * @return the resulting picture
   */
  public Picture getPictureWithWidth(int
width){
    // set up the scale transform
    double xFactor = (double) width /
this.getWidth();
```

```
    Picture result = scale(xFactor,xFactor);
    return result;
  }//end getPictureWithWidth method
```

A scaled replica of the current Picture object

This method receives an integer value that specifies the desired width in pixels of a new **Picture** object that is a scaled replica of the current **Picture** object. In this case, the specified width was set to the width of the picture of the rose *(see Listing 7)*.

Call the scale method

Listing 8 computes a scale factor that must be applied to the current picture to produce a new picture with the specified width. Then Listing 8 calls the **scale** method, passing the same value for both the x and y scale factors required by the **scale** method. *(I explained the **scale** method in the previous lesson titled Applying Affine Transforms to Picture Objects: Multimedia Programming with Java. See Resources.)*

The **scale** method creates and returns a reference to a properly scaled replica of the current **Picture** object, which is saved and then returned by the code in Listing 8.

The drawString method

Please return your attention to Listing 7, which calls the **drawString** method on the new scaled picture of the butterfly to draw some text on the picture.

Source code for the drawString method

The **drawString** method of the **SimplePicture** class is shown in Listing 9.

Listing 9. The drawString method of the SimplePicture class.

```
 /**
  * Method to draw a string at the given
location on the
  * picture
  * @param text the text to draw
  * @param xPos the left x for the text
  * @param yPos the top y for the text
  */
 public void drawString(String text, int xPos,
int yPos){
   addMessage(text,xPos,yPos);
 }//end drawString method
```

As you can see, this method contains a single statement, which calls the method named **addMessage** to do all the work.

The addMessage method

The **addMessage** method is shown in its entirety in Listing 10.

Listing 10. The addMessage method of the SimplePicture class.

```
 /**
  * Method to draw a message as a string on
the buffered
  * image
  * @param message the message to draw on the
buffered
  * image
  * @param xPos  the leftmost point of the
string in x
  * @param yPos  the bottom of the string in y
  */
 public void addMessage(
                      String message, int
xPos, int yPos){
   // get a graphics context to use to draw on
the
   // buffered image
   Graphics2D graphics2d =
bufferedImage.createGraphics();

   // set the color to white
   graphics2d.setPaint(Color.white);

   // set the font to Helvetica bold style and
size 16
   graphics2d.setFont(new
Font("Helvetica",Font.BOLD,16));

   // draw the message
   graphics2d.drawString(message,xPos,yPos);

 }//end addMessage method
```

A relatively straightforward method

This method begins by getting a reference to the graphics context on the **BufferedImage** object belonging to the current **Picture** object as type **Graphics2D**.

Then it calls the following three methods of the **Graphics2D** class to perform the actions shown:

- **setPaint** - set the text color to white.
- **setFont** - set the font to the typeface, style, and size shown
- **drawString** - draw the text on the **BufferedImage** object at the position specified by the x and y coordinate values.

The Graphics2D class

If you are familiar with the use of the **Graphics2D** class, you should have no problem understanding the code in Listing 10. If not, you may want to go back and study the topic. *(See the links to my Graphics2D lessons in Resources.)*

Call the show method

The call to the **drawString** method followed by the call to the **show** method in Listing 7 produced the width-controlled picture of the butterfly with white text shown as the bottom picture in Figure 2. As explained above, the width of the top picture of the rose in Figure 2 was used to specify the width of the new picture of the butterfly.

The getPictureWithHeight method

Listing 11 makes two consecutive calls to the **getPictureWithHeight** method to create new pictures of the rose and the butterfly having the same height while preserving the aspect ratio of each picture.

Listing 11. Create new pictures of the rose and the butterfly with the same height.

```
    Picture pixD =
pixA.getPictureWithHeight(200);
    Picture pixE =
pixB.getPictureWithHeight(200);
```

Very similar code as before

Code in the **getPictureWithHeight** method is very similar to the code in the **getPictureWithWidth** method that I explained earlier. Therefore, it shouldn't be necessary to provide another explanation of the code. *(You can view the **getPictureWithHeight** method in its entirety in Listing 17.)*

The translatePicture method

Listing 12 calls the **translatePicture** method that I explained in the previous lesson *(see Resources)* to copy the picture of the rose into the right side of a new **Picture** object.

Listing 12. Copy the picture of the rose into the right side of a new picture.

```
    Picture pixF =

translatePicture(pixD,pixE.getWidth(),0);
```

This produced the picture that is shown in Figure 3, except that the butterfly is not yet a part of the picture.

The copyPicture method

Listing 13 calls the **copyPicture** method to copy the picture of the butterfly into the picture already containing the image of the rose. *(See Figure 3.)*

Listing 13. Copy the butterfly into the left side of the picture with the rose.

```
    pixF.copyPicture(pixE);
    pixF.setTitle("pixF");
    pixF.show();
```

When the **copyPicture** method is used to copy one picture into another, it always aligns the two pictures at the upper-left corner. I also explained the **copyPicture** method in the previous lesson. *(See Resources.)*

This results in a **Picture** object containing side-by-side images of the butterfly and the rose, both with the same height as shown in Figure 3.

The getPixels method

That brings us to the last method of the **SimplePicture** class that I will explain in this lesson. The **getPixels** method, shown in Listing 14, constructs, populates, and returns a reference to a one-dimensional array of type **Pixel[]**.

Listing 14. The getPixels method of the SimplePicture class.

```
 /**
  * Method to get a one-dimensional array of
Pixels for
  * this simple picture
  * @return a one-dimensional array of Pixel
objects
  * starting with y=0
  * to y=height-1 and x=0 to x=width-1.
  */
 public Pixel[] getPixels(){
   int width = getWidth();
   int height = getHeight();
   Pixel[] pixelArray = new Pixel[width *
height];

   // loop through height rows from top to
bottom
   for (int row = 0; row < height; row++)
     for (int col = 0; col < width; col++)
       pixelArray[row * width + col] =
                                    new
```

```
Pixel(this,col,row);

   return pixelArray;
 }//end getPixels method
```

A description of the array contents

Each element in the returned array contains a reference to a **Pixel** object. Each **Pixel** object represents one of the physical pixels in the current picture.

The first element in the array represents the single pixel in the upper-left corner of the picture. Successive elements represent adjacent pixels moving from left to right across the row.

When the number of elements reaches the width of the picture, the next element represents the left-most pixel in the second row of pixels, etc.

A useful representation in some cases

This representation of pixels is very useful when you need to perform the same operation on every pixel in the picture. It allows for the use of a single **for** loop to access and operate on each pixel. It also eliminates the requirement to know the width and the height of the picture as would be the case for nested **for** loops.

Not so handy in other cases

On the other hand, this representation isn't so handy when you need to keep track of the location of the individual pixels in terms of rows and columns. You have seen code in earlier lessons *(see Resources)* involving a pair of nested **for** loops that is easier to use in those cases.

Reduce the red color component value by a factor of two

Listing 15 begins by creating a new **Picture** object that is a copy of the picture shown in Figure 3.

Listing 15. Reduce the red color component value by a factor of two.

```
    Picture pixG = new Picture(pixF);
    pixG.setTitle("pixG");
    Pixel[] pixels = pixG.getPixels();
    int red = 0;
    for(int cnt = 0;cnt <
pixels.length;cnt++){
      red = pixels[cnt].getRed();
      pixels[cnt].setRed((int)(red*0.5));
    }//end for loop
```

```
    pixG.show();

  }//end run method
```

Get a reference to an array containing pixel data

Then Listing 15 calls the **getPixels** method on the new **Picture** object to get a reference to a one-dimensional array containing pixel data as described above.

Iterate on the array modifying the pixel values

Then Listing 15 executes a **for** loop in which the value of the red color component in each pixel is reduced by a factor of two.

Call the show method

Finally, Listing 15 shows the modified picture, producing the screen output shown in the bottom picture in Figure 4.

The end of the run method and the end of the program

Listing 14 also signals the end of the **run** method, causing the **run** method to return control to the **main** method shown in Listing 3. Having nothing more to do, the **main** method terminates, which causes the program to terminate.

Run the program

I encourage you to copy the code from Listing 19, compile the code, and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

I explained and illustrated the following six methods from the **SimplePicture** class in this lesson:

- Picture **getPictureWithHeight**(int height)
- Picture **getPictureWithWidth**(int width)
- Pixel[] **getPixels**()
- void **addMessage**(String message, int xPos, int yPos)
- void **drawString**(String text, int xPos, int yPos)
- boolean **loadPictureAndShowIt**(String fileName)

You learned how to:

- Cause two or more pictures to have the same width or the same height and otherwise maintain their individual aspect ratios.
- Create composite pictures containing side-by-side images.
- Draw text on a picture.
- Apply the same operation to every pixel in a **Picture** object.

What's next?

As I mentioned earlier, with the exception of the **explore** method, this completes the explanation of the **SimplePicture** class. I will defer an explanation of the **explore** method until the next lesson when I begin explaining the **PictureExplorer** class.

In the next and future lessons, you will learn about the **PictureExplorer** class, which allows you to determine the numeric color values for any pixel in a picture by placing a cursor on the pixel. The pixel position is controlled by clicking or dragging the mouse within the picture, clicking buttons, or typing coordinate values. You can also zoom in and out to view the pixel in more or less detail and you can see the actual color of the pixel in a large square.

Resources

- Creative Commons Attribution 3.0 United States License
- Media Computation book in Java - numerous downloads available
- Introduction to Computing and Programming with Java: A Multimedia Approach
- DrJava download site
- DrJava, the JavaPLT group at Rice University
- DrJava Open Source License
- The Essence of OOP using Java, The this and super Keywords
- Threads of Control
- Painting in AWT and Swing
- Wikipedia Turtle Graphics
- IsA or HasA
- Vector Cad-Cam XI Lathe Tutorial
- Classification of 3D to 2D projections
- Color model from Wikipedia
- Light and color: an introduction by Norman Koren
- Color Principles - Hue, Saturation, and Value
- 200 Implementing the Model-View-Controller Paradigm using Observer and Observable
- 300 Java 2D Graphics, Nested Top-Level Classes and Interfaces
- 302 Java 2D Graphics, The Point2D Class
- 304 Java 2D Graphics, The Graphics2D Class
- 306 Java 2D Graphics, Simple Affine Transforms
- 308 Java 2D Graphics, The Shape Interface, Part 1
- 310 Java 2D Graphics, The Shape Interface, Part 2

- 359 Creating a lasso for editing digital photos in Java

Complete program listings

Complete listings of the programs discussed in this lesson are shown in Listing 16 through Listing 19 below.

Listing 16. Source code for Ericson's Picture class.

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.image.BufferedImage;
import java.text.*;

/**
 * A class that represents a picture.  This
class inherits
 * from SimplePicture and allows the student
to add
 * functionality to the Picture class.
 *
 * Copyright Georgia Institute of Technology
2004-2005
 * @author Barbara Ericson
ericson@cc.gatech.edu
 */
public class Picture extends SimplePicture
{
  ///////////////////// constructors
//////////////////////

  /**
   * Constructor that takes no arguments
   */
  public Picture ()
  {
    /* not needed but use it to show students
the implicit
     * call to super()
     * child constructors always call a parent
constructor
     */
    super();
  }

  /**
   * Constructor that takes a file name and
creates the
   * picture
   * @param fileName the name of the file to
create the
   * picture from
```

```
   */
  public Picture(String fileName)
  {
    // let the parent class handle this
fileName
    super(fileName);
  }

  /**
   * Constructor that takes the width and
height
   * @param width the width of the desired
picture
   * @param height the height of the desired
picture
   */
  public Picture(int width, int height)
  {
    // let the parent class handle this width
and height
    super(width,height);
  }

  /**
   * Constructor that takes a picture and
creates a
   * copy of that picture
   */
  public Picture(Picture copyPicture)
  {
    // let the parent class do the copy
    super(copyPicture);
  }

  /**
   * Constructor that takes a buffered image
   * @param image the buffered image to use
   */
  public Picture(BufferedImage image)
  {
    super(image);
  }

  ////////////////////// methods
/////////////////////////

  /**
   * Method to return a string with
information about this
   * picture.
   * @return a string with information about
the picture
   * such as fileName, height and width.
   */
  public String toString()
  {
```

```
    String output =
      "Picture, filename " + getFileName() +
      " height " + getHeight()
      + " width " + getWidth();
    return output;

  }

} // this } is the end of class Picture, put
all new
  // methods before this
```

Listing 17. Source code for Ericson's SimplePicture class.

```
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import javax.swing.ImageIcon;
import java.awt.*;
import java.io.*;
import java.awt.geom.*;

/**
 * A class that represents a simple picture.  A
simple
 * picture may have an associated file name and a
title.
 * A simple picture has pixels, width, and height.
A
 * simple picture uses a BufferedImage to hold the
pixels.
 * You can show a simple picture in a PictureFrame
(a
 * JFrame).
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimplePicture implements
DigitalPicture
{

  ////////////////////// Fields
//////////////////////////

  /**
   * the file name associated with the simple
picture
   */
  private String fileName;

  /**
   * the title of the simple picture
```

```
   */
  private String title;

  /**
   * buffered image to hold pixels for the simple
picture
   */
  private BufferedImage bufferedImage;

  /**
   * frame used to display the simple picture
   */
  private PictureFrame pictureFrame;

  /**
   * extension for this file (jpg or bmp)
   */
  private String extension;


 ////////////////////// Constructors
//////////////////

 /**
  * A Constructor that takes no arguments.  All
fields
  * will be null. A no-argument constructor must
be given
  * in order for a class to be able to be
subclassed.  By
  * default all subclasses will implicitly call
this in
  * their parent's no argument constructor unless
a
  * different call to super() is explicitly made
as the
  * first line of code in a constructor.
  */
 public SimplePicture()
 {this(200,100);}

 /**
  * A Constructor that takes a file name and uses
the
  * file to create a picture
  * @param fileName the file name to use in
creating the
  * picture
  */
 public SimplePicture(String fileName)
 {

   // load the picture into the buffered image
   load(fileName);

 }
```

```
 /**
  * A constructor that takes the width and height
desired
  * for a picture and creates a buffered image of
that
  * size.  This constructor doesn't  show the
picture.
  * @param width the desired width
  * @param height the desired height
  */
 public  SimplePicture(int width, int height)
 {
   bufferedImage = new BufferedImage(
               width, height,
BufferedImage.TYPE_INT_RGB);
   title = "None";
   fileName = "None";
   extension = "jpg";
   setAllPixelsToAColor(Color.white);
 }

 /**
  * A constructor that takes the width and height
desired
  * for a picture and creates a buffered image of
that
  * size.  It also takes the color to use for the
  * background of the picture.
  * @param width the desired width
  * @param height the desired height
  * @param theColor the background color for the
picture
  */
 public  SimplePicture(
                   int width, int height, Color
theColor)
 {
   this(width,height);
   setAllPixelsToAColor(theColor);
 }

 /**
  * A Constructor that takes a picture to copy
  * information from
  * @param copyPicture the picture to copy from
  */
 public SimplePicture(SimplePicture copyPicture)
 {
   if (copyPicture.fileName != null)
   {
      this.fileName = new
String(copyPicture.fileName);
      this.extension = copyPicture.extension;
   }
   if (copyPicture.title != null)
```

```
      this.title = new String(copyPicture.title);
   if (copyPicture.bufferedImage != null)
   {
     this.bufferedImage =
                  new
BufferedImage(copyPicture.getWidth(),

copyPicture.getHeight(),

BufferedImage.TYPE_INT_RGB);
     this.copyPicture(copyPicture);
   }
 }

 /**
  * A constructor that takes a buffered image
  * @param image the buffered image
  */
 public SimplePicture(BufferedImage image)
 {
   this.bufferedImage = image;
   title = "None";
   fileName = "None";
   extension = "jpg";
 }

 ////////////////////////// Methods
//////////////////////

 /**
  * Method to get the extension for this picture
  * @return the extendsion (jpg or bmp)
  */
 public String getExtension() { return extension;
}


 /**
  * Method that will copy all of the passed source
  * picture into the current picture object
  * @param sourcePicture  the picture object to
copy
  */
 public void copyPicture(SimplePicture
sourcePicture)
 {
   Pixel sourcePixel = null;
   Pixel targetPixel = null;

   // loop through the columns
   for (int sourceX = 0, targetX = 0;
        sourceX < sourcePicture.getWidth() &&
        targetX < this.getWidth();
        sourceX++, targetX++)
   {
     // loop through the rows
```

```
     for (int sourceY = 0, targetY = 0;
          sourceY < sourcePicture.getHeight() &&
          targetY < this.getHeight();
          sourceY++, targetY++)
     {
       sourcePixel =

sourcePicture.getPixel(sourceX,sourceY);
       targetPixel =
this.getPixel(targetX,targetY);

targetPixel.setColor(sourcePixel.getColor());
     }
   }

 }

 /**
  * Method to set the color in the picture to the
passed
  * color
  * @param color the color to set to
  */
 public void setAllPixelsToAColor(Color color)
 {
   // loop through all x
   for (int x = 0; x < this.getWidth(); x++)
   {
     // loop through all y
     for (int y = 0; y < this.getHeight(); y++)
     {
       getPixel(x,y).setColor(color);
     }
   }
 }

 /**
  * Method to get the buffered image
  * @return the buffered image
  */
 public BufferedImage getBufferedImage()
 {
    return bufferedImage;
 }

 /**
  * Method to get a graphics object for this
picture to
  * use to draw on
  * @return a graphics object to use for drawing
  */
 public Graphics getGraphics()
 {
   return bufferedImage.getGraphics();
 }
```

```
 /**
  * Method to get a Graphics2D object for this
picture
  * which can be used to do 2D drawing on the
picture
  */
 public Graphics2D createGraphics()
 {
   return bufferedImage.createGraphics();
 }

 /**
  * Method to get the file name associated with
the
  * picture
  * @return  the file name associated with the
picture
  */
 public String getFileName() { return fileName; }

 /**
  * Method to set the file name
  * @param name the full pathname of the file
  */
 public void setFileName(String name)
 {
   fileName = name;
 }

 /**
  * Method to get the title of the picture
  * @return the title of the picture
  */
 public String getTitle()
 { return title; }

 /**
  * Method to set the title for the picture
  * @param title the title to use for the picture
  */
 public void setTitle(String title)
 {
   this.title = title;
   if (pictureFrame != null)
       pictureFrame.setTitle(title);
 }

 /**
  * Method to get the width of the picture in
pixels
  * @return the width of the picture in pixels
  */
 public int getWidth(){ return
bufferedImage.getWidth(); }

 /**
```

```
  * Method to get the height of the picture in
pixels
  * @return  the height of the picture in pixels
  */
 public int getHeight(){
  return bufferedImage.getHeight();
 }

 /**
  * Method to get the picture frame for the
picture
  * @return the picture frame associated with this
  * picture (it may be null)
  */
 public PictureFrame getPictureFrame()
                                    { return
pictureFrame; }

 /**
  * Method to set the picture frame for this
picture
  * @param pictureFrame the picture frame to use
  */
 public void setPictureFrame(PictureFrame
pictureFrame)
 {
   // set this picture objects' picture frame to
the
   // passed one
   this.pictureFrame = pictureFrame;
 }

 /**
  * Method to get an image from the picture
  * @return  the buffered image since it is an
image
  */
 public Image getImage()
 {
   return bufferedImage;
 }

 /**
  * Method to return the pixel value as an int for
the
  * given x and y location
  * @param x the x coordinate of the pixel
  * @param y the y coordinate of the pixel
  * @return the pixel value as an integer (alpha,
red,
  * green, blue)
  */
 public int getBasicPixel(int x, int y)
 {
    return bufferedImage.getRGB(x,y);
 }
```

```
 /**
  * Method to set the value of a pixel in the
picture
  * from an int
  * @param x the x coordinate of the pixel
  * @param y the y coordinate of the pixel
  * @param rgb the new rgb value of the pixel
(alpha, red,
  * green, blue)
  */
 public void setBasicPixel(int x, int y, int rgb)
 {
   bufferedImage.setRGB(x,y,rgb);
 }

 /**
  * Method to get a pixel object for the given x
and y
  * location
  * @param x  the x location of the pixel in the
picture
  * @param y  the y location of the pixel in the
picture
  * @return a Pixel object for this location
  */
 public Pixel getPixel(int x, int y)
 {
   // create the pixel object for this picture and
the
   // given x and y location
   Pixel pixel = new Pixel(this,x,y);
   return pixel;
 }

 /**
  * Method to get a one-dimensional array of
Pixels for
  * this simple picture
  * @return a one-dimensional array of Pixel
objects
  * starting with y=0
  * to y=height-1 and x=0 to x=width-1.
  */
 public Pixel[] getPixels()
 {
   int width = getWidth();
   int height = getHeight();
   Pixel[] pixelArray = new Pixel[width * height];

   // loop through height rows from top to bottom
   for (int row = 0; row < height; row++)
     for (int col = 0; col < width; col++)
       pixelArray[row * width + col] =
                                    new
Pixel(this,col,row);
```

```
   return pixelArray;
 }



 /**
  * Method to load the buffered image with the
passed
  * image
  * @param image  the image to use
  */
 public void load(Image image)
 {
   // get a graphics context to use to draw on the
   // buffered image
   Graphics2D graphics2d =
bufferedImage.createGraphics();

   // draw the image on the buffered image
starting
   // at 0,0
   graphics2d.drawImage(image,0,0,null);

   // show the new image
   show();
 }

 /**
  * Method to show the picture in a picture frame
  */
 public void show()
 {
    // if there is a current picture frame then
use it
   if (pictureFrame != null)
     pictureFrame.updateImageAndShowIt();

   // else create a new picture frame with this
picture
   else
     pictureFrame = new PictureFrame(this);
 }

 /**
  * Method to hide the picture
  */
 public void hide()
 {
   if (pictureFrame != null)
     pictureFrame.setVisible(false);
 }

 /**
  * Method to make this picture visible or not
  * @param flag true if you want it visible else
```

```
false
  */
 public void setVisible(boolean flag)
 {
   if (flag)
     this.show();
   else
     this.hide();
 }

 /**
  * Method to open a picture explorer on a copy of
this
  * simple picture
  */
 public void explore()
 {
   // create a copy of the current picture and
explore it
   new PictureExplorer(new SimplePicture(this));
 }

 /**
  * Method to force the picture to redraw itself.
This is
  * very useful after you have changed the pixels
in a
  * picture.
  */
 public void repaint()
 {
   // if there is a picture frame tell it to
repaint
   if (pictureFrame != null)
     pictureFrame.repaint();

   // else create a new picture frame
   else
     pictureFrame = new PictureFrame(this);
 }

 /**
  * Method to load the picture from the passed
file name
  * @param fileName the file name to use to load
the
  * picture from
  */
 public void loadOrFail(
                      String fileName) throws
IOException
 {
    // set the current picture's file name
   this.fileName = fileName;

   // set the extension
```

```
   int posDot = fileName.indexOf('.');
   if (posDot >= 0)
     this.extension = fileName.substring(posDot +
1);

   // if the current title is null use the file
name
   if (title == null)
     title = fileName;

   File file = new File(this.fileName);

   if (!file.canRead())
   {
     // try adding the media path
     file = new File(

FileChooser.getMediaPath(this.fileName));
     if (!file.canRead())
     {
       throw new IOException(this.fileName + "
could not"
       + " be opened. Check that you specified the
path");
     }
   }

   bufferedImage = ImageIO.read(file);
 }


 /**
  * Method to write the contents of the picture to
a file
  * with the passed name without throwing errors
  * (THIS MAY NOT BE A VALID DESCRIPTION - RGB)
  * @param fileName the name of the file to write
the
  * picture to
  * @return true if success else false
  */
 public boolean load(String fileName)
 {
     try {
         this.loadOrFail(fileName);
         return true;

     } catch (Exception ex) {
         System.out.println("There was an error
trying"
                                + " to open " +
fileName);
         bufferedImage = new
BufferedImage(600,200,

BufferedImage.TYPE_INT_RGB);
```

```
         addMessage("Couldn't load " +
fileName,5,100);
         return false;
     }

 }


 /**
  * Method to load the picture from the passed
file name
  * this just calls load(fileName) and is for name
  * compatibility
  * @param fileName the file name to use to load
the
  * picture from
  * @return true if success else false
  */
 public boolean loadImage(String fileName)
 {
     return load(fileName);
}

 /**
  * Method to draw a message as a string on the
buffered
  * image
  * @param message the message to draw on the
buffered
  * image
  * @param xPos  the leftmost point of the string
in x
  * @param yPos  the bottom of the string in y
  */
 public void addMessage(
                      String message, int xPos,
int yPos)
 {
   // get a graphics context to use to draw on the
   // buffered image
   Graphics2D graphics2d =
bufferedImage.createGraphics();

   // set the color to white
   graphics2d.setPaint(Color.white);

   // set the font to Helvetica bold style and
size 16
   graphics2d.setFont(new
Font("Helvetica",Font.BOLD,16));

   // draw the message
   graphics2d.drawString(message,xPos,yPos);

 }
```

```
 /**
  * Method to draw a string at the given location
on the
  * picture
  * @param text the text to draw
  * @param xPos the left x for the text
  * @param yPos the top y for the text
  */
 public void drawString(String text, int xPos, int
yPos)
 {
   addMessage(text,xPos,yPos);
 }

 /**
   * Method to create a new picture by scaling the
   * current picture by the given x and y factors
   * @param xFactor the amount to scale in x
   * @param yFactor the amount to scale in y
   * @return the resulting picture
   */
  public Picture scale(double xFactor, double
yFactor)
  {
    // set up the scale transform
    AffineTransform scaleTransform =
                                      new
AffineTransform();
    scaleTransform.scale(xFactor,yFactor);

    // create a new picture object that is the
right size
    Picture result = new Picture(
                           (int) (getWidth() *
xFactor),
                           (int) (getHeight() *
yFactor));

    // get the graphics 2d object to draw on the
result
    Graphics graphics = result.getGraphics();
    Graphics2D g2 = (Graphics2D) graphics;

    // draw the current image onto the result
image
    // scaled

g2.drawImage(this.getImage(),scaleTransform,null);

    return result;
  }

  /**
   * Method to create a new picture of the passed
width.
   * The aspect ratio of the width and height will
```

```
stay
   * the same.
   * @param width the desired width
   * @return the resulting picture
   */
  public Picture getPictureWithWidth(int width)
  {
    // set up the scale transform
    double xFactor = (double) width /
this.getWidth();
    Picture result = scale(xFactor,xFactor);
    return result;
  }

  /**
   * Method to create a new picture of the passed
height.
   * The aspect ratio of the width and height will
stay
   * the same.
   * @param height the desired height
   * @return the resulting picture
   */
  public Picture getPictureWithHeight(int height)
  {
    // set up the scale transform
    double yFactor = (double) height /
this.getHeight();
    Picture result = scale(yFactor,yFactor);
    return result;
  }

 /**
  * Method to load a picture from a file name and
show it
  * in a picture frame
  * @param fileName the file name to load the
picture
  * from
  * @return true if success else false
  */
 public boolean loadPictureAndShowIt(String
fileName)
 {
   boolean result = true;// the default is that it
worked

   // try to load the picture into the buffered
image from
   // the file name
   result = load(fileName);

   // show the picture in a picture frame
   show();

   return result;
```

```
 }

 /**
  * Method to write the contents of the picture to
a file
  * with the passed name
  * @param fileName the name of the file to write
the
  * picture to
  */
 public void writeOrFail(String fileName)
                                          throws
IOException
 {
   //the default is current
   String extension = this.extension;

   // create the file object
   File file = new File(fileName);
   File fileLoc = file.getParentFile();

   // canWrite is true only when the file exists
   // already! (alexr)
   if (!fileLoc.canWrite()) {
       // System.err.println(
       // "can't write the file but trying anyway?
...");
        throw new IOException(fileName +
        " could not be opened. Check to see if you
can"
        + " write to the directory.");
   }

   // get the extension
   int posDot = fileName.indexOf('.');
   if (posDot >= 0)
       extension = fileName.substring(posDot + 1);

   //write the contents of the buffered image to
the file
   // as jpeg
   ImageIO.write(bufferedImage, extension, file);

 }

 /**
  * Method to write the contents of the picture to
a file
  * with the passed name without throwing errors
  * @param fileName the name of the file to write
the
  * picture to
  * @return true if success else false
  */
 public boolean write(String fileName)
 {
```

```
     try {
         this.writeOrFail(fileName);
         return true;
     } catch (Exception ex) {
         System.out.println(
                     "There was an error trying to
write "
                     + fileName);
         return false;
     }

 }

 /**
  * Method to set the media path by setting the
directory
  * to use
  * @param directory the directory to use for the
media
  * path
  */
 public static void setMediaPath(String directory)
{
   FileChooser.setMediaPath(directory);
 }

 /**
  * Method to get the directory for the media
  * @param fileName the base file name to use
  * @return the full path name by appending
  * the file name to the media directory
  */
 public static String getMediaPath(String
fileName) {
   return FileChooser.getMediaPath(fileName);
 }

  /**
   * Method to get the coordinates of the
enclosing
   * rectangle after this transformation is
applied to
   * the current picture
   * @return the enclosing rectangle
   */
  public Rectangle2D getTransformEnclosingRect(

AffineTransform trans)
  {
    int width = getWidth();
    int height = getHeight();
    double maxX = width - 1;
    double maxY = height - 1;
    double minX, minY;
    Point2D.Double p1 = new Point2D.Double(0,0);
    Point2D.Double p2 = new
```

```
Point2D.Double(maxX,0);
    Point2D.Double p3 = new
Point2D.Double(maxX,maxY);
    Point2D.Double p4 = new
Point2D.Double(0,maxY);
    Point2D.Double result = new
Point2D.Double(0,0);
    Rectangle2D.Double rect = null;

    // get the new points and min x and y and max
x and y
    trans.deltaTransform(p1,result);
    minX = result.getX();
    maxX = result.getX();
    minY = result.getY();
    maxY = result.getY();
    trans.deltaTransform(p2,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p3,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p4,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());

    // create the bounding rectangle to return
    rect = new Rectangle2D.Double(
              minX,minY,maxX - minX + 1, maxY -
minY + 1);
    return rect;
  }

 /**
  * Method to return a string with information
about this
  * picture
  * @return a string with information about the
picture
  */
 public String toString()
 {
   String output =
     "Simple Picture, filename " + fileName +
     " height " + getHeight() + " width " +
getWidth();
   return output;
 }

} // end of SimplePicture class
```

Listing 18. Source code for Ericson's DigitalPicture interface.

```
import java.awt.Image;
import java.awt.image.BufferedImage;

/**
 * Interface to describe a digital picture.  A
digital
 * picture can have a associated file name.
It can have
 * a title.  It has pixels associated with it
and you can
 * get and set the pixels.  You can get an
Image from a
 * picture or a BufferedImage.  You can load
it from a
 * file name or image.  You can show a
picture.  You can
 * create a new image for it.
 *
 * Copyright Georgia Institute of Technology
2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public interface DigitalPicture
{
 // get the file name that the picture came
from
 public String getFileName();

 // get the title of the picture
 public String getTitle();

 // set the title of the picture
 public void setTitle(String title);

 // get the width of the picture in pixels
 public int getWidth();

 // get the height of the picture in pixels
 public int getHeight();

 // get the image from the picture
 public Image getImage();

 // get the buffered image
 public BufferedImage getBufferedImage();

 // get the pixel information as an int
 public int getBasicPixel(int x, int y);

 // set the pixel information
 public void setBasicPixel(int x, int y, int
```

```
rgb);

 // get the pixel information as an object
 public Pixel getPixel(int x, int y);

 // load the image into the picture
 public void load(Image image);

 // load the picture from a file
 public boolean load(String fileName);

 // show the picture
 public void show();
}
```

Listing 19. Source code for the program named Java360a.

```
/*Program Java360a
Copyright R.G.Baldwin 2009

The purpose of this program is to illustrate the use of
the following methods of the Picture class:

boolean loadPictureAndShowIt(String fileName)
Picture getPictureWithWidth(int width)
void drawString(String text,int xPos,int yPos)
void addMessage(String message, int xPos, int yPos)
Picture getPictureWithHeight(int height)
Pixel[] getPixels()

An attempt was also made to illustrate the following
methods to write Picture objects into image files.
However, the results were very unreliable. Sometimes the
program was able to write the file and sometimes it
wasn't. Sometimes when the file was written, it would
contain the image and sometimes it would be empty.

boolean write(String fileName)
void writeOrFail(String fileName)throws IOException

This program completes the illustrations of the methods of
the Picture class with the exception of the explore
method. The explore method will be explained in a future
lesson that is dedicated to that method alone.

The program begins by calling the loadPictureAndShowIt
method to load and show a picture of a rose. The title
shown in the JFrame object is "None"

Then the program reads an image file to create a picture
of a butterfly, which is much  larger than the picture of
the rose.
```

```
Then the program calls the getPictureWithWidth method to
create a new Picture object containing the butterfly image
with the width being set to match the width of the picture
of the rose. Note that the aspect ratio of the butterfly
picture is preserved.

Then the program calls the drawString method to draw a
white text string on the picture of the butterfly. The
drawString method calls the addMessage method to actually
draw the text on the image. The color white is fixed and
cannot be changed without modifying the method.

Then the program calls the getPictureWithHeight method
twice to create pictures of the rose and the butterfly
with the same height. Again, the original aspect ratio of
each image is preserved.

Then the program calls the Baldwin method named
translatePicture to copy the picture of the rose into the
right side of a new Picture object. It calls the
copyPicture method to copy the picture of the butterfly
into the left side of the same picture. The pictures used
as input to this operation are the pictures with the same
height. This produces a new Picture object containing
side-by-side images of the butterfly and the rose.

Then the program uses the getPixels method to create a
new picture of the butterfly and the rose side-by-side
with the value of the red color component reduced by a
factor of two. This is a very useful approach when you
want to perform the same operation on every pixel in a
Picture object.

Tested using Windows Vista Premium Home edition and
Ericson's multimedia library.
**************************************************/
import java.awt.Graphics2D;
import java.awt.Color;
import java.awt.geom.AffineTransform;


public class Main{
  public static void main(String[] args){
    new Runner().run();
  }//end main method
}//end class Main
//-----------------------------------------------------//

class Runner{
  void run(){

    //The following code will load and show the rose with
    // a title of "None"
    Picture pixA = new Picture(1,1);
    pixA.loadPictureAndShowIt("rose.jpg");
```

```
  //Create a picture of a butterfly, which is much
  // larger than the picture of the rose.
  Picture pixB = new Picture("butterfly1.jpg");
  pixB.setTitle("pixB");
  pixB.show();

  //Create a new Picture object containing the butterfly
  // image. The width of the new picture is set to match
  // the width of the picture of the rose. Note that the
  // aspect ratio is preserved.
  Picture pixC =
              pixB.getPictureWithWidth(pixA.getWidth());
  pixC.setTitle("pixC");

  //Draw white text on the picture of the butterfly.
  pixC.drawString("Same width as rose.",20,20);
  pixC.show();

  //Create pictures of the rose and the butterfly with
  // the same height and their original aspect ratios.
  Picture pixD = pixA.getPictureWithHeight(200);
  Picture pixE = pixB.getPictureWithHeight(200);

  //Copy the picture of the rose into the right side of
  // a new picture.
  Picture pixF =
               translatePicture(pixD,pixE.getWidth(),0);

  //Copy the picture of the butterfly into the left side
  // of the same picture. This results in a Picture
  // object containing side-by-side images of the
  // butterfly and the rose, both with the same height.
  pixF.copyPicture(pixE);
  pixF.setTitle("pixF");
  pixF.show();

  //Create and show a new picture of the butterfly and
  // the rose side-by-side and reduce the value of the
  // red color component by a factor of two. The
  // getPixels method is very useful when you want to
  // perform the same operation on every pixel in a
  // picture.
  Picture pixG = new Picture(pixF);
  pixG.setTitle("pixG");
  Pixel[] pixels = pixG.getPixels();
  int red = 0;
  for(int cnt = 0;cnt < pixels.length;cnt++){
    red = pixels[cnt].getRed();
    pixels[cnt].setRed((int)(red*0.5));
  }//end for loop
  pixG.show();

}//end run method
//-----------------------------------------------//
```

```
  //The following method accepts a reference to a Picture
  // object along with positive x and y translation
  // values. It creates and returns a new Picture object
  // that contains a translated version of the original
  // image with whitespace to the left of and/or above the
  // translated image. If either translation value is
  // negative, the method simply returns a reference to a
  // copy of the original picture.
  public Picture translatePicture(
                        Picture pix,double tx,double ty){
    if((tx < 0.0) || (ty < 0.0)){
      //Negative translation values are not supported.
      // Simply return a reference to a copy of the
      // incoming picture. Note that this constructor
      // creates a new picture by copying the image from
      // an existing picture.
      return new Picture(pix);
    }//end if

    //Set up the transform
    AffineTransform translateTransform =
                                    new AffineTransform();
    translateTransform.translate(tx,ty);

    //Compute the size of a rectangle that is of
    // sufficient size to contain and display the
    // translated image.
    int pixWidth = pix.getWidth() + (int)tx;
    int pixHeight = pix.getHeight() + (int)ty;

    //Create a new picture object that is the correct
    // size.
    Picture result = new Picture(pixWidth,pixHeight);

    //Get the graphics2D object to draw on the result.
    Graphics2D g2 = (Graphics2D)result.getGraphics();

    //Draw the translated image from pix onto the new
    // Picture object, applying the transform in the
    // process.
    g2.drawImage(pix.getImage(),translateTransform,null);

    return result;
  }//end translatePicture
  //-----------------------------------------------------//

}//end class Runner
```

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is object-oriented programming using Java and other OOP languages.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

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@DickBaldwin.com

-end-