Using the Java 2D AffineTransformOp Filter Class to Process Images

Learn how to write programs that use the AffineTransformOp image-filtering class of the Java 2D API for a variety of image-processing purposes.

Published: April 10, 2007
By Richard G. Baldwin

Java Programming Notes # 454

Preface

Part of a series

In an earlier lesson entitled A Framework for Experimenting with Java 2D Image-Processing Filters, I taught you how to write a framework program that makes it easy to use the image-filtering classes of the Java 2D API to process the pixels in an image and to display the processed image.

In the previous lesson entitled Using the Java 2D LookupOp Filter Class to Process Images, I taught you how to write programs that use the **LookupOp** image-filtering class of the Java 2D API for a variety of image-processing purposes.

At the close of that lesson, I told you that future lessons would teach you how to use the following image-filtering classes from the Java 2D API:

- **AffineTransformOp**
- **BandCombineOp**

- **ConvolveOp**
- **RescaleOp**
- **ColorConvertOp**

In this lesson, I will keep that promise and teach you how to use the **AffineTransformOp** image-filtering class to perform a variety of transformations on images. I will teach you how to use the remaining classes from the above list in future lessons.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

I also recommend that you pay particular attention to the lessons listed in the References section of this document.

General Background Information

Constructing images

Before getting into the programming details, it may be useful for you to review the concept of how images are constructed, stored, transported, and rendered in Java *(and in most modern computer environments for that matter).*

I provided a great deal of information on those topics in the earlier lesson entitled Processing Image Pixels using Java, Getting Started. Therefore, I won't repeat that information here. Rather, I will simply refer you back to the earlier lesson.

The framework program named ImgMod05

It will also be useful for you to understand the behavior of the framework program named **ImgMod05**. Therefore, I strongly recommend that you study the earlier lesson entitled A Framework for Experimenting with Java 2D Image-Processing Filters.

However, if you don't have the time to do that, I can summarize that framework program as follows:

Purpose of ImgMod05

The purpose of **ImgMod05** is to make it easy for you to experiment with the modification of images using the image-filtering classes of the Java 2D API and to display the modified version of the image along with the original image. *(See an example of the graphic output format in Figure 5.)*

The Replot button

The **ImgMod05** program GUI contains a **Replot** button *(as shown in Figure 5)*. At the beginning of the run, and each time thereafter that the **Replot** button is clicked:

- The image-processing method belonging to an object of specified image-processing class is invoked.
- The original image is passed to the image-processing method, which returns a reference to a processed image.
- The resulting processed image is displayed along with the original image.
- The processed image is written into an output JPEG file named **junk.jpg**.

Display of the images

When the **ImgMod05** program is started, the original image and the processed version of the image are displayed in a frame with the original image above the processed image *(as shown in Figure 5)*. The program attempts to set the size of the display so as to accommodate both images. If both images are not totally visible, the user can manually resize the display frame.

Input and output file format

The **ImgMod05** program will read gif and jpg input files and possibly some other input file types as well. The output file is always a JPEG file.

Typical usage

Enter the following at the command-line to run the **ImgMod05** program:

```
java ImgMod05 ProcessingProgramName ImageFileName
```

Preview

In this lesson, I will present and explain an image-processing program named **ImgMod40** that is compatible with the framework program named **ImgMod05**. This program provides a GUI that allows the user to perform the following transforms on an input image:

- Scaling
- Translation
- Rotation

- Mirror Image

With the exception of the Mirror Image transform, all of the transforms in the above list allow the user to input important parameters via the GUI.

The program GUI is a **JTabbedPane** with four pages. Let's begin by taking a look at each of those pages.

Scaling

The *Scaling* page of the GUI is shown in Figure 1.

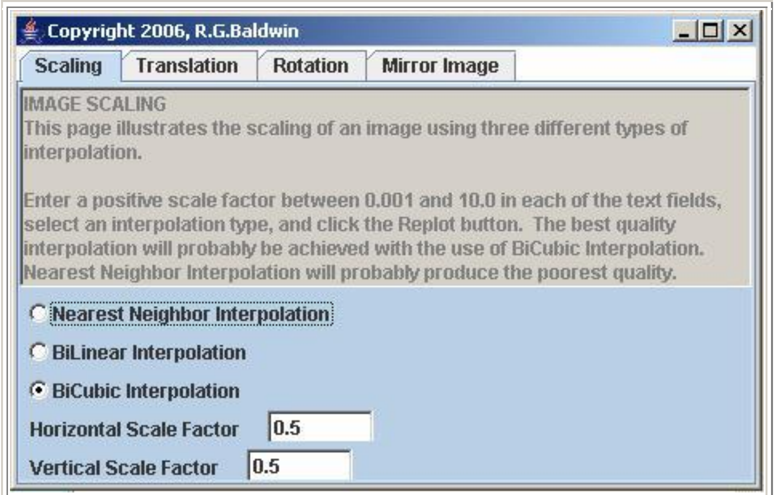


IMAGE SCALING
This page illustrates the scaling of an image using three different types of interpolation.

Enter a positive scale factor between 0.001 and 10.0 in each of the text fields, select an interpolation type, and click the Replot button. The best quality interpolation will probably be achieved with the use of BiCubic Interpolation. Nearest Neighbor Interpolation will probably produce the poorest quality.

Nearest Neighbor Interpolation [✓ BiCubic Interpolation] BiLinear Interpolation

Horizontal Scale Factor 0.5

Vertical Scale Factor 0.5

Figure 1

As you can see from Figure 1, the program allows the user to specify independent horizontal and vertical scale factors. In addition, the user is allowed to choose one of three optional interpolation schemes. *(I will have more to say about interpolation shortly.)*

Figure 6 shows an example of a scaled image using *bicubic* interpolation.

Translation

The *Translation* page from the GUI is shown in Figure 2.

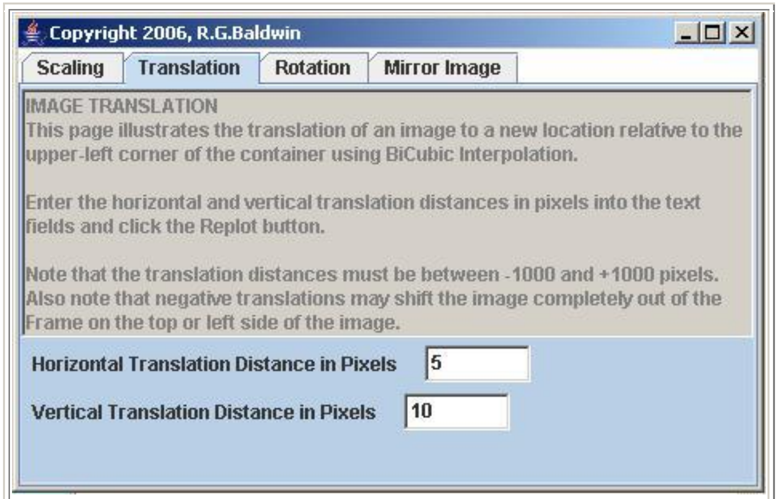


IMAGE TRANSLATION
This page illustrates the translation of an image to a new location relative to the upper-left corner of the container using BiCubic Interpolation.

Enter the horizontal and vertical translation distances in pixels into the text fields and click the Replot button.

Note that the translation distances must be between -1000 and +1000 pixels. Also note that negative translations may shift the image completely out of the Frame on the top or left side of the image.

Horizontal Translation Distance in Pixels 5

Vertical Translation Distance in Pixels 10

Figure 2

This program allows the user to translate the image to the right and down by specifying positive translation distances in pixels. Translation to the left and up can also be performed by specifying negative translation distances. However, translation to the left has the effect of chopping off part of the image on the left side, while translation up has the effect of chopping off part of the image at the top.

Figure 7 shows an example of an image that has been translated to the right and down.

Rotation

The *Rotation* page of the GUI is shown in Figure 3.

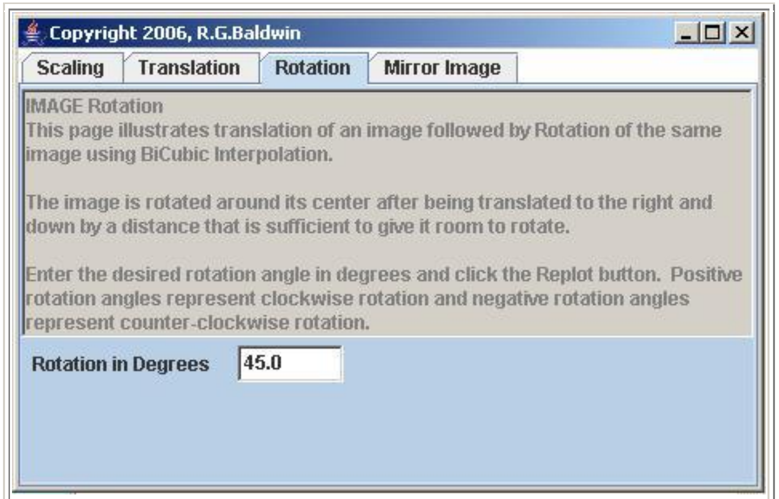


IMAGE Rotation
This page illustrates translation of an image followed by Rotation of the same image using BiCubic Interpolation.

The image is rotated around its center after being translated to the right and down by a distance that is sufficient to give it room to rotate.

Enter the desired rotation angle in degrees and click the Replot button. Positive rotation angles represent clockwise rotation and negative rotation angles represent counter-clockwise rotation.

Rotation in Degrees 45.0

Figure 3

Rotating the image.
Note that prior to performing the rotation, the image in Figure 8 was translated to the

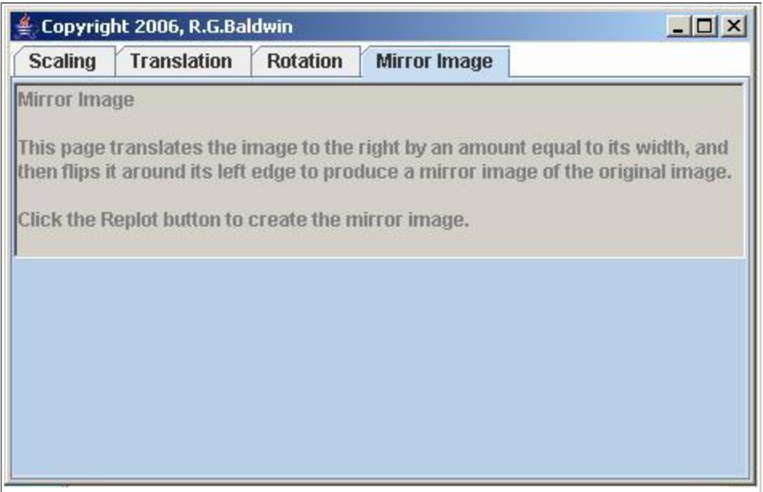
This program allows the user to rotate the image about its center by a positive or negative angle specified in degrees, where positive rotation is clockwise rotation.

An example of rotating an image is shown in Figure 8.

right and down a distance sufficient to make it possible to rotate it without chopping off its corners on the left and the top.

Mirror Image

The *Mirror Image* page in the GUI is shown in Figure 4.



Mirror Image

This page translates the image to the right by an amount equal to its width, and then flips it around its left edge to produce a mirror image of the original image.

Click the Replot button to create the mirror image.

Figure 4

The mirror image is created by translating the image to the right by a distance equal to its width, and then flipping it about its left edge. There are no user input parameters for this process. An example of creating the mirror image of an image is shown in Figure 9.

A very significant capability

In my opinion, *(with the possible exception of the* ***ColorConvertOp*** *class)*, the **AffineTransformOp** class is the most significant of all the image-filtering classes in the Java 2D API.

Many sequential transforms can be performed

Although this demonstration program doesn't allow the user to perform a series of different transforms in sequence on the same image, be aware that once you start writing your own code, you can create many different image-processing effects by performing multiple transforms in sequence.

> *(Actually you could do that with this program by running the program several times in succession using the output file from one run as the input file to the next run, but that wouldn't be very convenient.)*

Interpolation is a major programming issue

I will explain more about interpolation shortly. For now, suffice it to say that because of the interpolation issue, a great deal of programming effort would be required for you to write your own class that replicates the behavior of the **AffineTransformOp** class. That is the main reason that I consider the capability provided by this class to be so significant. Therefore, if this class will serve your needs, this is clearly a case where you should *use* the existing class instead of *inventing* a new class.

> *(It would also be very difficult to write your own class to replicate the behavior of the* ***ColorConvertOp*** *class. However, I may be wrong on this, but I consider the* ***AffineTransformOp*** *class to be more useful, more of the time than the* ***ColorConvertOp*** *class.)*

Creating new color values through interpolation

Whenever you change the size, the location, or the orientation of an image, you usually need to recreate the color values for all of the pixels in the transformed image on the basis of the color values contained in the original image. This is not a trivial computational task. *(I discuss this requirement in additional detail in a later section that explains the requirements for scaling and then rendering images.)*

The AffineTransformOp class provides three interpolation choices

When you use the **AffineTransformOp** class to transform one image into another image, you have three choices regarding how the color values for the new pixels will be created:

- Use the *nearest neighbor*
- Perform *bilinear* interpolation
- Perform *bicubic* interpolation

The *Scaling* page of the program GUI shown in Figure 1 makes it possible for the user to select which of the three interpolation schemes will be used.

Higher quality equates to higher computational cost

Generally speaking, the quality of the resulting image will improve and the computer time required to generate the new image will increase as you go down the above list from top to bottom. In other words, *bicubic* interpolation usually requires more computational effort and provides better output image quality than the *nearest neighbor* scheme. *Bilinear* interpolation falls somewhere in between the other two.

Technical material on interpolation

Paul Bourke provides a general discussion of *bicubic* interpolation on his bicubic site and also mentions both *bilinear* and *nearest neighbor* interpolation as well.

You can also find mathematical descriptions of the three interpolation schemes by clicking the hyperlinks in the above list of interpolation choices.

Scaling

Figure 5 and Figure 6 show the results of a scaling transform. For this transform, the width of the image was increased by a factor of 1.5 and the height was increased by a factor of 2.0. The *nearest neighbor* scheme was used to create the new pixel values for Figure 5. *Bicubic* interpolation was used for Figure 6.

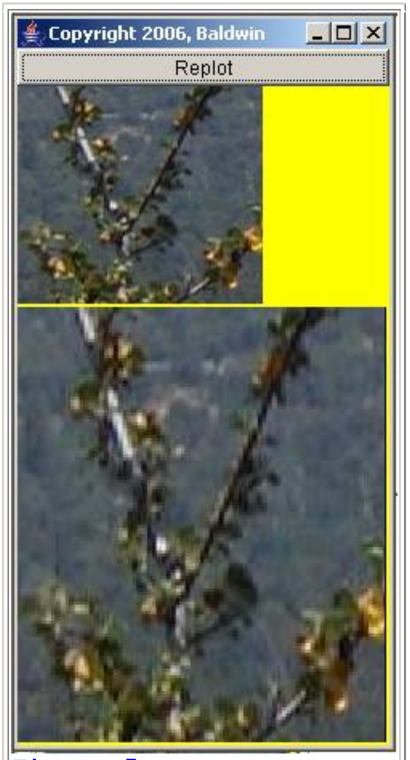


Figure 5

The image quality in Figure 6 is better

If you compare the bottom image in Figure 5 above with the bottom image in Figure 6 below, you should be able to see that the image quality in Figure 6 is superior to that in Figure 5.

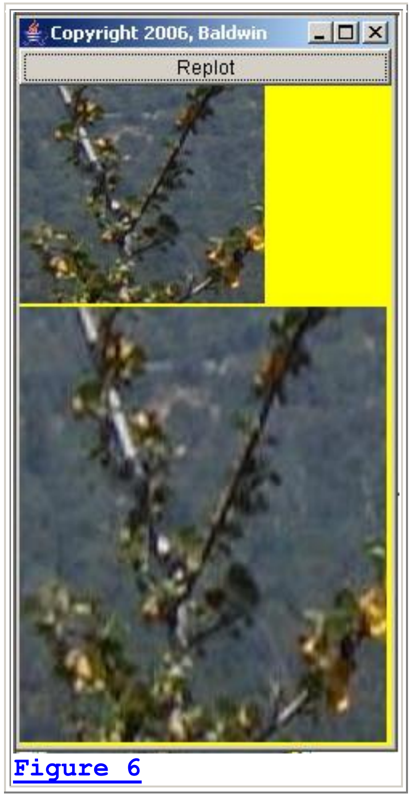


Figure 6

Neither image has outstanding quality

Although neither image shows outstanding quality, *(which is a common result of enlarging images of this type)*, the image produced using the *nearest neighbor* scheme in Figure 5 is more grainy than the image produced using *bicubic* interpolation in Figure 6.

Note the vertical stripes.
For example, note the vertical stripes in the light gray stem in the upper-left of the bottom image in Figure 5. Although there is some striping in this area of Figure 6, it isn't nearly as pronounced.

Translation

Figure 7 shows the result of translating the image by 15.25 pixels to the right and 20.75 pixels down. *(Bicubic interpolation was used for Figure 7.)*

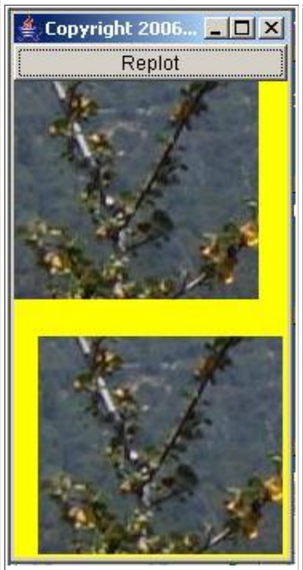


Figure 7

Ideally, the output image would be an exact copy of the input image in this case. However, there is probably some degradation due to the requirement to create the new pixel color values through interpolation of the original pixel color values.

Rotation

Figure 8 shows the result of first translating the image to the right and down a distance sufficient to give it room to rotate about its center without chopping off the corners, and then rotating it by 31.5 degrees in a clockwise direction about its center. *(Bicubic interpolation was used for Figure 8.)*

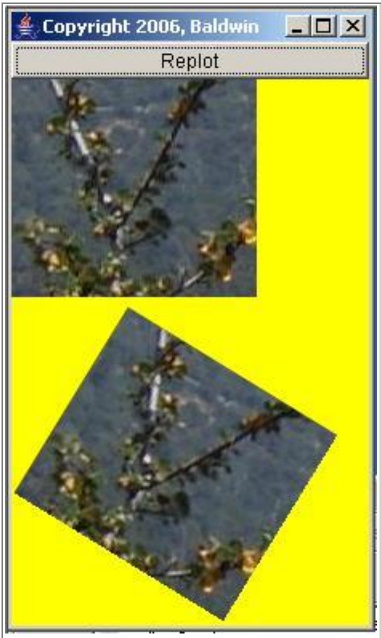


Figure 8

Mirror image

Figure 9 shows the result of using the **AffineTransformOp** class to translate the image to the right by a distance equal to its width, and then flipping it about its left edge. This creates a mirror image of the original image. *(Bicubic interpolation was used for Figure 9.)*

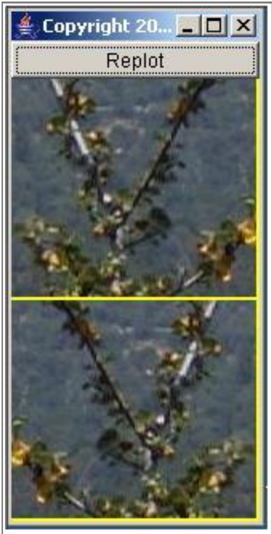


Figure 9

Now it's time to examine the code.

Discussion and Sample Code

The program named ImgMod40

As explained earlier, in this lesson, I will present and explain an image-processing program named **ImgMod40** that is compatible with the framework program named **ImgMod05**. This program provides a GUI that allows the user to perform the following transforms on an input image:

- Scaling
- Translation
- Rotation
- Mirror Image

With the exception of the Mirror Image transform, all of the transforms in the above list allow the user to input important parameters via the GUI.

Purpose

The purpose of this program is to show you how to write image-processing programs of this type, and also to illustrate a variety of different uses for the **AffineTransformOp** image-filtering class of the Java 2D API.

General comments

The program named **ImgMod038**, which I explained in the earlier lesson entitled Using the Java 2D LookupOp Filter Class to Process Images, contained a number of general comments that apply to this program also. You are encouraged to take a look at those comments.

Compatible with ImgMod50

This class is compatible with the use of the framework program named **ImgMod05**, which I explained in the earlier lesson entitled A Framework for Experimenting with Java 2D Image-Processing Filters. The framework program named **ImgMod05** displays the original and the processed images in the format shown in Figure 5. It also writes the processed image into an output file in JPEG format. The name of the output file is **junk.jpg** and it is written into the current directory.

The program GUI

Image-processing programs such as this one may provide a program GUI for data input making it possible for the user to modify the behavior of the image-processing method each time the **Replot** button *(shown in Figure 5)* is clicked.

This program creates a GUI consisting of a tabbed pane containing four pages. The tabs on the pages are labeled:

- Scaling

- Translation
- Rotation
- Mirror Image

Each page contains a set of controls that make it possible to process the image in a way that illustrates the processing concept indicated by the label on the tab.

The four pages of the tabbed pane are shown in Figure 1 through Figure 4.

Processing details for each page are provided in the comments in the code used to construct and process the individual pages.

Instructions for running the program

Enter the following at the command line to run this program:

```
java ImgMod05 ImgMod40 ImageFileName
```

If the program is unable to load the image file within ten seconds, it will abort with an error message.

The program was tested using J2SE 5.0 under WinXP.

Will discuss in fragments

I will discuss and explain this program in fragments. You can view a complete listing of the program in Listing 29 near the end of the lesson.

The class named ImgMod40

This program consists of a single class named **ImgMod40**, which begins in Listing 1.

```
class ImgMod40 extends Frame implements
ImgIntfc05{

  JTabbedPane tabbedPane = new JTabbedPane();
```

Listing 1

The class extends **Frame**, making an object of the class eligible to serve as its own program GUI.

The class also implements the interface named **ImgIntfc05** making it eligible to be run under control of the framework program named **ImgMod05**.

> *(Recall that image-processing programs that are compatible with the framework program named ImgMod05 must implement the interface named ImgIntfc05.)*

Listing 1 instantiates an object of the **JTabbedPane** class, which serves as the primary container for the GUI shown in Figure 1.

Instantiate control components for the GUI pages

Listing 2 instantiates a number of components required to construct the four pages of the GUI shown in Figure 1 through Figure 4. Those components that require local access only are instantiated closer to where they are used. Those that require broader access are instantiated in Listing 2 as instance variables.

```
  //Components used to construct the Scaling
page.
  Panel page00 = new Panel();
  TextField page00TextFieldHorizontal =
                                    new
TextField("0.5",6);
  TextField page00TextFieldVertical =
                                    new
TextField("0.5",6);
  //Components for radio buttons
  CheckboxGroup Page00Group = new
CheckboxGroup();
  Checkbox page00NearestNeighbor = new
Checkbox(
       "Nearest Neighbor
Interpolation",Page00Group,false);
  Checkbox page00Bilinear = new Checkbox(
               "Bilinear
Interpolation",Page00Group,false);
  Checkbox page00Bicubic = new Checkbox(
                 "Bicubic
Interpolation",Page00Group,true);


  //Components used to construct the
Translation page.
  Panel page01 = new Panel();
  TextField page01TextFieldHorizontal =
                                      new
TextField("5",6);
  TextField page01TextFieldVertical =
                                     new
TextField("10",6);


  //Components used to construct the Rotation
page.
  Panel page02 = new Panel();
  TextField page02TextField = new
TextField("45.0",6);


  //Components used to construct the Mirror
Image page.
```

```
  Panel page03 = new Panel();
```

Listing 2

The constructor

The constructor for the class is shown in its entirety in Listing 3. This is the primary constructor. It calls other methods to separate the construction of the GUI into easily understandable units. Each method that it calls constructs one page in the tabbed pane.

```
  ImgMod40(){//constructor

    constructPage00();
    tabbedPane.add(page00);//Add page to the
tabbedPane.

    constructPage01();
    tabbedPane.add(page01);//Add page to the
tabbedPane.

    constructPage02();
    tabbedPane.add(page02);//Add page to the
tabbedPane.

    constructPage03();
    tabbedPane.add(page03);//Add page to the
tabbedPane.

    add(tabbedPane);//Add tabbedPane to the
Frame.

    setTitle("Copyright 2006, R.G.Baldwin");
    setBounds(555,0,470,300);
    setVisible(true);

    //Define a WindowListener to terminate the
program.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent
e){
          System.exit(1);
        }//end windowClosing
      }//end windowAdapter
    );//end addWindowListener
  }//end constructor
```

Listing 3

The processImg method

The method named **processImg** is declared in the interface named **ImgIntfc05**, and must be defined by any program that is compatible with being executed under control of the framework program named **ImgMod05**.

This is the primary image-processing method of the program. It is called by the program named **ImgMod05** at startup, and each time thereafter that the user clicks the **Replot** button shown in Figure 5.

The **processImg** method is shown in its entirety in Listing 4.

```
  public BufferedImage processImg(BufferedImage
theImage){

    BufferedImage outputImage = null;

    //Process the page in the tabbed pane that
has been
    // selected by the user.
    switch(tabbedPane.getSelectedIndex()){
      case 0:outputImage =
processPage00(theImage);
              break;
      case 1:outputImage =
processPage01(theImage);
              break;
      case 2:outputImage =
processPage02(theImage);
              break;
      case 3:outputImage =
processPage03(theImage);
              break;
    }//end switch

    return outputImage;
  }//end processImg
```

Listing 4

Behavior of the processImg method

Each time the **processImg** method is invoked, it queries the **JTabbedPane** object in the GUI to determine which page has been selected by the user. Then, depending on which page has been selected, it invokes one of the following four methods to process the image in accordance with the concept embodied by the selected page:

- processPage00 - Scaling
- processPage01 - Translation
- processPage02 - Rotation
- processPage03 - Mirror Image

Scaling

Construct the Scaling page

The primary constructor shown in Listing 3 invokes the method named **constructPage00** to construct the *Scaling* page shown in Figure 1. The method named **constructPage00** is shown in its entirety in Listing 5.

```
  void constructPage00(){
    page00.setName("Scaling");//Label on the tab.
    page00.setLayout(new BorderLayout());

    //Create and add the instructional text to the
page.
    // This text appears in a disabled text area at
the
    // top of the page in the tabbed pane.
    String text ="IMAGE SCALING\n"
      + "This page illustrates the scaling of an
image "
      + "using three different types of
interpolation."
      + "\n\n"
      + "Enter a positive scale factor between
0.001 and "
      + "10.0 in each of the text fields, select an
"
      + "interpolation type, and click the Replot
button. "
      + "The best quality interpolation will
probably be "
      + "achieved with the use of Bicubic
Interpolation. "
      + "Nearest Neighbor Interpolation will
probably "
      + "produce the poorest quality.";

    //Note:  The number of columns specified for
the
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH
location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,7,1,

TextArea.SCROLLBARS_NONE);
    page00.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to the
page.
    Panel page00ControlPanel = new Panel();
    page00ControlPanel.setLayout(new
GridLayout(5,1));

    //Construct and populate the panels that
contain the
```

```
    // radio buttons, the labels, and the text
fields.
    // Add each such panel an a new cell in the
grid
    // layout going from the top of the grid to the
bottom
    // of the grid.

    //Begin with the radio buttons.  The purpose of
putting
    // the radio buttons on panels is to cause them
to be
    // left justified in their cells.
    Panel subControlPanel00 = new Panel();
    subControlPanel00.setLayout(new FlowLayout(

FlowLayout.LEFT));
    subControlPanel00.add(page00NearestNeighbor);
    page00ControlPanel.add(subControlPanel00);

    Panel subControlPanel01 = new Panel();
    subControlPanel01.setLayout(new FlowLayout(

FlowLayout.LEFT));
    subControlPanel01.add(page00Bilinear);
    page00ControlPanel.add(subControlPanel01);

    Panel subControlPanel02 = new Panel();
    subControlPanel02.setLayout(new FlowLayout(

FlowLayout.LEFT));
    subControlPanel02.add(page00Bicubic);
    page00ControlPanel.add(subControlPanel02);

    //Now create and populate panels that contain
labels
    // and associated TextField objects
    Panel subControlPanel03 = new Panel();
    subControlPanel03.setLayout(new FlowLayout(

FlowLayout.LEFT));
    subControlPanel03.add(new Label(
                               "Horizontal Scale
Factor"));

subControlPanel03.add(page00TextFieldHorizontal);
    page00ControlPanel.add(subControlPanel03);

    Panel subControlPanel04 = new Panel();
    subControlPanel04.setLayout(new FlowLayout(

FlowLayout.LEFT));
    subControlPanel04.add(new Label(
                                 "Vertical Scale
Factor"));
    subControlPanel04.add(page00TextFieldVertical);
    page00ControlPanel.add(subControlPanel04);


page00.add(page00ControlPanel,BorderLayout.CENTER);
  }//end constructPage00
```

Listing 5

Straightforward code

Although the code in Listing 5 is rather long and tedious, there is nothing complicated about it. If you use Figure 1 as a guide, you should be able to follow the code in Listing 5 with no difficulty.

The processPage00 method

When the user selects the *Scaling* tab in Figure 1 and clicks the **Replot** button shown in Figure 5, the **switch** statement in Listing 4 invokes the method named **processPage00** to process the image.

General processing methodology

In general, the methodology used to process an image using the **AffineTransformOp** image-filtering class of the Java 2D API consists of the following three steps:

1. Get an object of the **AffineTransform** class that reflects the type of transformation that is required. *(See the earlier lesson entitled Java 2D Graphics, Simple Affine Transforms for more information on the **AffineTransform** class.)*
2. Use the **AffineTransform** object to create an image-filtering object of the **AffineTransformOp** class with a specified interpolation scheme.
3. Invoke the **filter** method on the image-filtering object to apply the filter to the image.

As you will see, some of the above steps require additional code in preparation for accomplishing the step.

Processing the image

The **processPage00** method processes the image according to the controls located on the *Scaling* page shown in Figure 1. This method uses the **AffineTransformOp** filter class to process the image.

This method illustrates image scaling using independent horizontal and vertical scale factors, giving the user a choice of three different interpolation schemes. *(See the earlier lesson entitled Java 2D Graphics, Simple Affine Transforms for more information on the use of a scaling affine transform.)*

The **processPage00** method begins in Listing 6.

```
  BufferedImage processPage00(BufferedImage
theImage){

    //Set a non-zero default value for the
horizontal scale
    // factor.
    double horizontalScale = 0.001;
    try{//Get horizontalScale from the text
field.
      horizontalScale = Double.parseDouble(

page00TextFieldHorizontal.getText());
    }catch(java.lang.NumberFormatException e){
      page00TextFieldHorizontal.setText("Bad
Input");
      horizontalScale = 0.001; //Override bad
user input.
    }//end catch

    //Guarantee reasonable values for
horizontal scale
    if((horizontalScale < 0.001) ||

(horizontalScale > 10.0)){
      page00TextFieldHorizontal.setText("Bad
Input");
      horizontalScale = 0.001;//Override bad
user input.
    }//end if

    //Set a non-zero default value for the
vertical scale
    // factor.
    double verticalScale = 0.001;
    try{//Get verticalScale from the text
field.
      verticalScale = Double.parseDouble(

page00TextFieldVertical.getText());
    }catch(java.lang.NumberFormatException e){
      page00TextFieldHorizontal.setText("Bad
Input");
      verticalScale = 0.001; //Override bad
user input.
    }//end catch

    //Guarantee reasonable values for
verticalScale
    if((verticalScale < 0.001) ||
(verticalScale > 10.0)){
      page00TextFieldHorizontal.setText("Bad
Input");
      verticalScale = 0.001;//Override bad user
input.
    }//end if
```

Listing 6

Getting the scale factors

The first task that the method must accomplish is to get the horizontal and vertical scale factors that were entered by the user into the text fields in Figure 1. This is accomplished by the code in Listing 6. The code in Listing 6 is straightforward, and should not require further explanation beyond the comments embedded in the code.

Identify the desired interpolation scheme

The next task that the method must perform is to determine which interpolation scheme has been selected by the user and to become prepared to use that interpolation scheme in the creation of the image-filtering object. This is accomplished in Listing 7.

```
    int interpolationScheme;

    if(page00Bicubic.getState() == true){
      interpolationScheme =

AffineTransformOp.TYPE_BICUBIC;
    }else if(page00Bilinear.getState() ==
true){
      interpolationScheme =

AffineTransformOp.TYPE_BILINEAR;
    }else{//page00NearestNeighbor must be
selected
      interpolationScheme =

AffineTransformOp.TYPE_NEAREST_NEIGHBOR;
    }//end else
```

Listing 7

Listing 7 gets the selected interpolation scheme from the radio buttons and reflects that selection in an **int** variable named **interpolationScheme**. This variable will be used later in the creation of the image-filtering object.

Create the required AffineTransform object

As indicated in the above list of three required steps, an **AffineTransform** object is required later to create the image-filtering object. The creation of that object is accomplished by Listing 8.

```
    AffineTransform transformObj =

AffineTransform.getScaleInstance(
horizontalScale,verticalScale);
```

Listing 8

An examination of the documentation for the **AffineTransform** class shows that there are several different ways to create such an object. The statement in Listing 8 is probably the simplest of those ways.

> *(The **getScaleInstance** method of the **AffineTransform** class is a convenience method that is designed to make it easy to create scaling transforms.)*

Create the image-filtering object

Listing 9 accomplishes the second step in the above list of three required steps.

```
    AffineTransformOp filterObj = new
AffineTransformOp(

transformObj,interpolationScheme);
```

Listing 9

Listing 9 instantiates a new object of the **AffineTransformOp** class passing a reference to the **AffineTransform** object, along with the variable that identifies the interpolation scheme to the constructor for the **AffineTransformOp** class. The resulting object will be used to filter the image.

Now for something different

For all of the image-filtering classes in the Java 2D API *(except for the **AffineTransformOp** class)*, all that is necessary to filter the image and to return a reference to a **BufferedImage** object *(that encapsulates the filtered image)* is to execute a statement similar to the following:

```
return filterObj.filter(theImage,null);
```

For example, this is very similar to the code that I used to filter the images in the earlier lesson entitled Using the Java 2D LookupOp Filter Class to Process Images.

However, for reasons that I am unable to explain, when I use that approach for the **AffineTransformOp** class, the **ColorModel** of the **BufferedImage** object that is returned to the framework program named **ImgMod05** is not compatible with the method used by that program to write the output JPEG file. This results in an output file in which the image data appears to be scrambled. Therefore, it was necessary for me to find and use an alternative approach that provides better control over the color model.

An alternative approach

Figure 10 contains partial documentation for the **filter** method of the **AffineTransformOp** class.

```
public final BufferedImage filter(BufferedImage src,
                                  BufferedImage dst)
```

Transforms the source `BufferedImage` and stores the results in the destination `BufferedImage`. If the color models for the two images do not match, a color conversion into the destination color model is performed. If the destination image is null, a `BufferedImage` is created with the source `ColorModel`.

Parameters:
`src` - The `BufferedImage` to transform.
`dst` - The `BufferedImage` in which to store the results of the transformation.
Returns:
The filtered `BufferedImage`.

Figure 10

Two ways to access the filtered image

According to the information in Figure 10, there are two ways to gain access to the filtered image:

1. Capture the return value from the **filter** method, which is a reference to a **BufferedImage** object that encapsulates the filtered image. *(With this approach, null can be passed as the second parameter to the **filter** method.)*
2. Provide a reference to a **BufferedImage** object as the second parameter to the **filter** method. *(The method will deposit the filtered image in this **BufferedImage** object.)*

The simple return statement that I have used with the other image-filtering classes of the Java 2D API is based on the first approach in the above list.

The somewhat more complicated alternative approach that I was forced to use in this program is based on the second approach in the above list.

The createCompatibleDestImage method

In summary, this alternative approach invokes the **createCompatibleDestImage** method of the **AffineTransformOp** class to create *"a zeroed destination image with the correct size and number of bands."* A reference to this object is passed as the second parameter to the **filter** method.

The **createCompatibleDestImage** method requires two parameters:

- src - The **BufferedImage** to be transformed.
- destCM - The **ColorModel** of the destination image.

Explicit control over the color model

I'm not certain that it was the **ColorModel** that caused the problem in the first place. However, as you will see in Listing 10, I forced the **ColorModel** of the destination object to match the **ColorModel** of the image being filtered *(by invoking the **getColorModel** method on the image being filtered to create the **ColorModel** for the second parameter to the **createCompatibleDestImage** method)* and that resolved the problem.

Create the destination object

Listing 10 creates a destination **BufferedImage** object that will be passed to the **filter** method to receive the filtered image.

```
    BufferedImage dest =

filterObj.createCompatibleDestImage(

theImage,theImage.getColorModel());
```

Listing 10

Filter the image and return the filtered image

Listing 11 invokes the **filter** method to apply the filter to the image and save the filtered image in the destination object. Then the code in Listing 11 returns a reference to the destination object. This reference is eventually returned to the framework program named **ImgMod05**, where the filtered image is displayed as shown in Figure 5 and also written into an output JPEG file named **junk.jpg**.

```
    filterObj.filter(theImage, dest);

    //Return the destination object's
reference.
    return dest;

  }//end processPage00
```

Listing 11

Listing 11 also signals the end of the method named **processPage00**, and the end of the explanation of the scaling transform under the control of the *Scaling* page shown in Figure 1.

Translation

Construct the Translation page

The primary constructor shown in Listing 3 invokes the method named **constructPage01** to construct the *Translation* page shown in Figure 2. The method named **constructPage01** is shown in its entirety in Listing 12.

```
  void constructPage01(){
    page01.setName("Translation");//Label on the
tab.
    page01.setLayout(new BorderLayout());

    //Create and add the instructional text to the
page.
    // This text appears in a disabled text area at
the
    // top of the page in the tabbed pane.
    String text ="IMAGE TRANSLATION\n"
      + "This page illustrates the translation of
an "
      + "image to a new location relative to the "
      + "upper-left corner of the container using
Bicubic "
      + "Interpolation.\n\n"
      + "Enter the horizontal and vertical
translation "
      + "distances in pixels into the text fields
and "
      + "click the Replot button.\n\n"
      + "Note that the translation distances must
be "
      + "between -1000 and +1000 pixels.  Also note
that "
      + "negative translations may shift the image
"
      + "completely out of the Frame on the top or
the "
      + "left side of the image.";

    //Note:  The number of columns specified for
the
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH
location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,9,1,

TextArea.SCROLLBARS_NONE);
    page01.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to the
page.
    Panel page01ControlPanel = new Panel();
    page01ControlPanel.setLayout(new
GridLayout(3,1));

    //Place each label and its corresponding text
field
    // on a panel.  Place the panels in the cells
in the
    // grid layout from top to bottom.  Note that
there
```

```
    // is an empty cell at the bottom for cosmetic
    // purposes.
    Panel subControlPanel00 = new Panel();
    subControlPanel00.setLayout(
                          new
FlowLayout(FlowLayout.LEFT));
    subControlPanel00.add(new Label(
             "Horizontal Translation Distance in
Pixels"));

subControlPanel00.add(page01TextFieldHorizontal);
    page01ControlPanel.add(subControlPanel00);

    Panel subControlPanel01 = new Panel();
    subControlPanel01.setLayout(
                          new
FlowLayout(FlowLayout.LEFT));
    subControlPanel01.add(new Label(
               "Vertical Translation Distance in
Pixels"));
    subControlPanel01.add(page01TextFieldVertical);
    page01ControlPanel.add(subControlPanel01);


page01.add(page01ControlPanel,BorderLayout.CENTER);
  }//end constructPage01
```

Listing 12

If you use Figure 2 as a guide, you should have no problems following the code in Listing 12.

Processing the image

The **processPage01** method is called from within the **switch** statement in Listing 4 to process the image according to the controls located on the *Translation* page shown in Figure 2. As before, this method uses the **AffineTransformOp** filter class to process the image.

This method illustrates image translation using independent horizontal and vertical distance values. The method uses *bicubic* Interpolation.

Get translation distances and set the interpolation scheme

The **processPage01** method begins in Listing 13.

```
  BufferedImage processPage01(BufferedImage
theImage){

    double horizontalDistance = 0.0;
    try{//Get horizontalDistance from the text
field.
      horizontalDistance = Double.parseDouble(
page01TextFieldHorizontal.getText());
    }catch(java.lang.NumberFormatException e){
      page01TextFieldHorizontal.setText("Bad
Input");
      horizontalDistance = 0.0; //Override bad
user input.
    }//end catch

    //Guarantee reasonable values for
horizontalDistance
    if((horizontalDistance < -1000.0) ||
                            (horizontalDistance
> 1000.0)){
      page01TextFieldHorizontal.setText("Bad
Input");
      horizontalDistance = 0.0;//Override bad
user input.
    }//end if

    double verticalDistance = 0.0;
    try{//Get verticalDistance from the text
field.
      verticalDistance = Double.parseDouble(

page01TextFieldVertical.getText());
    }catch(java.lang.NumberFormatException e){
      page01TextFieldHorizontal.setText("Bad
Input");
      verticalDistance = 0.0; //Override bad
user input.
    }//end catch

    //Guarantee reasonable values for
verticalDistance
    if((verticalDistance < -1000.0) ||
                              (verticalDistance
> 1000.0)){
      page01TextFieldHorizontal.setText("Bad
Input");
      verticalDistance = 0.0;//Override bad
user input.
    }//end if

    //Set the interpolation scheme to the best
available.
    // Note that this page doesn't allow the
user to
    // select the interpolation scheme.
    int interpolationScheme =

AffineTransformOp.TYPE_BICUBIC;
```

Listing 13

The first two tasks that must be accomplished by the method are to get the translation distances that were entered by the user into the text fields in Figure 2, and to set the interpolation scheme. This is accomplished by the code in Listing 13, which is straightforward, and shouldn't require further explanation.

Create the required AffineTransform object

As explained earlier, the methodology used to process an image using the **AffineTransformOp** image-filtering class of the Java 2D API consists of three steps.

The first step is to get an object of the **AffineTransform** class that reflects the type of transformation that is required. *(See the earlier lesson entitled Java 2D Graphics, Simple Affine Transforms for more information on the **AffineTransform** class.)* This step is accomplished by the code in Listing 14.

```
    AffineTransform transformObj =

AffineTransform.getTranslateInstance(

horizontalDistance,verticalDistance);
```

Listing 14

The getTranslateInstance method

Listing 14 creates an **AffineTransform** object for translation that matches the user input from the control panel shown in Figure 2. The code in Listing 14 invokes the static **getTranslateInstance** convenience method of the **AffineTransform** class to create the object.

Note that even though the actual translation is ultimately performed in terms of integer pixels, the **getTranslateInstance** method of the **AffineTransform** class requires the horizontal and vertical translation distances to be provided as type **double**.

What does it mean to translate an image?

By way of explanation, my earlier article entitled Java 2D Graphics, Simple Affine Transforms has this to say about translation:

> *"The purpose of translation is to move the origin of the coordinate system in device space.*
>
> *For example, the default position of the origin is the upper left-hand corner of the component on which the graphic is being displayed. Assume that the component is a **Frame** object that is four inches on each side. You might like for the origin to be in the center of the **Frame** instead of at the top left-hand corner. You could accomplish this by translating the origin by two inches in both the horizontal and vertical directions.*

Or, you might like for the origin to be just barely inside the borders of the ***Frame*** *object instead of outside the borders as is the default. This can be accomplished by getting the widths of the top border and left border by invoking* ***getInsets*** *on the* ***Frame****, and then using those values to translate the origin to a location that is just barely inside the borders."*

Moving the origin

Thus, when you translate an image, you are actually moving the origin. The pixels, in turn, move along with the origin. A positive horizontal translation distance will cause the image to appear to move to the right in its container and a negative horizontal distance will cause the image to appear to move to the left. Similarly, a positive vertical distance will cause the image to appear to move down and a negative vertical distance will cause the image to appear to move up.

Understanding this concept of moving the origin will become somewhat more important later when I explain the code that produces the mirror image.

Create a filtering object

The second required step that I described earlier is to use the **AffineTransform** object to create an image-filtering object of the class **AffineTransformOp** with a specified interpolation scheme. This is accomplished in Listing 15, which is essentially the same as the code in Listing 9.

```
    AffineTransformOp filterObj = new
AffineTransformOp(

transformObj,interpolationScheme);
```

Listing 15

Filter the image and return the filtered image

The third required step that I described earlier is to invoke the **filter** method on the image-filtering object to apply the filter to the image. This is accomplished in Listing 16.

```
    //Create a destination object.
    BufferedImage dest =

filterObj.createCompatibleDestImage(

theImage,theImage.getColorModel());

    //Filter the image and save the filtered
image in the
    // destination object.
    filterObj.filter(theImage, dest);
    //Return a reference to the destination
object.
    return dest;
  }//end processPage01
```

Listing 16

The code in Listing 16 is essentially the same as the code that I explained earlier in Listing 10 and Listing 11

Listing 16 also signals the end of the **processPage01** method, and the end of the explanation of the *Translation* page sown in Figure 2.

Rotation

Construct the Rotation page

The primary constructor shown in Listing 3 invokes the method named **constructPage02** to construct the *Rotation* page shown in Figure 3. The method named **constructPage02** is shown in its entirety in Listing 17.

```
  void constructPage02(){
    page02.setName("Rotation");//Label on the tab.
    page02.setLayout(new BorderLayout());

    //Create and add the instructional text to the
page.
    // This text appears in a disabled text area at
the
    // top of the page in the tabbed pane.
    String text ="IMAGE Rotation\n"
      + "This page illustrates translation of an
image "
      + "followed by Rotation of the same image
using "
      + "Bicubic Interpolation.\n\n"
      + "The image is rotated around its center
after "
      + "being translated to the right and down by
a "
      + "distance that is sufficient to give it
room to "
      + "rotate.\n\n"
      + "Enter the desired rotation angle in
degrees and "
      + "click the Replot button.  Positive
rotation "
      + "angles represent clockwise rotation and
negative "
      + "rotation angles represent counter-
clockwise "
```

```
      + "rotation.";

    //Note:  The number of columns specified for
the
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH
location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,9,1,

TextArea.SCROLLBARS_NONE);
    page02.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to the
page.
    // Use a control panel with a GridLayout for
    // cosmetic purposes.  Note that there are two
empty
    // cells at the bottom of the grid.
    Panel page02ControlPanel = new Panel();
    page02ControlPanel.setLayout(new
GridLayout(3,1));

    //Place the label and the text field on a panel
and
    // place that panel in the top cell in the
grid.
    Panel subControlPanel00 = new Panel();
    subControlPanel00.setLayout(new FlowLayout(

FlowLayout.LEFT));
    //Note, a positive value in degrees represents
    // clockwise rotation.
    subControlPanel00.add(new Label(
                               "Rotation in
Degrees"));
    subControlPanel00.add(page02TextField);
    page02ControlPanel.add(subControlPanel00);


page02.add(page02ControlPanel,BorderLayout.CENTER);
  }//end constructPage02
```

Listing 17

The method named **ConstructPage02** constructs the *Rotation* page shown in Figure 3. If you use Figure 3 as a guide, you should have no problems following the code in Listing 17.

Processing the image

The **processPage02** method is called from within the **switch** statement in Listing 4 to process the image according to the controls located on the *Rotation* page shown in Figure 3. As before, this method uses the **AffineTransformOp** filter class to process the image.

This method illustrates image rotation where the angle of rotation is specified by the user in degrees. A positive angle in degrees results in clockwise rotation. A negative angle in degrees results in counter-clockwise rotation.

The **processPage02** method uses *bicubic* Interpolation.

Get the rotation angle in degrees

The **processPage02** method begins in Listing 18. The code in Listing 18 gets the user input in degrees rotation from the text field in Figure 3. Then it converts the angle from degrees to radians.

```
  BufferedImage processPage02(BufferedImage 
theImage){

    //Get the rotation angle in degrees.  A 
positive angle
    // in degrees corresponds to clockwise 
rotation.
    double rotationAngleInDegrees = 0.0;
    try{//Get rotationAngleInDegrees from the 
text field.
      rotationAngleInDegrees = 
Double.parseDouble(

page02TextField.getText());
    }catch(java.lang.NumberFormatException e){
      page02TextField.setText("Bad Input");
      rotationAngleInDegrees = 0.0;//Override 
bad input.
    }//end catch

    //Compute the rotation angle in radians.
    double rotationAngleInRadians = 

rotationAngleInDegrees*Math.PI/180.0;
```

Listing 18

If you use Figure 3 as a guide, you should have no problems following the code in Listing 18.

Set the interpolation scheme

Listing 19 sets the interpolation scheme to *bicubic*.

```
    int interpolationScheme = 

AffineTransformOp.TYPE_BICUBIC;
```

Listing 19

Translate the image down and to the right

If the image in the top of Figure 8 were to be simply rotated about its center, the resulting image would be too wide and too tall to fit in the same amount of space. Therefore, before rotating the image, this program attempts to translate the image down and to the right a sufficient distance to allow it to be rotated about its center without chopping of the corners on the left side and the top. This is accomplished by estimating the length of the image diagonal as the hypotenuse of a right triangle and then using that distance to perform the translation.

This translation operation is shown in Listing 20.

```
    //Translate the image down and to the right
far enough
    // that the corners won't be chopped off by
the top and
    // left edges of the container when the
image is
    // rotated by 45 degrees.

    //Get the length of half the diagonal
dimension of the
    // image using the formula for the
hypotenuse of a
    // right triangle.
    int halfDiagonal = (int)(Math.sqrt(
         theImage.getWidth()*theImage.getWidth()
+

theImage.getHeight()*theImage.getHeight())/2.0);

    //Set the horizontal and vertical
translation
    // distances.
    int horizontalDistance =
                      halfDiagonal -
theImage.getWidth()/2;
    int verticalDistance =
                     halfDiagonal -
theImage.getHeight()/2;

    //Create an Affine Transform object that can
be used
    // to translate the image by the distances
computed
    // above.
    AffineTransform transformObj =

AffineTransform.getTranslateInstance(

horizontalDistance,verticalDistance);

    //Get a translation filter object based on
the
    // AffineTransform object.
    AffineTransformOp filterObj = new
AffineTransformOp(

transformObj,AffineTransformOp.TYPE_BICUBIC);

    //Perform the translation and save the
modified image
    // as type BufferedImage.  This image will
be the input
    // to the rotation transform.
    BufferedImage translatedImage =

filterObj.filter(theImage, null);
```

Listing 20

Since you already understand image translation, further explanation of the code in Listing 20 shouldn't be necessary.

Get the AffineTransform object

Listing 21 invokes the static **getRotateInstance** method of the **AffineTransform** class to get an **AffineTransform** object suitable for causing the image to be rotated around a point that represents the center of the image before it was translated to the right and down. *(This satisfies the first step in the earlier list of three required steps.)*

```
    transformObj =
AffineTransform.getRotateInstance(
                rotationAngleInRadians,
                horizontalDistance +
theImage.getWidth()/2,
                verticalDistance +
theImage.getHeight()/2);
```

Listing 21

The angle in radians

The first parameter to the **getRotateInstance** method in Listing 21 specifies the rotation angle in radians *(note the conversion from degrees to radians in Listing 18 above).*

The anchor point

The second and third parameters specify the coordinates of an anchor point around which the image is to be rotated. According to Sun, this overloaded version of the **getRotateInstance** method:

> *"Returns a transform that rotates coordinates around an anchor point. This operation is equivalent to translating the coordinates so that the anchor point is at the origin (S1), then rotating them about the new origin (S2), and finally*

translating so that the intermediate origin is restored to the coordinates of the original anchor point (S3)."

Coordinate considerations

When the original image was translated by the filtering operation in Listing 20, and the translated image was returned in an object of type **BufferedImage**, the coordinate origin was effectively reset back to the top left corner of the new larger image. Because the actual picture that constitutes the original image was shifted to the right and down, the coordinates of the center of that picture were then greater than was the case before the translation took place. The expressions passed as the second and third parameters in Listing 21 specify the new coordinates of the center of the picture.

Get a rotation image-filtering object

Listing 22 gets a rotation filter object based on the transform object and the specified interpolation scheme. *(This satisfies the second step in the earlier list of three required steps.)*

```
    filterObj = new AffineTransformOp(

transformObj,interpolationScheme);
```

Listing 22

Note that Listing 22 re-uses the variable named **filterObj** of type **AffineTransformOp** that was declared in Listing 20 and originally used to translate the image down and to the right.

Apply the filter

Listing 23 satisfies the third step in the earlier list of three required steps by applying the rotation filter to the image. Listing 23 also returns a reference to the processed image such that it will eventually be displayed by the program named **ImgMod05**, as shown in Figure 8.

```
    BufferedImage dest =

filterObj.createCompatibleDestImage(

translatedImage,theImage.getColorModel());

    //Filter the image and save the filtered
image in the
    // destination object.
    filterObj.filter(translatedImage, dest);

    //Return a reference to the destination
object.
    return dest;

  }//end processPage02
```

Listing 23

The code in Listing 23 is essentially the same as the code that I explained in Listing 10 and Listing 11.

Listing 23 also signals the end of the method named **processPage02**, and the end of the explanation of the processing of the *Rotation* page shown in Figure 3.

Mirror Image

The *Mirror Image* operation consists of the concatenation of a translation operation followed by a scaling operation.

What does it really mean to scale an image?

To understand the process of creating a mirror image, we first need to understand what it really means to scale an image.

The representation of an image

An image consists of a set of red, green, and blue sample values specified at a corresponding set of coordinates. You can think of these sample values as representing elevation samples taken from a set of three 3D surfaces. One surface represents red, one represents green, and the other represents blue.

Stretching or shrinking the surfaces

When we scale the image, we effectively multiply each of the coordinate values by a scale factor without modifying the sample values. This has the effect of causing the original sample values to occur at a different set of coordinate values, either stretching or shrinking the surfaces that they represent.

If the scale factor is greater than 1.0, this stretches the surface causing the distance between the sample values to increase. If the scale factor is less than 1.0, this causes the surface to shrink and the distance between the sample values decreases.

Reconstructing the three surfaces

For rendering purposes, once we have scaled the image, we need to estimate the elevation values of each surface at each pixel location in a uniform grid of pixels. Therefore, the new surfaces described by the elevation sample values at the new locations must be reflected onto the grid that represents the pixels.

Interpolation

This is where interpolation comes into play. If the new locations of the sample values don't fall exactly on the locations of pixels in the grid, those sample values must be used to estimate the elevation of the surfaces at the pixel locations. Even if the new locations of the sample values do fall on the locations of pixels in the grid, if the surface was stretched, then interpolation must be used to estimate the elevation of the surface at pixel locations in between the new locations of the sample values.

What if the scale factors are negative?

An interesting thing happens if, for example, the horizontal scale factor is negative. This causes the locations of samples that were originally to the right of the origin to be moved to the negative plane on the left of the origin. This is essentially the process that is used to create the mirror image shown in Figure 9 under control of the page shown in Figure 4. In this case, the origin is translated to the right by an amount equal to the width of the original image. Then the horizontal coordinate value for each pixel is multiplied by -1 causing the locations of all the samples to be moved to the left of the new origin.

Construct the Mirror Image page

The primary constructor shown in Listing 3 invokes the method named **constructPage03** to construct the *Mirror Image* page shown in Figure 4. The method named **constructPage03** is shown in its entirety in Listing 24.

```
  void constructPage03(){
    page03.setName("Mirror Image");//Label on
the tab.
    page03.setLayout(new BorderLayout());

    //Create and add the instructional text to
the page.
    //This text appears in a disabled text area
at the
    // top of the page in the tabbed pane.
    String text ="Mirror Image\n\n"
      + "This page translates the image to the
right by "
      + "an amount equal to its width, and then
flips it "
      + "around its left edge to produce a
mirror image "
      + "of the original image.\n\n"
      + "Click the Replot button to create the
mirror "
      + "image.";

    //Note:  The number of columns specified
for the
    // following TextArea is immaterial because
the
    // TextArea object is placed in the NORTH
location of
```

```
    // a BorderLayout.
    TextArea textArea = new TextArea(text,6,1,

TextArea.SCROLLBARS_NONE);
    page03.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

  }//end constructPage03
```

Listing 24

The processPage03 method

When the user selects the *Mirror Image* tab in Figure 4 and clicks the **Replot** button in Figure 9, the **switch** statement in Listing 4 invokes the method named **processPage03** to process the image. The **processPage03** method begins in Listing 25.

Note that unlike *Scaling*, *Translation*, and *Rotation*, there is no affine transform for *Mirror Image*. Rather, as mentioned earlier, this method illustrates the use of a horizontal translation followed by scaling with a negative scale factor to produce a mirror image of the original image.

```
  BufferedImage processPage03(BufferedImage
theImage){

    AffineTransform transformObj =

AffineTransform.getTranslateInstance(

theImage.getWidth(),0);
```

Listing 25

Listing 25 gets an **AffineTransform** object that can be used to shift the image to the right by an amount equal to its width.

A transformation matrix

Although I don't plan to get into the details here, as I explained in the earlier lesson entitled Java 2D Graphics, Simple Affine Transforms, an **AffineTransform** is a linear transform, so the transformation can be expressed in the matrix notation of linear algebra. An arbitrary **AffineTransform** can be mathematically expressed by six numbers in a matrix.

Modification of the transform matrix

Those six numbers can be manipulated and modified at will in numerous ways prior to the actual application of the transform process. One of the most interesting aspects of such manipulation is that the matrices for different kinds of transforms can be combined through concatenation into a single matrix. Then, when the transformation is actually performed using the resulting transformation matrix, the result will be as if each of the individual transformations had been performed separately and in sequence.

The scale method of the AffineTransform class

Although it is possible to perform such matrix modifications by working directly with the six values in the matrix, for some cases, there is an easier way. For example, one of the methods of the **AffineTransform** class is a convenience method named **scale**.

The **scale** method concatenates an existing transform matrix with a scaling transform matrix, producing a matrix that performs both transforms at the same time.

Listing 26 invokes the **scale** method to concatenate the translation matrix created in Listing 25 with a scaling matrix. In this case, the horizontal scale factor is -1.0 and the vertical scale factor is 1.0. When this transform matrix is applied to an image, it will be translated and scaled in a single operation.

```
    transformObj.scale(-1.0, 1.0);
```

Listing 26

Display the transform matrix

For illustration purposes only, Listing 27 displays the contents of the resulting transform matrix.

```
    //Display the six values in the
transformation matrix.
    double[] theMatrix = new double[6];
    transformObj.getMatrix(theMatrix);

    //Display first row of values by displaying
every
    // other element in the array starting with
element
    // zero.
    for(int cnt = 0; cnt < 6; cnt+=2){
      System.out.print(theMatrix[cnt] + "\t");
    }//end for loop

    //Display second row of values displaying
every
    // other element in the array starting with
element
    // number one.
    System.out.println();//new line
    for(int cnt = 1; cnt < 6; cnt+=2){
      System.out.print(theMatrix[cnt] + "\t");
    }//end for loop
    System.out.println();//end of line
    System.out.println();//blank line
```

```
Listing 27
```

The code in listing 27 is not required for the *Mirror Image* transform to be performed. Rather, it is provided to allow you to examine the values in the transformation matrix. If you don't care about the mathematics involved, you can completely ignore the code in Listing 27.

Perform the three required steps

Listing 28 performs the three required steps that were listed earlier, using code that has been previously explained.

```
    //Get a translation filter object based on 
the
    // AffineTransform object.
    AffineTransformOp filterObj = new 
AffineTransformOp(

transformObj,AffineTransformOp.TYPE_BICUBIC);

    BufferedImage dest = 

filterObj.createCompatibleDestImage(

theImage,theImage.getColorModel());

    //Filter the image and save the filtered 
image in the
    // destination object.
    filterObj.filter(theImage, dest);

    //Return a reference to the destination 
object.
    return dest;

  }//end processPage03

Listing 28
```

The end of the program

Listing 28 also signals the end of the method named **processPage03**, and the end of the explanation of the *Mirror Image* transform that produces results similar to those shown in Figure 9 under control of the page shown in Figure 4.

Finally, Listing 28 also signals the end of the explanation of the program named **ImgMod40**.

Run the Program

I encourage you to copy the code from Listing 29 into your text editor, compile it, and execute it. Experiment with it, making changes, and observing the results of your changes.

Remember, you will also need to compile the code for the framework program named **ImgMod05** and the interface named **ImgIntfc05**. You will find that source code in the earlier lesson entitled A Framework for Experimenting with Java 2D Image-Processing Filters.

You will also need one or more JPEG image files to experiment with. You should have no difficulty finding such files at a variety of locations on the web. I recommend that you stick with relatively small images so that both the original image and the processed image will fit in the vertical space on your screen in the format shown in Figure 8.

Summary

In this lesson, I provided and explained an image-processing program named **ImgMod40** that is compatible with the framework program named **ImgMod05**.

The purpose of this program is to show you how to write such programs, and also to illustrate a variety of different uses for the **AffineTransformOp** class of the Java 2D API.

Four specific uses of the **AffineTransformOp** class were illustrated, and you should be able to devise many more.

What's Next?

Future lessons in this series will teach you how to use the following image-filtering classes from the Java 2D API:

- **BandCombineOp**
- **ConvolveOp**
- **RescaleOp**
- **ColorConvertOp**

References

- 400 Processing Image Pixels using Java, Getting Started
- 402 Processing Image Pixels using Java, Creating a Spotlight
- 404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
- 406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
- 408 Processing Image Pixels, Performing Convolution on Images
- 410 Processing Image Pixels, Understanding Image Convolution in Java
- 412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
- 414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
- 416 Processing Image Pixels, An Improved Image-Processing Framework in Java
- 450 A Framework for Experimenting with Java 2D Image-Processing Filters

- 452 Using the Java 2D LookupOp Filter Class to Process Images

Complete Program Listing

A complete listing of the program discussed in this lesson is shown in Listing 29 below.

```
/*File ImgMod40.java
Copyright 2006, R.G.Baldwin

The purpose of this class is to illustrate a variety of
different uses for the AffineTransformOp filter class of
the Java 2D API.

See general comments in the class named ImgMod038.

This class is compatible with the use of the driver program
named ImgMod05.

The driver program named ImgMod05 displays the original and
the modified images.  It also writes the modified image
into an output file in JPEG format.  The name of the output
file is junk.jpg and it is written into the current
directory.

Image-processing programs such as this one may provide a
GUI for data input making it possible for the user to
modify the behavior of the image-processing method each
time the Replot button is clicked.  Such a GUI is provided
for this program.

Enter the following at the command line to run this
program:

java ImgMod05 ImgMod40 ImageFileName

If the program is unable to load the image file within ten
seconds, it will abort with an error message.

This program creates a GUI consisting of a tabbed pane
containing four pages.  The tabs on the pages are labeled:

Scaling
Translation
Rotation
Mirror Image

Each page contains a set of controls that makes it possible
to process the image in a way that illustrates the
processing concept indicated by the label on the tab.
Processing details for each page are provided in the
comments in the code used to construct and process the
individual pages.

Tested using J2SE 5.0 under WinXP.
```

```
**********************************************************/

import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.geom.AffineTransform;

class ImgMod40 extends Frame implements ImgIntfc05{
  //Primary container used to construct the GUI.
  JTabbedPane tabbedPane = new JTabbedPane();

  //Components used to construct the Scaling page.
  // Components that require local access only are defined
  // locally.  Others are defined here as instance
  // variables.
  Panel page00 = new Panel();
  TextField page00TextFieldHorizontal =
                                    new TextField("0.5",6);
  TextField page00TextFieldVertical =
                                    new TextField("0.5",6);
  //Components for radio buttons
  CheckboxGroup Page00Group = new CheckboxGroup();
  Checkbox page00NearestNeighbor = new Checkbox(
       "Nearest Neighbor Interpolation",Page00Group,false);
  Checkbox page00Bilinear = new Checkbox(
               "Bilinear Interpolation",Page00Group,false);
  Checkbox page00Bicubic = new Checkbox(
                 "Bicubic Interpolation",Page00Group,true);

  //Components used to construct the Translation page.
  // Components that require local access only are defined
  // locally.  Others are defined here as instance
  // variables.
  Panel page01 = new Panel();
  TextField page01TextFieldHorizontal =
                                      new TextField("5",6);
  TextField page01TextFieldVertical =
                                     new TextField("10",6);

  //Components used to construct the Rotation page.
  // Components that require local access only are defined
  // locally.  Others are defined here as instance
  // variables.
  Panel page02 = new Panel();
  TextField page02TextField = new TextField("45.0",6);

  //Components used to construct the Mirror Image page.
  // Components that require local access only are defined
  // locally.  Others are defined here as instance
  // variables.
  Panel page03 = new Panel();

  //------------------------------------------------------//

  //This is the primary constructor.  It calls other
```

```
// methods to separate the construction of the GUI into
// easily understandable units.  Each method that it
// calls constructs one page in the tabbed pane.
ImgMod40(){//constructor

  constructPage00();
  tabbedPane.add(page00);//Add page to the tabbedPane.

  constructPage01();
  tabbedPane.add(page01);//Add page to the tabbedPane.

  constructPage02();
  tabbedPane.add(page02);//Add page to the tabbedPane.

  constructPage03();
  tabbedPane.add(page03);//Add page to the tabbedPane.

  add(tabbedPane);//Add tabbedPane to the Frame.

  setTitle("Copyright 2006, R.G.Baldwin");
  setBounds(555,0,470,300);
  setVisible(true);

  //Define a WindowListener to terminate the program.
  addWindowListener(
    new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(1);
      }//end windowClosing
    }//end windowAdapter
  );//end addWindowListener
}//end constructor
//-----------------------------------------------------//

//This method constructs the Scaling page.
// This method is called from the primary constructor.
void constructPage00(){
  page00.setName("Scaling");//Label on the tab.
  page00.setLayout(new BorderLayout());

  //Create and add the instructional text to the page.
  // This text appears in a disabled text area at the
  // top of the page in the tabbed pane.
  String text ="IMAGE SCALING\n"
    + "This page illustrates the scaling of an image "
    + "using three different types of interpolation."
    + "\n\n"
    + "Enter a positive scale factor between 0.001 and "
    + "10.0 in each of the text fields, select an "
    + "interpolation type, and click the Replot button. "
    + "The best quality interpolation will probably be "
    + "achieved with the use of Bicubic Interpolation. "
    + "Nearest Neighbor Interpolation will probably "
    + "produce the poorest quality.";

  //Note:  The number of columns specified for the
```

```
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,7,1,
                                 TextArea.SCROLLBARS_NONE);
    page00.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to the page.
    Panel page00ControlPanel = new Panel();
    page00ControlPanel.setLayout(new GridLayout(5,1));

    //Construct and populate the panels that contain the
    // radio buttons, the labels, and the text fields.
    // Add each such panel an a new cell in the grid
    // layout going from the top of the grid to the bottom
    // of the grid.

    //Begin with the radio buttons.  The purpose of putting
    // the radio buttons on panels is to cause them to be
    // left justified in their cells.
    Panel subControlPanel00 = new Panel();
    subControlPanel00.setLayout(new FlowLayout(
                                     FlowLayout.LEFT));
    subControlPanel00.add(page00NearestNeighbor);
    page00ControlPanel.add(subControlPanel00);

    Panel subControlPanel01 = new Panel();
    subControlPanel01.setLayout(new FlowLayout(
                                     FlowLayout.LEFT));
    subControlPanel01.add(page00Bilinear);
    page00ControlPanel.add(subControlPanel01);

    Panel subControlPanel02 = new Panel();
    subControlPanel02.setLayout(new FlowLayout(
                                     FlowLayout.LEFT));
    subControlPanel02.add(page00Bicubic);
    page00ControlPanel.add(subControlPanel02);

    //Now create and populate panels that contain labels
    // and associated TextField objects
    Panel subControlPanel03 = new Panel();
    subControlPanel03.setLayout(new FlowLayout(
                                     FlowLayout.LEFT));
    subControlPanel03.add(new Label(
                           "Horizontal Scale Factor"));
    subControlPanel03.add(page00TextFieldHorizontal);
    page00ControlPanel.add(subControlPanel03);

    Panel subControlPanel04 = new Panel();
    subControlPanel04.setLayout(new FlowLayout(
                                     FlowLayout.LEFT));
    subControlPanel04.add(new Label(
                             "Vertical Scale Factor"));
    subControlPanel04.add(page00TextFieldVertical);
    page00ControlPanel.add(subControlPanel04);
    page00.add(page00ControlPanel,BorderLayout.CENTER);
  }//end constructPage00
  //-------------------------------------------------------//

  //This method processes the image according to the
  // controls located on the Scaling page.
  //This method uses the AffineTransformOp filter class to
  // process the image.
  //The method is called from within the switch statement
  // in the method named processImg, which is the primary
  // image-processing method in this program.
  //This method illustrates image scaling, giving the user
  // a choice of three different interpolation schemes.
  //See the earlier lesson at
  // http://www.developer.com/java/other/article.php
  // /626051
  // for additional information on the use of a scaling
  // affine transform.
  BufferedImage processPage00(BufferedImage theImage){

    //Set a non-zero default value for the horizontal scale
    // factor.
    double horizontalScale = 0.001;
    try{//Get horizontalScale from the text field.
      horizontalScale = Double.parseDouble(
                      page00TextFieldHorizontal.getText());
    }catch(java.lang.NumberFormatException e){
      page00TextFieldHorizontal.setText("Bad Input");
      horizontalScale = 0.001; //Override bad user input.
    }//end catch

    //Guarantee reasonable values for horizontal scale
    if((horizontalScale < 0.001) ||
                             (horizontalScale > 10.0)){
      page00TextFieldHorizontal.setText("Bad Input");
      horizontalScale = 0.001;//Override bad user input.
    }//end if

    //Set a non-zero default value for the vertical scale
    // factor.
    double verticalScale = 0.001;
    try{//Get verticalScale from the text field.
      verticalScale = Double.parseDouble(
                        page00TextFieldVertical.getText());
    }catch(java.lang.NumberFormatException e){
      page00TextFieldHorizontal.setText("Bad Input");
      verticalScale = 0.001; //Override bad user input.
    }//end catch

    //Guarantee reasonable values for verticalScale
    if((verticalScale < 0.001) || (verticalScale > 10.0)){
      page00TextFieldHorizontal.setText("Bad Input");
      verticalScale = 0.001;//Override bad user input.
    }//end if
    //Get the selected interpolation scheme from the radio
    // buttons and reflect that selection in an int value
    // corresponding to the selected button.
    int interpolationScheme;
    if(page00Bicubic.getState() == true){
      interpolationScheme =
                         AffineTransformOp.TYPE_BICUBIC;
    }else if(page00Bilinear.getState() == true){
      interpolationScheme =
                        AffineTransformOp.TYPE_BILINEAR;
    }else{//page00NearestNeighbor must be selected
      interpolationScheme =
                AffineTransformOp.TYPE_NEAREST_NEIGHBOR;
    }//end else

    //An AffineTransform object is required later to create
    // the filter object.  An examination of the
    // documentation for the AffineTransform class will
    // show that there are several different ways to create
    // such an object.  The following statement is probably
    // the simplest of those ways.
    //Create an AffineTransform object for scaling that
    // matches the user input from the control panel.
    AffineTransform transformObj =
                      AffineTransform.getScaleInstance(
                         horizontalScale,verticalScale);

    //At this point, you could use the methods of the
    // AffineTransform class to get, modify, and restore
    // the transform matrix in order to modify the
    // behavior of the transformation process.  See the
    // earlier lesson at
    // http://www.developer.com/java/other/article.php
    // /626051
    // for additional information on this topic.

    //Use the AffineTransform object to create a filtering
    // object.
    AffineTransformOp filterObj = new AffineTransformOp(
                      transformObj,interpolationScheme);

    /*Note:  Normally, I would perform the filtering
     *operation and return the filtered result simply by
     *executing the following statement:

     *  return filterObj.filter(theImage, null);

     *However, for reasons that I am unable to explain,
     *when I do that for the AffineTransformOp class, the
     *ColorModel of the BufferedImage object that is
     *returned to the framework program named ImgMod05 is
     *not compatible with the method used by that program
     *to write the output JPEG file.  This results in an
     *output file in which the image data appears to be
     *scrambled.  Therefore, it was necessary for me to
     *use the following alternative code instead.*/
    //Create a destination BufferedImage object to receive
    // the filtered image.  Force the ColorModel of the
    // destination object to match the ColorModel of the
    // incoming object.
    BufferedImage dest =
                      filterObj.createCompatibleDestImage(
                        theImage,theImage.getColorModel());

    //Filter the image and save the filtered image in the
    // destination object.
    filterObj.filter(theImage, dest);

    //Return a reference to the destination object.
    return dest;

  }//end processPage00
  //-----------------------------------------------------//

  //This method constructs the Translation page.
  //The method is called from the primary constructor.
  void constructPage01(){
    page01.setName("Translation");//Label on the tab.
    page01.setLayout(new BorderLayout());

    //Create and add the instructional text to the page.
    // This text appears in a disabled text area at the
    // top of the page in the tabbed pane.
    String text ="IMAGE TRANSLATION\n"
      + "This page illustrates the translation of an "
      + "image to a new location relative to the "
      + "upper-left corner of the container using Bicubic "
      + "Interpolation.\n\n"
      + "Enter the horizontal and vertical translation "
      + "distances in pixels into the text fields and "
      + "click the Replot button.\n\n"
      + "Note that the translation distances must be "
      + "between -1000 and +1000 pixels.  Also note that "
      + "negative translations may shift the image "
      + "completely out of the Frame on the top or the "
      + "left side of the image.";

    //Note:  The number of columns specified for the
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,9,1,
                                TextArea.SCROLLBARS_NONE);
    page01.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to the page.
    Panel page01ControlPanel = new Panel();
    page01ControlPanel.setLayout(new GridLayout(3,1));

    //Place each label and its corresponding text field
    // on a panel.  Place the panels in the cells in the
    // grid layout from top to bottom.  Note that there
    // is an empty cell at the bottom for cosmetic
    // purposes.
    Panel subControlPanel00 = new Panel();
    subControlPanel00.setLayout(
                          new FlowLayout(FlowLayout.LEFT));
    subControlPanel00.add(new Label(
             "Horizontal Translation Distance in Pixels"));
    subControlPanel00.add(page01TextFieldHorizontal);
    page01ControlPanel.add(subControlPanel00);

    Panel subControlPanel01 = new Panel();
    subControlPanel01.setLayout(
                          new FlowLayout(FlowLayout.LEFT));
    subControlPanel01.add(new Label(
               "Vertical Translation Distance in Pixels"));
    subControlPanel01.add(page01TextFieldVertical);
    page01ControlPanel.add(subControlPanel01);

    page01.add(page01ControlPanel,BorderLayout.CENTER);
  }//end constructPage01
  //-----------------------------------------------------//

  //This method processes the image according to the
  // controls located on the Translation page.
  //This method uses the AffineTransformOp filter class
  // to process the image.  The method is called from
  // within the switch statement in the method
  // named processImg, which is the primary image
  // processing method in this program.
  //This method illustrates image translation using Bicubic
  // Interpolation.
  BufferedImage processPage01(BufferedImage theImage){

    double horizontalDistance = 0.0;
    try{//Get horizontalDistance from the text field.
      horizontalDistance = Double.parseDouble(
                      page01TextFieldHorizontal.getText());
    }catch(java.lang.NumberFormatException e){
      page01TextFieldHorizontal.setText("Bad Input");
      horizontalDistance = 0.0; //Override bad user input.
    }//end catch

    //Guarantee reasonable values for horizontalDistance
    if((horizontalDistance < -1000.0) ||
                            (horizontalDistance > 1000.0)){
      page01TextFieldHorizontal.setText("Bad Input");
      horizontalDistance = 0.0;//Override bad user input.
    }//end if

    double verticalDistance = 0.0;
    try{//Get verticalDistance from the text field.
      verticalDistance = Double.parseDouble(
                        page01TextFieldVertical.getText());
    }catch(java.lang.NumberFormatException e){
      page01TextFieldHorizontal.setText("Bad Input");
      verticalDistance = 0.0; //Override bad user input.
    }//end catch

    //Guarantee reasonable values for verticalDistance
    if((verticalDistance < -1000.0) ||
                              (verticalDistance > 1000.0)){
      page01TextFieldHorizontal.setText("Bad Input");
      verticalDistance = 0.0;//Override bad user input.
    }//end if

    //Set the interpolation scheme to the best available.
    // Note that this page doesn't allow the user to
    // select the interpolation scheme.
    int interpolationScheme =
                            AffineTransformOp.TYPE_BICUBIC;

    //Create an AffineTransform object for translation that
    // matches the user input from the control panel.
    //Note that even though the actual translation is
    // performed in terms of integer pixels, the
    // getTranslateInstance method requires the horizontal
    // and vertical distances to be provided as type
    // double.  A positive horizontal distance will cause
    // the image to appear to move to the right in its
    // container and a negative horizontal distance will
    // cause the image to appear to move to the left.
    // Similarly, a positive vertical distance will cause
    // the image to appear to move down and a negative
    // vertical distance will cause the image to appear to
    // move up.
    AffineTransform transformObj =
                    AffineTransform.getTranslateInstance(
                      horizontalDistance,verticalDistance);

    //Use the AffineTransform object to create a filtering
    // object.
    AffineTransformOp filterObj = new AffineTransformOp(
                         transformObj,interpolationScheme);

    /*Note:  Normally, I would perform the filtering
     *operation and return the filtered result simply by
     *executing the following statement:

     *  return filterObj.filter(theImage, null);

     *However, for reasons that I am unable to explain,
     *when I do that for the AffineTransformOp class, the
     *ColorModel of the BufferedImage object that is
     *returned to the framework program named ImgMod05 is
     *not compatible with the method used by that program
     *to write the output JPEG file.  This results in an
     *output file in which the image data appears to be
     *scrambled.  Therefore, it was necessary for me to
     *use the following alternative code instead.*/
    //Create a destination BufferedImage object to receive
    // the filtered image.  Force the ColorModel of the
    // destination object to match the ColorModel of the
    // incoming object.
    BufferedImage dest =
                      filterObj.createCompatibleDestImage(
                        theImage,theImage.getColorModel());

    //Filter the image and save the filtered image in the
    // destination object.
    filterObj.filter(theImage, dest);

    //Return a reference to the destination object.
    return dest;

  }//end processPage01
  //-----------------------------------------------------//

  //This method constructs the Rotation page.
  //This method is called from the primary constructor.
  // It illustrates the translation of an image followed by
  // rotation of the translated image.
  void constructPage02(){
    page02.setName("Rotation");//Label on the tab.
    page02.setLayout(new BorderLayout());

    //Create and add the instructional text to the page.
    // This text appears in a disabled text area at the
    // top of the page in the tabbed pane.
    String text ="IMAGE Rotation\n"
      + "This page illustrates translation of an image "
      + "followed by Rotation of the same image using "
      + "Bicubic Interpolation.\n\n"
      + "The image is rotated around its center after "
      + "being translated to the right and down by a "
      + "distance that is sufficient to give it room to "
      + "rotate.\n\n"
      + "Enter the desired rotation angle in degrees and "
      + "click the Replot button.  Positive rotation "
      + "angles represent clockwise rotation and negative "
      + "rotation angles represent counter-clockwise "
      + "rotation.";

    //Note:  The number of columns specified for the
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,9,1,
                                 TextArea.SCROLLBARS_NONE);
    page02.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to the page.
    // Use a control panel with a GridLayout for
    // cosmetic purposes.  Note that there are two empty
    // cells at the bottom of the grid.
    Panel page02ControlPanel = new Panel();
    page02ControlPanel.setLayout(new GridLayout(3,1));

    //Place the label and the text field on a panel and
    // place that panel in the top cell in the grid.
    Panel subControlPanel00 = new Panel();
    subControlPanel00.setLayout(new FlowLayout(
                                      FlowLayout.LEFT));
    //Note, a positive value in degrees represents
    // clockwise rotation.
    subControlPanel00.add(new Label(
                                "Rotation in Degrees"));
    subControlPanel00.add(page02TextField);
    page02ControlPanel.add(subControlPanel00);

    page02.add(page02ControlPanel,BorderLayout.CENTER);
  }//end constructPage02
  //-----------------------------------------------------//

  //This method processes the image according to the
  // controls located on the Rotation page.
  //This method uses the AffineTransformOp filter class
  // to process the image.  The method is called from
  // within the switch statement in the method named
  // processImg, which is the primary image-processing
  // method in this program.
  //This method illustrates image Translation followed by
  //image Rotation using Bicubic Interpolation.
  BufferedImage processPage02(BufferedImage theImage){

    //Get the rotation angle in degrees.  A positive angle
    // in degrees corresponds to clockwise rotation.
    double rotationAngleInDegrees = 0.0;
    try{//Get rotationAngleInDegrees from the text field.
      rotationAngleInDegrees = Double.parseDouble(
                             page02TextField.getText());
    }catch(java.lang.NumberFormatException e){
      page02TextField.setText("Bad Input");
      rotationAngleInDegrees = 0.0;//Override bad input.
    }//end catch

    //Compute the rotation angle in radians.
    double rotationAngleInRadians =
                  rotationAngleInDegrees*Math.PI/180.0;

    //Set the interpolation scheme.
    int interpolationScheme =
                        AffineTransformOp.TYPE_BICUBIC;

    //Translate the image down and to the right far enough
    // that the corners won't be chopped off by the top and
    // left edges of the container when the image is
    // rotated by 45 degrees.

    //Get the length of half the diagonal dimension of the
    // image using the formula for the hypotenuse of a
    // right triangle.
    int halfDiagonal = (int)(Math.sqrt(
         theImage.getWidth()*theImage.getWidth() +
           theImage.getHeight()*theImage.getHeight())/2.0);

    //Set the horizontal and vertical translation
    // distances.
    int horizontalDistance =
                      halfDiagonal - theImage.getWidth()/2;
    int verticalDistance =
                     halfDiagonal - theImage.getHeight()/2;

    //Create an Affine Transform object that can be used
    // to translate the image by the distances computed
    // above.
    AffineTransform transformObj =
                    AffineTransform.getTranslateInstance(
                      horizontalDistance,verticalDistance);

    //Get a translation filter object based on the
    // AffineTransform object.
    AffineTransformOp filterObj = new AffineTransformOp(
              transformObj,AffineTransformOp.TYPE_BICUBIC);

    //Perform the translation and save the modified image
    // as type BufferedImage.  This image will be the input
    // to the rotation transform.
    BufferedImage translatedImage =
                          filterObj.filter(theImage, null);

    //Now rotate the image around a point that is at the
    // center of the original image.  Begin by getting a
    // rotation transform object.  The second and third
    // parameters specify the point about which the image
    // will be rotated.
    transformObj = AffineTransform.getRotateInstance(
                rotationAngleInRadians,
                horizontalDistance + theImage.getWidth()/2,
                verticalDistance + theImage.getHeight()/2);

    //Now get a rotation filter object based on the
    // transform object and the specified interpolation
    // scheme.
    filterObj = new AffineTransformOp(
                         transformObj,interpolationScheme);

    /*Note:  Normally, I would perform the filtering
     *operation and return the filtered result simply by
     *executing the following statement:

     *  return filterObj.filter(translatedImage, null);

     *However, for reasons that I am unable to explain,
     *when I do that for the AffineTransformOp class, the
     *ColorModel of the BufferedImage object that is
     *returned to the framework program named ImgMod05 is
     *not compatible with the method used by that program
     *to write the output JPEG file.  This results in an
     *output file in which the image data appears to be
     *scrambled.  Therefore, it was necessary for me to
     *use the following alternative code instead.*/

    //Create a destination BufferedImage object to receive
    // the filtered image.  Force the ColorModel of the
    // destination object to match the ColorModel of the
    // incoming object.
    BufferedImage dest =
               filterObj.createCompatibleDestImage(
                 translatedImage,theImage.getColorModel());

    //Filter the image and save the filtered image in the
    // destination object.
    filterObj.filter(translatedImage, dest);

    //Return a reference to the destination object.
    return dest;

  }//end processPage02
  //-------------------------------------------------------//

  //This method constructs the Mirror Image page.
  //This method is called from the primary constructor.
  void constructPage03(){
    page03.setName("Mirror Image");//Label on the tab.
    page03.setLayout(new BorderLayout());

    //Create and add the instructional text to the page.
    //This text appears in a disabled text area at the
    // top of the page in the tabbed pane.
    String text ="Mirror Image\n\n"
      + "This page translates the image to the right by "
      + "an amount equal to its width, and then flips it "
      + "around its left edge to produce a mirror image "
      + "of the original image.\n\n"
      + "Click the Replot button to create the mirror "
      + "image.";

    //Note:  The number of columns specified for the
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,6,1,
                                 TextArea.SCROLLBARS_NONE);
    page03.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

  }//end constructPage03
  //-------------------------------------------------------//

  //This method processes the image according to the
  // Mirror Image page.
  //This method uses the AffineTransformOp filter class to
```

```
// process the image.  The method is called from within
// the switch statement in the method named processImg,
// which is the primary image processing method in this
// program.
//Note that unlike Scaling, Translation, and Rotation,
// there is no affine transform for Mirror Image. Rather,
// this method illustrates the use of horizontal
// translation followed by scaling with negative scale
// factors to produce a mirror image of the original
// image.
BufferedImage processPage03(BufferedImage theImage){

  //Get an AffineTransform object that can be used to
  // shift the image to the right by an amount equal to
  // its width.
  AffineTransform transformObj =
                  AffineTransform.getTranslateInstance(
                                theImage.getWidth(),0);

  //Concatenate this transform with a scaling
  // transformation.
  transformObj.scale(-1.0, 1.0);

  //Display the six values in the transformation matrix.
  double[] theMatrix = new double[6];
  transformObj.getMatrix(theMatrix);

  //Display first row of values by displaying every
  // other element in the array starting with element
  // zero.
  for(int cnt = 0; cnt < 6; cnt+=2){
    System.out.print(theMatrix[cnt] + "\t");
  }//end for loop

  //Display second row of values displaying every
  // other element in the array starting with element
  // number one.
  System.out.println();//new line
  for(int cnt = 1; cnt < 6; cnt+=2){
    System.out.print(theMatrix[cnt] + "\t");
  }//end for loop
  System.out.println();//end of line
  System.out.println();//blank line

  //Get a translation filter object based on the
  // AffineTransform object.
  AffineTransformOp filterObj = new AffineTransformOp(
          transformObj,AffineTransformOp.TYPE_BICUBIC);

  /*Note:  Normally, I would perform the filtering
   *operation and return the filtered result simply by
   *executing the following statement:

   *  return filterObj.filter(theImage, null);

   *However, for reasons that I am unable to explain,
     *when I do that for the AffineTransformOp class, the
     *ColorModel of the BufferedImage object that is
     *returned to the framework program named ImgMod05 is
     *not compatible with the method used by that program
     *to write the output JPEG file.  This results in an
     *output file in which the image data appears to be
     *scrambled.  Therefore, it was necessary for me to
     *use the following alternative code instead.*/

    //Create a destination BufferedImage object to receive
    // the filtered image.  Force the ColorModel of the
    // destination object to match the ColorModel of the
    // incoming object.
    BufferedImage dest =
                      filterObj.createCompatibleDestImage(
                        theImage,theImage.getColorModel());

    //Filter the image and save the filtered image in the
    // destination object.
    filterObj.filter(theImage, dest);

    //Return a reference to the destination object.
    return dest;

  }//end processPage03
  //-------------------------------------------------------//

  //The following method must be defined to implement the
  // ImgIntfc05 interface.  It is called by the framework
  // program named ImgMod05.
  public BufferedImage processImg(BufferedImage theImage){

    BufferedImage outputImage = null;

    //Process the page in the tabbed pane that has been
    // selected by the user.
    switch(tabbedPane.getSelectedIndex()){
      case 0:outputImage = processPage00(theImage);
             break;
      case 1:outputImage = processPage01(theImage);
             break;
      case 2:outputImage = processPage02(theImage);
             break;
      case 3:outputImage = processPage03(theImage);
             break;
    }//end switch

    return outputImage;
  }//end processImg
}//end class ImgMod40
```

Listing 29

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

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

Keywords
java 2D image pixel framework filter AffineTransform AffineTransformOp

-end-