Using the Java 2D LookupOp Filter Class to Process Images

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

Published: January 16, 2007
By Richard G. Baldwin

Java Programming Notes # 452

Preface

One lesson in a series

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

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

- **LookupOp**
- **AffineTransformOp**
- **BandCombineOp**
- **ConvolveOp**
- **RescaleOp**
- **ColorConvertOp**

In this lesson, I will keep that promise and teach you how to use the **LookupOp** class for a variety of purposes. I will teach you how to use the other 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 computers 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.

The Replot button

The **ImgMod05** program GUI contains a **Replot** button *(as shown in Figure 1)*. 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 modified image.
- The resulting modified image is displayed along with the original image.
- The modified 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 1)*. 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 two different image-processing programs that are compatible with the framework program named **ImgMod05**.

The program named ImgMod38

The first program, named **ImgMod38**, is designed to show the *essential requirements* of writing a program that runs under control of the framework program named **ImgMod05** and uses an image-filtering class of the Java 2D API to modify an image.

The program named ImgMod39

The second program, named **ImgMod39**, is designed to show how you can expand on those essential requirements to create a program that allows for user input and supports a great deal of experimentation.

Discussion and Sample Code

I will discuss the two programs in this lesson in fragments. A complete listing of **ImgMod38** is provided in Listing 37 near the end of the lesson. A complete listing of **ImgMod39** is provided in Listing 38 near the end of the lesson.

Complete listings of the framework program named **ImgMod05** and its required interface named **ImgIntfc05** were provided at the end of the earlier lesson entitled A Framework for Experimenting with Java 2D Image-Processing Filters.

The Program Named ImgMod38

Purpose

The purpose of this program is to provide a simple example of an image-processing class that is compatible with the use of the framework program named **ImgMod05**, and which illustrates a single usage of the **LookupOp** image-filtering class from the Java 2D API.

Secondary purpose
A secondary purpose of this program is to provide program comments that will be referred to by future programs to avoid repetition of those comments.

Must implement the interface named ImgIntfc05

A class that is compatible with the framework program named **ImgMod05** is required to implement the interface named **ImgIntfc05**. This, in turn, requires the class to define the method named **processImg**, which receives one parameter of type **BufferedImage** and returns a reference of type **BufferedImage**.

The required signature for the **processImg** method is:

```
public BufferedImage processImg(BufferedImage input);
```

The **processImg** method receives a reference to a **BufferedImage** object containing the image that is to be processed. The **processImg** method must return a reference to a **BufferedImage** object containing the processed image.

A color-inverter method

In this example, the method named **processImg** is a color inverter method. Figure 1 shows a sample of the output produced by this program when it is run under control of the framework program named **ImgMod05**.

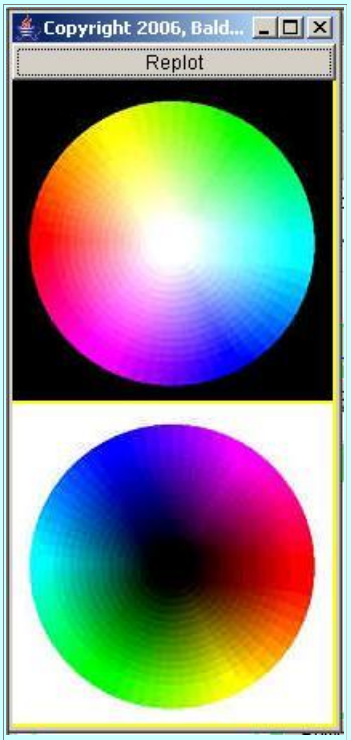

Figure 1

Inverting the colors

The method named **processImg** as defined in this class receives an incoming reference to an image as a parameter of type **BufferedImage**. The method returns a reference to an image as type **BufferedImage** where all of the color values in the pixels have been inverted by subtracting the color values from 255.

The type of the image

The method has been demonstrated to work properly only for the case where the incoming **BufferedImage** object was constructed for image type **BufferedImage.TYPE_INT_RGB**. However, it may work properly for other image types as well.

No constructor parameters are allowed

Note that this class does not define a constructor. However, if it did define a constructor, that constructor would not be allowed to receive parameters. This is because the class named **ImgMod05** instantiates an object of this class by invoking the **newInstance** method of the class named **Class** using the name of this class as a **String**. That process does not allow for constructor parameters for the class being instantiated.

Program output

The framework program named **ImgMod05** displays the original and the modified images in the format shown in Figure 1, with the original image at the top and the modified image at the bottom.

The program named **ImgMod05** 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.

The Replot button

As shown in Figure 1, the GUI for the framework program named **ImgMod05** contains a **Replot** button. 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 the specified image processing class is invoked.
- The resulting modified image is displayed along with the original image.
- The modified image is written into the output JPEG file named **junk.jpg**.

An image-processing program GUI

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. However, no such GUI is provided by this program and clicking the **Replot** button is of no consequence.

ImgMod39
The use of an image-processing program GUI will be illustrated by the program named **ImgMod39**.

File formats

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

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

Usage

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

```
java ImgMod05 ImgMod38 ImageFileName
```

The image file must be provided by the user. However, it doesn't have to be in the current directory if a path to the file is included along with the file name on the command line.

The size of the display frame

When the 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 1.

The framework program named **ImgMod05** attempts to adjust the size of the display frame to accommodate both images. However, if the processed image doesn't fit in the display, the user can manually resize the display frame in order to view both images.

Program testing

This program was tested using J2SE5.0 under WinXP.

The class named ImgMod38

This program consists of the single class named **ImgMod38**. That class begins in Listing 1.

```
class ImgMod38 implements ImgIntfc05{

  public BufferedImage processImg(BufferedImage
theImage){

Listing 1
```

As shown in Listing 1, the class implements the required interface named **ImgIntfc05**. This in turn requires the class to define the method named **processImg**.

The beginning of the required method definition also begins in Listing 1.

A lookup table is required

This class uses the **LookupOp** class from the Java 2D API to invert all of the color values in the pixels. *(The alpha value belonging to the pixel is not modified.)*

The **filter** method that is later invoked on an object of the **LookupOp** class to modify the image uses a color value from a pixel as an ordinal index into a lookup table. It replaces the color value in the pixel with the value stored in the lookup table at that index. Thus, you can modify the color values in the pixels using just about any substitution algorithm that you can devise.

Constructing the lookup table

The lookup table can be constructed as an object of either of the following classes, both of which are subclasses of the abstract **LookupTable** class:

- **ByteLookupTable**
- **ShortLookupTable**

The examples in this lesson will use lookup tables constructed from the **ShortLookupTable** class.

One or more substitution arrays can be used

The **LookupTable** object can be constructed from one or more data arrays containing substitution values. If a single substitution array is used to construct the lookup table, that substitution array is applied to each of the red, green, and blue color bands. If three different substitution arrays are used to construct the lookup table, different substitution arrays are applied to each of the three color bands. This makes it possible to process the three color bands in different ways.

This program uses a single array to construct the lookup table. The use of three different arrays to construct the lookup table will be illustrated later in the program named **ImgMod39**.

Prepare the data for the lookup table

Listing 2 creates a single data array of type **short**, containing 256 elements, where the value of each element is equal to 255 minus the element index. This is the basis of the color inversion algorithm illustrated by Figure 1.

```
    short[] lookupData = new short[256];

    for (int cnt = 0; cnt < 256; cnt++){
      lookupData[cnt] = (short)(255-cnt);
    }//end for loop

Listing 2
```

Create the lookup table

Listing 3 instantiates a new object of the **ShortLookupTable** class, passing a reference to the substitution array object as the second parameter to the constructor for the class.

```
    ShortLookupTable lookupTable =
                          new
ShortLookupTable(0,lookupData);

Listing 3
```

Two overloaded constructors available

As of the date of this writing, there are two overloaded constructors for the **ShortLookupTable** class. The version used in Listing 3 is the version that should be used when the values in a single substitution array are to be applied to the pixel color values in all three of the red, green, and blue color bands.

The other overloaded constructor
The other overloaded version of the constructor will be used in the program named **ImgMod39** later in this lesson.

The first parameter to the constructor

According to Sun, the first constructor parameter is an offset, the value of which is *"subtracted from the input values before indexing into the array."* In other words, this makes it possible to specify a set of 256 consecutive elements in the substitution array when the **length** of the array is greater than 256. In this program, the **length** of the substitution array is 256 and the offset value is 0.

Create the filter object

Listing 4 instantiates a new object of the **LookupOp** class and saves that object's reference in a variable of type **BufferedImageOp**.

```
    BufferedImageOp thresholdOp =
                              new
LookupOp(lookupTable,null);

Listing 4
```

The BufferedImageOp interface

BufferedImageOp is an interface that is implemented by the **LookupOp** class, and by all of the image-filtering classes in the earlier list except for the **BandCombineOp** class.

Some other differences
The **BandCombineOp** class differs from the other image-filtering classes in some other ways as well, which I will describe in a future lesson that

The LookupOp constructor

As of the date of this writing, there is only one constructor for the **LookupOp** class. The first parameter to the constructor is a reference to the lookup table object that was created in Listing 3. The second parameter is an optional reference to a **RenderingHints** object.

Note that I elected not use a **RenderingHints** object for the programs in this lesson.

A RenderingHints object
Briefly, a **RenderingHints** object makes it possible for you to have some control over the quality of the final rendering of the image by controlling properties such as dithering and anti-aliasing.

Apply the filter to the incoming image

Listing 5 invokes the **filter** method of the **LookupOp** class to filter the incoming image and to return a reference to the filtered image object as type **BufferedImage**.

```
    return thresholdOp.filter(theImage, null);
  }//end processImg

}//end class ImgMod38

Listing 5
```

Two overloaded versions of the filter method

As of the date of this writing, there are two overloaded versions of the **filter** method defined for the **LookupOp** class.

The overloaded version used in Listing 5 accepts a reference to the incoming image as type **BufferedImage** and returns a reference to the filtered image as type **BufferedImage**. The other version accepts a reference to the incoming image as type **Raster**, and returns a reference to the filtered image as type **WritableRaster**.

The second parameter

The second parameter to the **filter** method in Listing 5 can optionally specify an existing **BufferedImage** object to serve as a destination for the processed image, in addition to returning a reference to the processed image object.

Will defer explanation of the Raster class

With the exception of the **BandCombineOp** class, all of the classes in the earlier list can operate on the image either as type **BufferedImage** or as type **Raster**. However, the **BandCombineOp** class can only operate on images as type **Raster**. Therefore, I will defer a discussion of the use of the **Raster** class until a future lesson in which I will explain the use of the **BandCombineOp** class.

The end of the program named ImgMod38

Listing 5 signals the end of the **processImg** method as well as the end of the program named **ImgMod38**. Hopefully, this simple program has taught you the essentials of using the framework program named **ImgMod05**, along with the **LookupOp** image filtering class of the Java 2D API for the filtering of images.

As you can see, when there is no need to provide for a variation in image-processing parameters, it is a simple task to write an *ImgMod05-compatible* program to implement a color substitution algorithm. As an upgrade, the program named **ImgMod39** will provide for user-defined image-processing parameters.

The Program Named ImgMod39

Image-processing programs such as this one may provide a GUI for user 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.

Purpose of the program

The purpose of this program is to illustrate a variety of different uses for the **LookupOp** class of the Java 2D API. Three such uses are illustrated by the three tabbed pages in the program GUI shown in Figure 2, Figure 3, and Figure 4.

Color inversion

The GUI for this program is constructed using a **JTabbedPane** object. The page shown in Figure 2 makes it possible for the user to invert the colors on none, or on any combination of the red, green, and blue color bands of the input image.

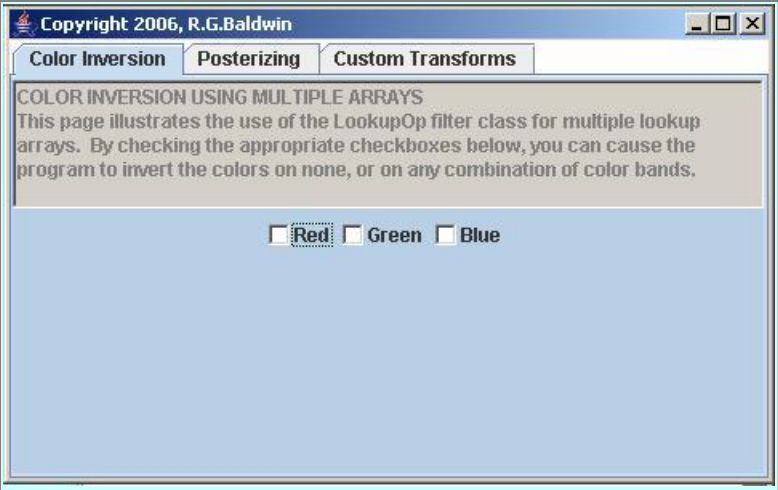

Figure 2

Posterizing

The page shown in Figure 3 makes it possible for the user to apply a process to the image commonly known as *posterizing* the image. I will explain what is meant by *posterizing* in conjunction with the explanation of the program later in this lesson.

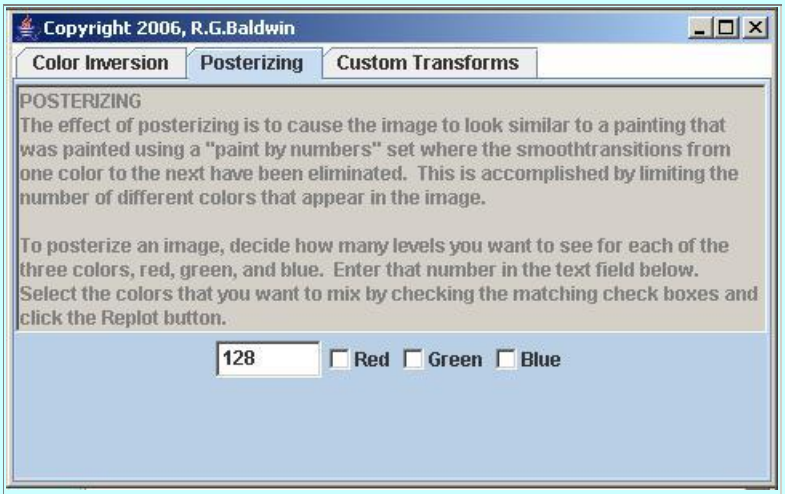

Figure 3

Custom transforms

The page shown in Figure 4 makes it possible for the user to apply either of two different custom transforms to the image. I will explain the custom transforms in conjunction with the explanation of the program later.

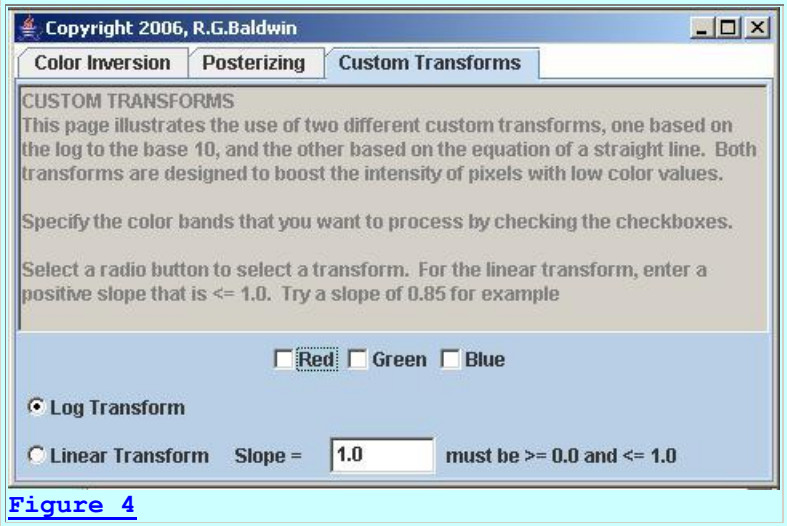

Figure 4

Three substitution arrays are used

In each of the three cases illustrated by Figure 2 through Figure 4, the program uses three different substitution arrays to process the red, green, and blue color bands individually or in combination *(as opposed to the use of a single substitution array in the case of the earlier program named **ImgMod38**)*.

The procedure for running the program

The general comments provided for the program named **ImgMod38** apply to this program also. Enter the following at the command line to run this program:

```
java ImgMod05 ImgMod39 ImageFileName
```

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

A JTabbedPane object

As mentioned earlier, this program creates a GUI consisting of a **JTabbedPane** object containing three pages. As shown in Figure 2, the tabs on the pages are labeled:

- Color Inversion
- Posterizing
- Custom Transforms

Each page in the tabbed pane contains a set of controls that make it possible to process an image in a way that illustrates the image-processing concepts indicated by the labels on the tabs. Processing details for each page will be provided in the discussion of the code later.

A complete program listing

A complete listing of the program is provided in Listing 38 near the end of the lesson. The program was tested using J2SE 5.0 under WinXP.

The ImgMod39 class

The class definition for the **ImgMod39** class begins in Listing 6. The class extends the **Frame** class, making an object of the class eligible to serve as its own GUI. The class also implements the **ImgIntfc05** interface making it eligible for being executed under control of the program named **ImgMod05.**

```
class ImgMod39 extends Frame implements
ImgIntfc05{

  JTabbedPane tabbedPane = new JTabbedPane();

Listing 6
```

Listing 6 also instantiates an object of the **JTabbedPane** class, which serves as the primary GUI as shown in Figure 2.

Components for the *Color Inversion* page

Listing 7 declares and initializes some of the components required to construct the *Color Inversion* page shown in Figure 2. Those components that require local access only are defined locally where they are needed. The others are defined in Listing 7 as instance variables.

```
  Panel page00 = new Panel();
  Checkbox page00RedCkBx = new Checkbox("Red");
  Checkbox page00GreenCkBx = new
Checkbox("Green");
  Checkbox page00BlueCkBx = new
Checkbox("Blue");

Listing 7
```

Components for the *Posterizing* page

Listing 8 declares and initializes some of the components needed to construct the *Posterizing* page shown in Figure 3. Once again, components that require local access only are defined locally where they are needed. Others are defined in Listing 8 as instance variables.

```
  Panel page01 = new Panel();
  TextField page01TextField = new
TextField("128",6);
  Checkbox page01RedCkBx = new Checkbox("Red");
  Checkbox page01GreenCkBx = new
Checkbox("Green");
  Checkbox page01BlueCkBx = new
Checkbox("Blue");
```

Listing 8

Components for the *Custom Transforms* page

Finally, Listing 9 declares and initializes some of the components needed to construct the *Custom Transforms* page shown in Figure 4. As before, components that require local access only are defined locally close to where they are needed. Others are defined in Listing 9 as instance variables.

```
  Panel page02 = new Panel();
  TextField page02TextField = new
TextField("1.0",6);
  Checkbox page02RedCkBx = new Checkbox("Red");
  Checkbox page02GreenCkBx = new
Checkbox("Green");
  Checkbox page02BlueCkBx = new
Checkbox("Blue");
  CheckboxGroup group = new CheckboxGroup();
  Checkbox page02LogRadioButton =
                  new Checkbox("Log
Transform",group,true);
  Checkbox page02LinearRadioButton =
              new Checkbox("Linear
Transform",group,false);
```

Listing 9

The constructor

Listing 10 shows the constructor for the **ImgMod39** class. Recall that this constructor is not allowed to receive parameters because of the way that an object of the class is instantiated *(using the **newInstance** method of the class named **Class**).*

Sub-divide the construction process

Listing 10 contains the primary constructor. The code in Listing 10 calls other methods to construct the individual GUI pages shown in Figure 2 through Figure 4. This serves to separate the construction of the GUI into easily understandable units. Each method that it calls constructs one page in the tabbed pane.

```
  ImgMod39(){//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.

    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 10

The code in Listing 10 is straightforward and shouldn't require further explanation.

The Color Inversion Page

Construct the Color Inversion page

Listing 11 shows the beginning of the method named **constructPage00**, which is used to construct the *Color Inversion* page shown in Figure 2. This method is called from the primary constructor shown in Listing 10.

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

Listing 11

The label on the tab

Listing 11 sets the ***name*** property of the page *(which is actually a **Panel** object)* to the **String** value *Color Inversion*. This property is used by the system to establish the label that appears on the tab in Figure 2.

The layout manager

Listing 11 also sets the ***layout*** property for the page to **BorderLayout**. The **TextArea** object shown in Figure 2 will be placed in the **BorderLayout.NORTH** location on the page. The three **Checkbox** objects will be placed in another **Panel** object, which in turn will be placed in the **BorderLayout.CENTER** location on the page.

Create and place the TextArea object

Listing 12 creates and populates a disabled **TextArea** object containing usage instructions and adds it to the page. You can view this component in Figure 2.

```
    String text ="COLOR INVERSION USING
MULTIPLE ARRAYS\n"
      + "This page illustrates the use of the
LookupOp "
      + "filter class for multiple lookup
arrays.  By "
      + "checking the appropriate checkboxes
below, you "
      + "can cause the program to invert the
colors on "
      + "none, or on any combination of color
bands.";

    //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,4,1,

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

Listing 12
```

The code in Listing 12 is straightforward and shouldn't require further explanation.

Construct the control panel

Listing 13 constructs a control panel containing the three **Checkbox** objects and adds it to the page.

```
    Panel page00ControlPanel = new Panel();
    page00ControlPanel.add(page00RedCkBx);
    page00ControlPanel.add(page00GreenCkBx);
    page00ControlPanel.add(page00BlueCkBx);

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

Listing 13

Listing 13 also signals the end of the method named **constructPage00**.

An aside - The processImg method

Lest we forget, in order to be compatible with the framework program named **ImgMod05**, this class must define a method named **processImg** with a signature matching that given in the interface named **ImgIntfc05**. This method is called by the framework program named **ImgMod05** in order to process the image.

The **processImg** method is shown in its entirety in Listing 14. It receives a reference to an input image encapsulated in an object of type **BufferedImage** and returns a reference to a modified version of the image encapsulated in another object of type **BufferedImage**.

```
  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;
    }//end switch

    return outputImage;
  }//end processImg
```

Listing 14

The switch statement

The most interesting thing about Listing 14 is the use of a **switch** statement, in conjunction with the **getSelectedIndex** method of the **JTabbedPane** class, to determine which page in Figure 2 has been selected by the user.

The return value from the **getSelectedIndex** method is 0 if the user has selected the leftmost tab *(the Color Inversion page*) in Figure 2. The return value increases by 1 to indicate the selection of each successive tab going from left to right in Figure 2. Depending on the value of the returned index value, the **switch** statement causes one of the following three methods to be invoked to process the image:

- processPage00
- processPage01
- processPage02

Returning the processed image

In each case, the reference to the modified image that is returned by the image-processing method is stored in the variable named **outputImage**. That reference is then returned to the framework program named **ImgMod05** as the return value from the method named **processImg**.

The processPage00 method

The method named **processPage00** begins in Listing 15. This method is called to process the image whenever the user selects the *Color Inversion* page in Figure 2 and then clicks the **Replot** button in Figure 1.

The processPage00 method
The method named **processPage00** is also called as the default image-processing method at startup.

This method processes the image according to the check boxes that have been checked by the user in Figure 2. The method uses the **LookupOp** image-filtering class to process the image using lookup data from three separate arrays, one each for the red, green, and blue color bands.

This method performs color inversion as illustrated in Figure 1. *(The effect of color inversion is to produce an output in which the image is similar to the negative of a color photograph.)*

Why invert?

I explained in the earlier lesson entitled A Framework for Experimenting with Java 2D Image-Processing Filters that color inversion has certain properties that make it useful for a variety of purposes. One of those uses is changing the colors in an image that has been *selected* during an editing process.

For example, Figure 5 below shows a cropped screen shot taken from the WYSIWYG HTML editor that I am using to write this lesson with the image in Figure 1 above having been *selected* in the editor. *(Selection of the image for editing caused the colors in the image to change.)*

Figure 5

The colors are inverted

If you compare Figure 5 with Figure 1, you will see that *selection* of the image in Figure 1 caused all of the colors to be inverted. As a result, the top image in Figure 1 looks like the bottom image in Figure 5, and the top image in Figure 5 looks like the bottom image in Figure 1.

Enough talk, let's see some code

The method named **processPage00** begins in Listing 15. As you can see, the method receives an incoming parameter referring to the input image object and returns a reference to the processed image.

```
  BufferedImage processPage00(BufferedImage
theImage){

    short[] red = null;
    short[] green = null;
    short[] blue = null;
```

Listing 15

Listing 15 declares references to three one-dimensional array objects of type **short[]** that will be populated with data that is used later to populate the lookup table.

Create a *straight* array object and an *inverting* array object

Listing 16 creates and populates two one-dimensional array objects of type **short[]**. One of the array objects *(noInvert)* is populated with data values that will simply reproduce the colors in the image being processed. The other array object *(invert)* is populated with data values that will invert the colors in the image being processed.

```
    short[] noInvert = new short[256];
    short[] invert = new short[256];
    for(int cnt = 0;cnt < 256;cnt++){
      invert[cnt] = (short)(255 -
cnt);//inverted data
      noInvert[cnt] = (short)cnt;//straight
lookup data
    }//end for loop

  }//end processPage00

Listing 16
```

Establish the default case of no color inversion

Listing 17 points the three references declared in Listing 15 to the array object containing data values that will simply reproduce the colors in the image being processed. This is the default case if no checkboxes are checked.

```
    red = noInvert;
    green = noInvert;
    blue = noInvert;

Listing 17
```

Re-point array references based on selected check boxes

Listing 18 examines the *Red*, *Green*, and *Blue* check boxes in Figure 2. If any checkbox has been checked, the code in Listing 18 modifies the reference to the substitution array for that color. If the checkbox has been checked, the reference is caused to point to the array object containing data values that will cause the colors in the image being processed to be inverted.

```
    if(page00RedCkBx.getState() == true){
      red = invert;
    }//end if
    if(page00GreenCkBx.getState() == true){
      green = invert;
    }//end if
    if(page00BlueCkBx.getState() == true){
```

```
      blue = invert;
    }//end if

Listing 18
```

Thus, any combination *(including none)* of the three color bands can be inverted by checking the appropriate checkbox in Figure 2 and then clicking the **Replot** button in Figure 1.

Inverting individual color bands

For example, whereas the bottom image in Figure 1 shows the result of inverting all three color bands in the color wheel in the top image, the bottom image in the left panel in Figure 6 shows the result of inverting only the red color band, and the bottom image in the right panel in Figure 6 shows the result of inverting only the green color band.

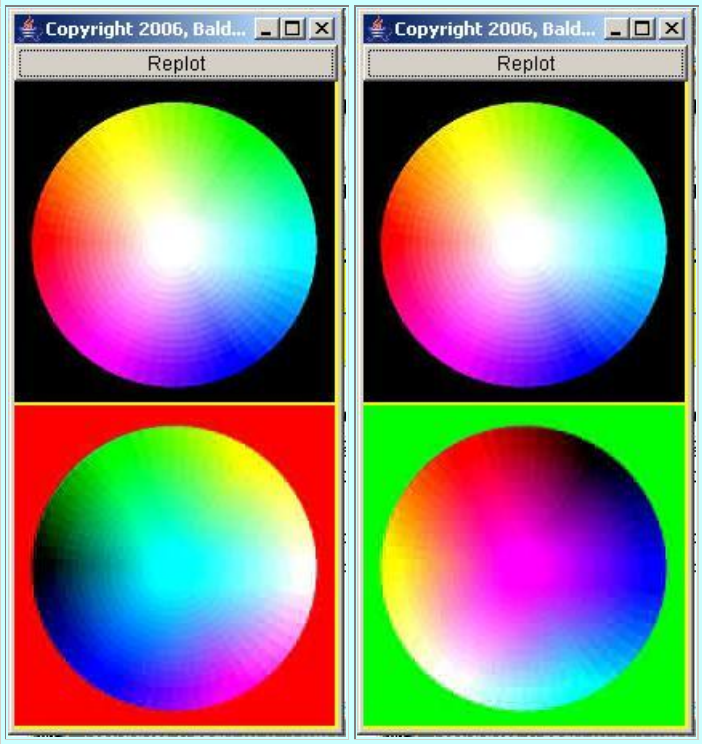

Figure 6

Process the image and return the result

Listing 19 invokes the method named **processImageForThePage** to actually process the image, passing a reference to the image along with the three substitution arrays to the method. *(This is a common method that will be used by all three image-processing schemes illustrated by this program.)*

```
    return
processImageForThePage(theImage,red,green,blue);
```

```
  }//end processPage00

Listing 19
```

Listing 19 signals the end of the method named **processPage00**.

The method named processImageForThePage

The common image-processing method named **processImageForThePage** begins in Listing 20.

This method uses the **LookupOp** class from the Java 2D API along with three separate data substitution arrays to process the color values in the corresponding color bands. One substitution array is applied to the red color band, one is applied to the green color band, and one is applied to the blue color band.

Create and populate a 2D array of type short[][]

Listing 20 creates and populates a 2D array of type **short[][]** with data for the lookup table. *(Note that this is a 2D array, rather than a 1D array as was the case in Listing 2 where a single substitution data array was used to process all three color bands.)*

```
  BufferedImage processImageForThePage(

BufferedImage theImage,
                                     short[]
red,
                                     short[]
green,
                                     short[]
blue){

    short[][] lookupData = new
short[][]{red,green,blue};

Listing 20
```

Create the lookup table and the filter object

Listing 21 executes code that is essentially the same as the code that was explained in Listing 3 and Listing 4 to create the lookup table and the filter object.

```
    //Create the lookup table.  The first
parameter is an
    // offset for extracting data from the
array object.
    //In this case, all of the data is
extracted from the
    // array object beginning at an index of 0.
    ShortLookupTable lookupTable =
```

```
                          new
ShortLookupTable(0,lookupData);

    //Create the filter object. The second
parameter
    // provides the opportunity to use
RenderingHints.
    BufferedImageOp filterObject =
                               new
LookupOp(lookupTable,null);
```

Listing 21

Display the lookup table

For illustration purposes only, the code in Listing 22 invokes a method named **displayTableData** to work backwards from the **filterObject** to get and display some data from the lookup table. Note that this is not an image-processing requirement.

```
    displayTableData(filterObject);
```

Listing 22

I will leave it up to you to examine and understand the code in the method named **displayTableData**, which you will find in Listing 38.

Apply the filter and return the filtered image

Listing 23 applies the filter to the incoming image and returns a reference to the resulting **BufferedImage** object.

```
    return filterObject.filter(theImage, null);
  }//end processImageForThePage
```

Listing 23

The code in Listing 23 is essentially the same as the code that I explained in Listing 5.

Listing 23 signals the end of the method named **processImageForThePage**.

Listing 23 also signals the end of the explanation for the *Color Inversion* page shown in Figure 2.

The Posterizing Page

Posterizing is a process of reducing the number of colors in an image to a relatively small number. The effect is to cause the image to look similar to a painting that was painted using a *"paint by numbers"* set where the smooth transitions from one color to the next have been

eliminated. This is accomplished by limiting the number of different colors that appear in the image.

For example, Figure 7 shows an image that has been posterized to what I believe should be 27 colors.

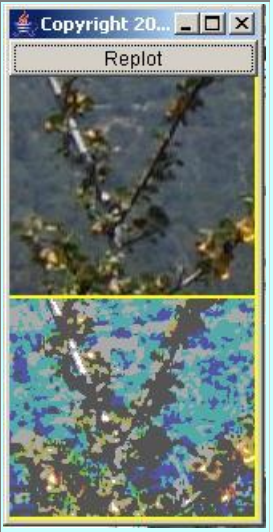

Figure 7

Posterizing results with a color wheel

The left panel of Figure 8 shows the result of eliminating all the colors from the color wheel except red and then posterizing the resulting image into only three shades of red.

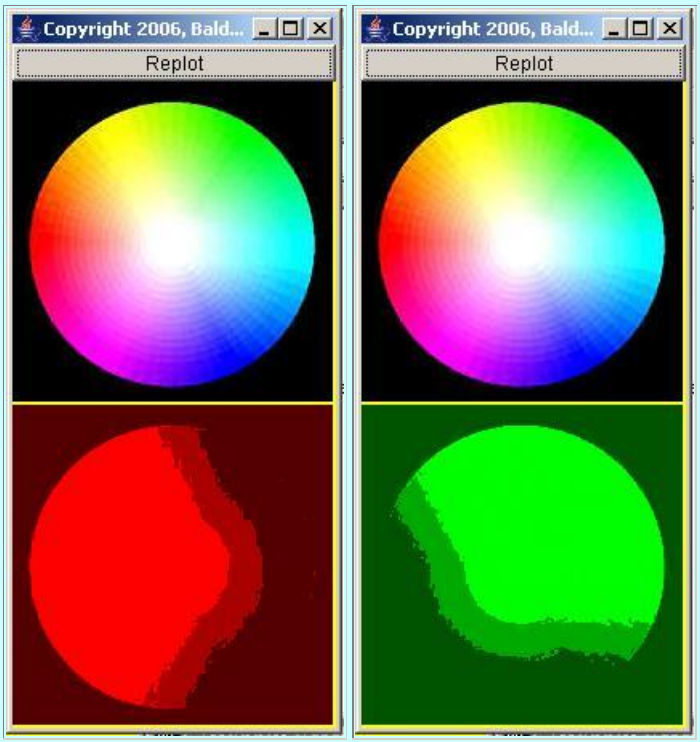

Figure 8

The right panel of Figure 8 shows the result of eliminating all the colors from the color wheel except green and then posterizing the resulting image into three shades of green.

Similarly, the left panel of Figure 9 shows the result of eliminating all the colors from the color wheel except blue and then posterizing the resulting image into three shades of blue.

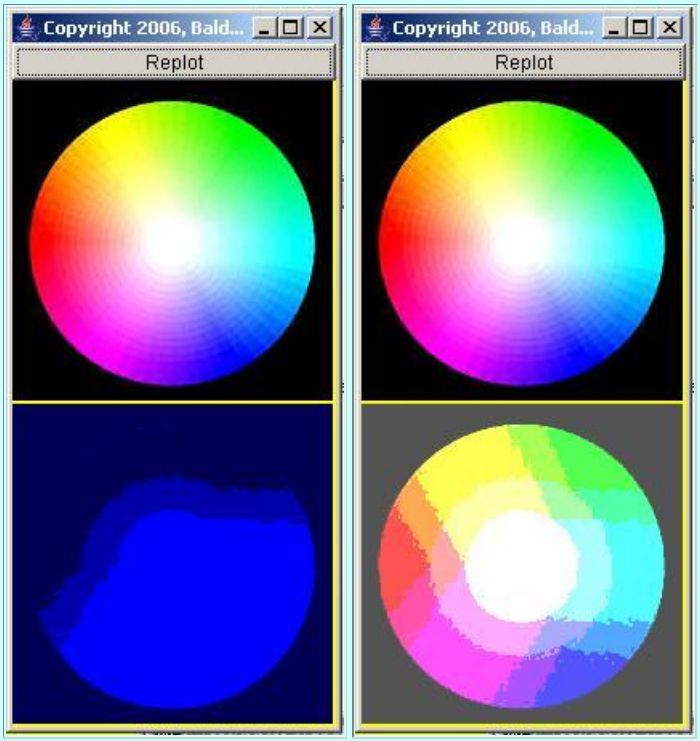

Figure 9

Finally, the right panel of Figure 9 shows the result of putting the posterized versions of red, green, and blue back together to produce a posterized version of the original color wheel.

Should have 27 colors

According to my calculations, when each of the three colors is allowed to have three shades, then the maximum number of possible colors in the posterized version of the color wheel should be three cubed or 27. However, the differences between some of those colors must be very subtle because I am only able to count about twenty different colors in the bottom image of Figure 9.

Now for some code

That is probably enough posterizing examples to give you an idea of the behavior of posterizing. Now let's look at the code that accomplishes posterizing.

The method named constructPage01

The method named **constructPage01** is shown in its entirety in Listing 24. This method constructs the *Posterizing* page shown in Figure 3. The method is called from the primary constructor in Listing 10.

```
  void constructPage01(){
    page01.setName("Posterizing");//Label on the
tab.
```

```
    page01.setLayout(new BorderLayout());

    //Create and add the instructional text to the
page.
    String text = "POSTERIZING\n"
      + "The effect of posterizing is to cause the
image "
      + "to look similar to a painting that was
painted "
      + "using a \"paint by numbers\" set where the
smooth"
      + "transitions from one color to the next
have been "
      + "eliminated.  This is accomplished by
limiting "
      + "the number of different colors that appear
in "
      + "the image.\n\n"
      + "To posterize an image, decide how many
levels "
      + "you want to see for each of the three
colors, "
      + "red, green, and blue.  Enter that number
in the "
      + "text field below.  Select the colors that
you "
      + "want to mix by checking the matching check
boxes "
      + "and click the Replot button.";

    //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.add(page01TextField);
    page01ControlPanel.add(page01RedCkBx);
    page01ControlPanel.add(page01GreenCkBx);
    page01ControlPanel.add(page01BlueCkBx);

page01.add(page01ControlPanel,BorderLayout.CENTER);

  }//end constructPage01
```

Listing 24

You should have no difficulty understanding the code in Listing 24 if you compare the code with the image in Figure 3. Therefore, further explanation of the code in Listing 24 should not be necessary.

The method named processPage01

The method named **processPage01** begins in Listing 25. This method applies a posterizing algorithm to the image using the controls located on the *Posterizing* page in Figure 3. The method is called from within the **switch** statement in the method named **processImg** in Listing 14. Note that this method processes the image using three separate arrays.

```
  BufferedImage processPage01(BufferedImage
theImage){

    int numberLevels = 1;
    try{//Get input value from the text field.
      numberLevels =

Integer.parseInt(page01TextField.getText());
    }catch(java.lang.NumberFormatException e){
      page01TextField.setText("Bad Input");
      numberLevels = 1;//Override bad user
input.
    }//end catch

    //Guarantee that the number of levels falls
within the
    // allowable range.  Don't allow divison by
0.
    if((numberLevels <= 0) || (numberLevels >'
256)){
      page01TextField.setText("Bad Input");
      numberLevels = 1;//Override bad user
input.
    }//end if
```

Listing 25

Get the number of levels

The **processPage01** method in Listing 25 begins by getting user input specifying the number of levels or shades that will be allowed for each of the primary colors, red, green, and blue. The code in Listing 25 is straightforward.

Compute the bin size

Limiting the number of allowable shades for each primary color will cause the number of different values contained in the substitution arrays to be less than 256. The values in the substitution arrays will represent a stair-step arrangement of values with several adjacent

elements containing the same value at each level. Listing 26 computes the number of adjacent elements that will contain the same value on the basis of the number of levels.

```
    int binSize = 256/numberLevels;

Listing 26
```

Create the substitution array objects

Listing 27 creates array objects that will be populated with substitution data that is used to populate the lookup table. Note that by default these arrays are populated with all zero values.

```
    short[] red = new short[256];
    short[] green = new short[256];
    short[] blue = new short[256];

Listing 27
```

Populate a master substitution array

Listing 28 implements an algorithm that creates the substitution data in the stair-step fashion described above. This code creates and populates an array object with master data that will be used to populate the specific arrays for the colors that are to be posterized.

```
    short[] masterData = new short[256];
    for(int cnt = 0;cnt < 256;cnt++){
      short value =
               (short)((cnt/binSize)*binSize +
binSize - 1);
      //Clip the values at 0 and 255.
      if(value >'= 256) value = 255;
      if(value < 0) value = 0;//Probably not
possible.
      masterData[cnt] = value;
    }//end for loop

Listing 28
```

Set up the substitution arrays

Listing 29 examines the check boxes in Figure 3 to determine which colors are to be posterized. If any checkbox has been checked, Listing 29 causes the corresponding array reference to point to the array containing the stair-step data created in Listing 28. Otherwise, the array reference will point to an array that contains all zero values by default.

```
    if(page01RedCkBx.getState() == true){
      red = masterData;
    }//end if
    if(page01GreenCkBx.getState() == true){
```

```
      green = masterData;
    }//end if
    if(page01BlueCkBx.getState() == true){
      blue = masterData;
    }//end if

Listing 29
```

Filter the image and return the resulting modified image

Listing 30 calls the method named **processImageForThePage** to filter the image and to return the resulting modified image. This is the same method that was used to process the image for the *Color Inversion* page that I explained earlier *(see Listing 20)*.

```
    return
processImageForThePage(theImage,red,green,blue);

  }//end processPage01

Listing 30
```

Listing 30 also signals the end of the method named **processPage01**, and the end of the discussion of the *Posterizing* page shown in Figure 3.

The Custom Transforms Page

The *Custom Transforms* page is shown in Figure 4. This page makes it possible for the user to select either of two custom substitution arrays to transform the color values from the input image into a different set of color values in the output image. The main purpose of this page is not to provide color transformations that are particularly useful. Rather, the purpose is to illustrate that you can easily implement any custom color substitution algorithm that you can devise that would be useful for your purposes.

Linear Transform

The left panel in Figure 10 shows the result of filtering an image by checking all three checkboxes, selecting the *Linear Transform* radio button, and setting the *Slope* to 0.9 in the *Custom Transforms* page shown in Figure 4.

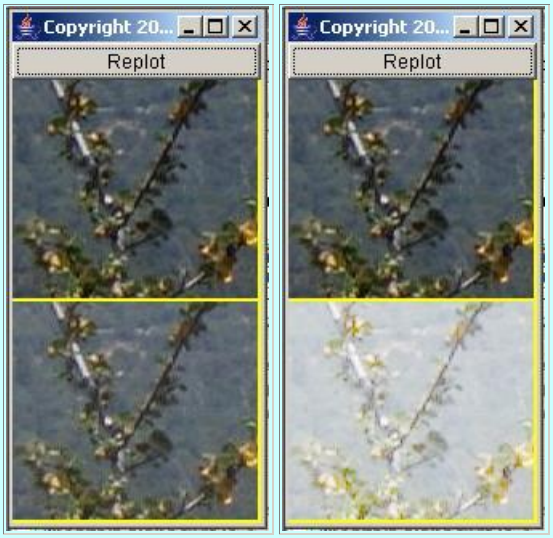

Figure 10

The output is lighter

As you can see, the output image in the left panel is somewhat lighter than the input image. Setting the *Slope* to 1.0 would cause the output image to be an unmodified copy of the input image. Setting the *Slope* to lower values would cause the output image to be progressively lighter, with a value of 0.1 causing the output image to be almost completely white.

Log Transform

The right panel in Figure 10 shows the result of filtering an image by checking all three checkboxes and selecting the *Log Transform* radio button in the *Custom Transforms* page shown in Figure 4.

This option doesn't allow any user input parameters *(such as the Slope in the Linear Transform)*. As you can see, the output image is very light in the bottom-right panel of Figure 10. Thus, the log transform is a rather severe lightening filter.

Similar to decibel conversion
For those who may be interested, this is somewhat analogous to converting a 3D surface to decibels through a log base 10 conversion of the elevation values. See the earlier lesson entitled Plotting 3D Surfaces using Java.

The method named constructPage02

The method named **constructPage02** is shown in its entirety in Listing 31. This method is used to construct the page shown in Figure 4. The method is called from the primary constructor in Listing 10.

```
  void constructPage02(){
    page02.setName("Custom Transforms");//Tab
label.
    page02.setLayout(new BorderLayout());
```

```
    //Create and add the instructional text to the
page.
    String text = "CUSTOM TRANSFORMS\n"
      + "This page illustrates the use of two
different "
      + "custom transforms, one based on the log to
the "
      + "base 10, and the other based on the
equation of "
      + "a straight line.  Both transforms are
designed "
      + "to boost the intensity of pixels with low
color "
      + "values.\n\n"
      + "Specify the color bands that you want to
process "
      + "by checking the checkboxes.\n\n"
      + "Select a radio button to select a
transform.  "
      + "For the linear transform, enter a positive
slope "
      + "that is <= 1.0.  Try a slope of 0.85 for "
      + "example\n\n"
      + "Then click the Replot button.";

    //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.
    Panel page02ControlPanel = new Panel();
    page02ControlPanel.setLayout(new
GridLayout(3,1));

    Panel subControlPanel00 = new Panel();
    subControlPanel00.add(page02RedCkBx);
    subControlPanel00.add(page02GreenCkBx);
    subControlPanel00.add(page02BlueCkBx);
    page02ControlPanel.add(subControlPanel00);

    Panel subControlPanel01 = new Panel();
    subControlPanel01.setLayout(
                             new
FlowLayout(FlowLayout.LEFT));
    subControlPanel01.add(page02LogRa
    page02ControlPanel.add(subControlPanel01);
```

```
    Panel subControlPanel02 = new Panel();
    subControlPanel02.setLayout(
                            new
FlowLayout(FlowLayout.LEFT));
    subControlPanel02.add(page02LinearRadioButton);
    subControlPanel02.add(new Label("  Slope ="));
    subControlPanel02.add(page02TextField);
    subControlPanel02.add(
                    new Label("must be >'= 0.0 and
<= 1.0"));
    page02ControlPanel.add(subControlPanel02);


page02.add(page02ControlPanel,BorderLayout.CENTER);

  }//end constructPage02

Listing 31
```

Tedious but straightforward

Although somewhat tedious, the code in Listing 31 is straightforward. If you use Figure 4 as a guide, you should have no trouble understanding the code in Listing 31.

The method named processPage02

The method named **processPage02** begins in Listing 32. This method processes the image according to the controls shown on the *Custom Transforms* page in Figure 4. This method is called from within a **switch** statement in Listing 14. Note that as with the pages in Figure 2 and Figure 3, this method processes the image using three different substitution arrays, one each for red, green, and blue.

The method named **processPage02** transforms the values in the color bands according to either a *log* transform, or a *linear* transform. The choice of which transform algorithm to use is determined by the radio button selected by the user in Figure 4.

The net effect of both transforms is to emphasize or boost the intensity of colors having low values, thus causing the image to become brighter. However, the two transforms achieve this effect in different ways.

Create substitution array objects

Listing 32 creates array objects that will be populated with data that is later used to populate the lookup table. By default these arrays are populated with all zero values.

```
  BufferedImage processPage02(BufferedImage
theImage){

    short[] red = new short[256];
```

```
    short[] green = new short[256];
    short[] blue = new short[256];

Listing 32
```

Create and populate the master data array object

Listing 33 shows the beginning of the algorithm that will create and populate an array object with master data that will be used to populate the specific arrays for the colors that are selected to be processed.

```
    short[] masterData = new short[256];
    for(int cnt = 0;cnt < 256;cnt++){
      short value = 0;

Listing 33
```

Populate according to linear or log selection

Listing 34 shows the **if** portion of an **if-else** statement that is used to select between a log or linear transform based on the state of the two radio buttons in Figure 4. The code in Listing 34 creates the substitution data values for a log conversion of the color values.

```
      if(page02LogRadioButton.getState() ==
true){
        //Perform a log conversion
        if(cnt == 0){
          //Avoid computing the log of 0.
Substitute the
          // log of 1 instead.  (Note that with
J2SE 5.0,
          // I could have used a static import
directive
          // in order to eliminate the explicit
reference
          // to the Math class in the following
          // expressions.)
          value =

(short)(Math.log10(1.0)*255/Math.log10(255));
        }else{
          value =

(short)(Math.log10(cnt)*255/Math.log10(255));
        }//end else

Listing 34
```

Purpose is not to explain logarithms

Since my purpose here is not to teach you about logarithms, I won't attempt to explain the rationale behind the code in Listing 34. If you already understand logarithms, you should have no trouble understanding the code in Listing 34. If not, just accept the code in Listing 34 as one of many possible transform algorithms that can be used to convert the color values in the input image into a different set of color values in the output image through substitution.

The else clause

Listing 35 shows the **else** portion of the **if-else** statement that began in Listing 34.

```
      }else{//Linear conversion must have been
selected
        //Perform a linear conversion
        double slope = 0;
        try{//Get the slope from the text
field.
          slope =

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

        //Guarantee that the slope is positive
and <= 1.0.
        if((slope < 0.0) || (slope >' 1.0)){
          page02TextField.setText("Bad Input");
          slope = 0.0;//Override user input on
bad input.
        }//end if

        //Compute the intercept of a straight
line with the
        // y-axis using the slope provided by
the user.
        // Cause the line to go through a y-
value of 255
        // at an x-value of 255.
        int yIntercept = (int)(255.0 -
255.0*slope);

        //Compute the value of y for each value
of x(cnt)
        // using the equation of a straight
line, which
        // is, y = slope*x + yIntercept
        value = (short)(slope*cnt +
yIntercept);

        //Guard against roundoff errors that
might cause
```

```
        // the color values to go slightly
outside their
        // allowed range of 0 through 255.
        if(value < 0) value = 0;
        if(value >' 255) value = 255;
      }//end else

      masterData[cnt] = value;
    }//end for loop
```

Listing 35

The code in Listing 35 creates a set of substitution values based on the *slope* and the *y-intercept* using the equation for a straight line.

Purpose is not to explain analytical geometry

Once again, since my purpose here is not to teach you analytical geometry, I won't attempt to explain the rationale behind the code in Listing 35. If you already understand such things as the equation of a straight line, you should have no trouble understanding the code in Listing 35. If not, just accept the code in Listing 35 as another one of many possible transform algorithms that can be used to convert the color values in the input image into a different set of color values in the output image through substitution.

Many different custom transforms are possible

There are many ways to come up with the substitution values for the custom transforms and the purpose of this section of the lesson is simply to illustrate two of them. You may find other ways to develop substitution values that better serve your image-processing needs.

Wrapping it up

The remainder of the **processPage02** method is shown in Listing 36. This code is very similar to the code in the previously-explained methods, and therefore shouldn't require further explanation.

```
    //Examine the check boxes.  If any checkbox
has been
    // checked, reset the corresponding array to
point it
    // to the array containing the master data.
Otherwise,
    // it will contain all zero values by
default.
    if(page02RedCkBx.getState() == true){
      red = masterData;
    }//end if
    if(page02GreenCkBx.getState() == true){
      green = masterData;
    }//end if
```

```
    if(page02BlueCkBx.getState() == true){
      blue = masterData;
    }//end if

    //Process the image and return the processed
result.
    return
processImageForThePage(theImage,red,green,blue);

  }//end processPage02

Listing 36
```

Listing 36 signals the end of the explanation for the program named **ImgMod39**.

Run the Programs

I encourage you to copy the code from Listing 37 and Listing 38 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.

Summary

In this lesson, I provided and explained two different image-processing programs that are compatible with the framework program named **ImgMod05**. The purpose of these two programs is to show you how to write such programs, and also to illustrate a variety of different uses for the **LookupOp** class of the Java 2D API. Three specific uses of the **LookupOp** class were illustrated, and you should be able to devise many more.

Along the way, I also showed you how to construct and use a **JTabbedPane** object as a program GUI. *(If my memory serves me correctly, this is the first time that I have used a **JTabbedPane** in a lesson that I have published.)*

What's Next?

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

- **AffineTransformOp**
- **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

Complete Program Listings

Complete listings of the programs discussed in this lesson are shown in Listing 37 and Listing 38 below.

```
/*File ImgMod38.java
Copyright 2006, R.G.Baldwin

The purpose of this class is to provide a simple example of
an image processing class that is compatible with the use
of the program named ImgMod05, and which illustrates a
single usage of the LookupOp class from the image
processing portion of the Java 2D API.  (Future programs
will illustrate other uses of the LookupOp class.)

A class that is compatible with ImgMod05 is required to
implement the interface named ImgIntfc05. This, in turn,
requires the class to define the method named processImg,
which receives one parameter of type BufferedImage and
returns a reference of type BufferedImage.

The required signature for the processImg method is:

public BufferedImage processImg(BufferedImage input);

The processImg method receives a reference to a
BufferedImage object containing the image that is to be
processed

The processImg method must return a reference to a
BufferedImage object containing the processed image.
```

In this example, the method named processImg is a color inverter method.

The method named processImg as defined in this class receives an incoming reference to an image as a parameter of type BufferedImage. The method returns a reference to an image as type BufferedImage where all of the color values in the pixels have been inverted by subtracting the color values from 255.

The method has been demonstrated to work properly only for the case where the incoming BufferedImage object was constructed for image type BufferedImage.TYPE_INT_RGB. However, it may work properly for other image types as well.

Note that this class does not define a constructor. However, if it did define a constructor, that constructor would not be allowed to receive parameters. This is because the class named ImgMod05 instantiates an object of this class by invoking the newInstance method of the Class class passing the name of this class to the newInstance method as a String parameter. That process does not allow for constructor parameters for the class being instantiated.

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.

The output GUI for the driver program named ImgMod05 contains a Replot button. At the beginning of the run, and each time thereafter that the Replot button is clicked:
-The image processing method belonging to the image processing object is invoked,
-The resulting modified image is displayed along with the original image.

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. However, no such GUI is provided by this program and clicking the Replot button is of no consequence.

The driver program named ImgMod05 reads gif and jpg input files and possibly some other input file types as well. The output file is always a JPEG file.

Usage:

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

```
java ImgMod05 ImgMod38 ImageFileName

The image file must be provided by the user.  However, it
doesn't have to be in the current directory if a path to
the file is included along with the file name on the
command line.

When the 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.

The driver program named ImgMod05 attempts to adjust the
size of the display frame to accommodate both images.  If
the processed image doesn't fit in the display, the user
can manually resize the display frame in order to view both
images.

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

Tested using J2SE5.0 under WinXP.
**********************************************************/

import java.awt.image.*;

class ImgMod38 implements ImgIntfc05{

  //The following method must be defined to implement the
  // ImgIntfc05 interface.
  public BufferedImage processImg(BufferedImage theImage){

    //Use the LookupOp class from the Java 2D API to
    // invert all of the color values in the pixels.  The
    // alpha value is not modified.

    //Create the data for the lookup table.
    short[] lookupData = new short[256];
    for (int cnt = 0; cnt < 256; cnt++){
      lookupData[cnt] = (short)(255-cnt);
    }//end for loop

    //Create the lookup table
    ShortLookupTable lookupTable =
                         new ShortLookupTable(0,lookupData);

    //Create the filter object.
    BufferedImageOp thresholdOp =
                             new LookupOp(lookupTable,null);

    //Apply the filter to the incoming image and return
    // a reference to the resulting BufferedImage object.
    return thresholdOp.filter(theImage, null);
  }//end processImg

}//end class ImgMod38
```

Listing 37

Listing 38

```
/*File ImgMod39.java
Copyright 2006, R.G.Baldwin

The purpose of this class is to illustrate a variety of
different uses for the LookupOp class of the Java 2D API.
In each case, the program uses three data arrays to
process the red, green, and blue color bands individually
or in combination.

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 user 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 ImgMod39 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 three pages.  The tabs on the pages are labeled:

Color Inversion
Posterizing
Custom Transforms

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 labels on the tabs.
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.*;

class ImgMod39 extends Frame implements ImgIntfc05{
  //Primary container used to construct the GUI.
  JTabbedPane tabbedPane = new JTabbedPane();

  //Components used to construct the page in the
  // JTabbedPane that shows Color Inversion on the tab.
  // Components that require local access only are defined
  // locally.  Others are defined here as instance
  // variables.
  Panel page00 = new Panel();
  Checkbox page00RedCkBx = new Checkbox("Red");
  Checkbox page00GreenCkBx = new Checkbox("Green");
  Checkbox page00BlueCkBx = new Checkbox("Blue");

  //Components used to construct the Posterizing page in
  // the JTabbedPane. Components that require local access
  // only are defined locally.  Others are defined here as
  // instance variables.
  Panel page01 = new Panel();
  TextField page01TextField = new TextField("128",6);
  Checkbox page01RedCkBx = new Checkbox("Red");
  Checkbox page01GreenCkBx = new Checkbox("Green");
  Checkbox page01BlueCkBx = new Checkbox("Blue");

  //Components used to construct the Custom Transforms
  // page in the JTabbedPane.  Components that require
  // local access only are defined locally.  Others are
  // defined here as instance variables.
  Panel page02 = new Panel();
  TextField page02TextField = new TextField("1.0",6);
  Checkbox page02RedCkBx = new Checkbox("Red");
  Checkbox page02GreenCkBx = new Checkbox("Green");
  Checkbox page02BlueCkBx = new Checkbox("Blue");
  CheckboxGroup group = new CheckboxGroup();
  Checkbox page02LogRadioButton =
                  new Checkbox("Log Transform",group,true);
  Checkbox page02LinearRadioButton =
              new Checkbox("Linear Transform",group,false);

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

  //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.
  ImgMod39(){//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.

    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 page in the tabbed pane that
  // shows Color Inversion on the tab.  This method is
  // called from the primary constructor.  It illustrates
  // color inversion using three arrays, one for each of
  // the red, green, and blue color bands.
  void constructPage00(){
    page00.setName("Color Inversion");//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 ="COLOR INVERSION USING MULTIPLE ARRAYS\n"
      + "This page illustrates the use of the LookupOp "
      + "filter class for multiple lookup arrays.  By "
      + "checking the appropriate checkboxes below, you "
      + "can cause the program to invert the colors on "
      + "none, or on any combination of color bands.";

    //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,4,1,
                                     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.add(page00RedCkBx);
    page00ControlPanel.add(page00GreenCkBx);
```

```
    page00ControlPanel.add(page00BlueCkBx);
    page00.add(page00ControlPanel,BorderLayout.CENTER);
  }//end constructPage00
  //-----------------------------------------------------//

  //This method processes the image according to the
  // controls located on the page in the tabbed pane that
  // shows Color Inversion on the tab.
  //This method uses the LookupOp image-filtering class to
  // process the image using lookup data from three
  // separate arrays, one each for the red, green, and blue
  // color bands.
  //This method is called from within a switch statement in
  // the method named processImg, which is the primary
  // image-processing method in this program.
  //This method illustrates color inversion.  The effect of
  // color inversion is to produce an output in which the
  // image is similar to the negative of a color
  // photograph.
  BufferedImage processPage00(BufferedImage theImage){
    //Create array objects that will be populated with data
    // that is used later to populate the lookup table.
    short[] red = null;
    short[] green = null;
    short[] blue = null;

    //Create and populate arrays for straight (noInvert)
    // lookup data and inverted lookup data.
    short[] noInvert = new short[256];
    short[] invert = new short[256];
    for(int cnt = 0;cnt < 256;cnt++){
      invert[cnt] = (short)(255 - cnt);//inverted data
      noInvert[cnt] = (short)cnt;//straight lookup data
    }//end for loop

    //Point the three color arrays to the array containing
    // the data that doesn't invert the colors.  This is
    // the default case if no checkboxes are checked.
    red = noInvert;
    green = noInvert;
    blue = noInvert;

    //Examine the check boxes.  If any checkbox has been
    // checked, point the corresponding color array to the
    // array containing the inversion data.
    if(page00RedCkBx.getState() == true){
      red = invert;
    }//end if
    if(page00GreenCkBx.getState() == true){
      green = invert;
    }//end if
    if(page00BlueCkBx.getState() == true){
      blue = invert;
    }//end if

    //Use the LookupOp class from the Java 2D API along
```

```
    // with three separate data arrays to process the
    // color values in the selected color bands.  The
    // alpha value is not modified.
    return processImageForThePage(theImage,red,green,blue);

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

  //This method constructs the page in the tabbed pane that
  // shows Posterizing on the tab.  This method is called
  // from the primary constructor.  It illustrates
  // posterizing.
  //Posterizing is a process of reducing the number of
  // colors in an image to a relatively small number.  The
  // effect is to cause the image to look similar to a
  // painting that was painted using a "paint by numbers"
  // set where the smooth transitions from one color to the
  // next have been eliminated.  This is accomplished by
  // limiting the number of different colors that appear
  // in the image.
  void constructPage01(){
    page01.setName("Posterizing");//Label on the tab.
    page01.setLayout(new BorderLayout());

    //Create and add the instructional text to the page.
    String text = "POSTERIZING\n"
      + "The effect of posterizing is to cause the image "
      + "to look similar to a painting that was painted "
      + "using a \"paint by numbers\" set where the smooth"
      + "transitions from one color to the next have been "
      + "eliminated.  This is accomplished by limiting "
      + "the number of different colors that appear in "
      + "the image.\n\n"
      + "To posterize an image, decide how many levels "
      + "you want to see for each of the three colors, "
      + "red, green, and blue.  Enter that number in the "
      + "text field below.  Select the colors that you "
      + "want to mix by checking the matching check boxes "
      + "and click the Replot button.";

    //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.add(page01TextField);
    page01ControlPanel.add(page01RedCkBx);
    page01ControlPanel.add(page01GreenCkBx);
    page01ControlPanel.add(page01BlueCkBx);
    page01.add(page01ControlPanel,BorderLayout.CENTER);
```

```
  }//end constructPage01
  //-----------------------------------------------------//

  //This method processes the image according to the
  // controls located on the page in the tabbed pane that
  // shows Posterizing on the tab.  This method is called
  // from within a switch statement in the method named
  // processImg.  Note that this method processes the image
  // using three arrays.
  BufferedImage processPage01(BufferedImage theImage){

    int numberLevels = 1;
    try{//Get input value from the text field.
      numberLevels =
                Integer.parseInt(page01TextField.getText());
    }catch(java.lang.NumberFormatException e){
      page01TextField.setText("Bad Input");
      numberLevels = 1;//Override bad user input.
    }//end catch

    //Guarantee that the number of levels falls within the
    // allowable range.  Don't allow divison by 0.
    if((numberLevels <= 0) || (numberLevels > 256)){
      page01TextField.setText("Bad Input");
      numberLevels = 1;//Override bad user input.
    }//end if

    //Compute the number of adjacent elements that will
    // specify the same color value.
    int binSize = 256/numberLevels;

    //Create array objects that will be populated with
    // data that is used to populate the lookup table.
    //Note that by default these arrays are populated with
    // all zero values.
    short[] red = new short[256];
    short[] green = new short[256];
    short[] blue = new short[256];

    //Create and populate an array object with master data
    // that will be used to populate the specific arrays
    // for the colors that are to be processed.
    short[] masterData = new short[256];
    for(int cnt = 0;cnt < 256;cnt++){
      short value =
               (short)((cnt/binSize)*binSize + binSize - 1);
      //Clip the values at 0 and 255.
      if(value >= 256) value = 255;
      if(value < 0) value = 0;//Probably not possible.
      masterData[cnt] = value;
    }//end for loop

    //Examine the check boxes.  If any checkbox has been
    // checked, reset the corresponding array to point it
    // to the array containing the master data.  Otherwise,
```

```
    // it will contain all zero values by default.
    if(page01RedCkBx.getState() == true){
      red = masterData;
    }//end if
    if(page01GreenCkBx.getState() == true){
      green = masterData;
    }//end if
    if(page01BlueCkBx.getState() == true){
      blue = masterData;
    }//end if

    //Process the image and return the result.
    return processImageForThePage(theImage,red,green,blue);

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

  //This method constructs the page in the tabbed pane that
  // shows Custom Transforms on the tab.  This method is
  // called from the primary constructor.  This page
  // illustrates the use of custom transformations of the
  // values in the color bands.
  void constructPage02(){
    page02.setName("Custom Transforms");//Tab label.
    page02.setLayout(new BorderLayout());

    //Create and add the instructional text to the page.
    String text = "CUSTOM TRANSFORMS\n"
      + "This page illustrates the use of two different "
      + "custom transforms, one based on the log to the "
      + "base 10, and the other based on the equation of "
      + "a straight line.  Both transforms are designed "
      + "to boost the intensity of pixels with low color "
      + "values.\n\n"
      + "Specify the color bands that you want to process "
      + "by checking the checkboxes.\n\n"
      + "Select a radio button to select a transform.  "
      + "For the linear transform, enter a positive slope "
      + "that is <= 1.0.  Try a slope of 0.85 for "
      + "example\n\n"
      + "Then click the Replot button.";

    //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.
    Panel page02ControlPanel = new Panel();
    page02ControlPanel.setLayout(new GridLayout(3,1));

    Panel subControlPanel00 = new Panel();
```

```
    subControlPanel00.add(page02RedCkBx);
    subControlPanel00.add(page02GreenCkBx);
    subControlPanel00.add(page02BlueCkBx);
    page02ControlPanel.add(subControlPanel00);

    Panel subControlPanel01 = new Panel();
    subControlPanel01.setLayout(
                          new FlowLayout(FlowLayout.LEFT));
    subControlPanel01.add(page02LogRadioButton);
    page02ControlPanel.add(subControlPanel01);

    Panel subControlPanel02 = new Panel();
    subControlPanel02.setLayout(
                          new FlowLayout(FlowLayout.LEFT));
    subControlPanel02.add(page02LinearRadioButton);
    subControlPanel02.add(new Label("  Slope ="));
    subControlPanel02.add(page02TextField);
    subControlPanel02.add(
                   new Label("must be >= 0.0 and <= 1.0"));
    page02ControlPanel.add(subControlPanel02);

    page02.add(page02ControlPanel,BorderLayout.CENTER);

  }//end constructPage02
  //-----------------------------------------------------//

  //This method processes the image according to the
  // controls located on the page in the tabbed pane that
  // shows Custom Transforms on the tab.  This method is
  // called from within a switch statement in the method
  // named processImg.  Note that this method processes the
  // image using three arrays.  It transforms the values
  // in the color bands according to either a log
  // transform, or a linear transform, with the choice
  // being made by the user through the selection of a
  // radio button.  The net effect of both transforms is to
  // emphasize or boost the intensity of colors having low
  // values, thus causing the image to become brighter.
  // The two transforms achieve this effect in different
  // ways, however.
  BufferedImage processPage02(BufferedImage theImage){

    //Create array objects that will be populated with
    // data that is used to populate the lookup table.
    // Note that by default these arrays are populated with
    // all zero values.
    short[] red = new short[256];
    short[] green = new short[256];
    short[] blue = new short[256];

    //Create and populate an array object with master data
    // that will be used to populate the specific arrays
    // for the colors that are selected to be processed.
    short[] masterData = new short[256];
    for(int cnt = 0;cnt < 256;cnt++){
      short value = 0;
```

```
      //Select between log or linear transformation based
      // on the state of two radio buttons.
      if(page02LogRadioButton.getState() == true){
        //Perform a log conversion
        if(cnt == 0){
          //Avoid computing the log of 0.  Substitute the
          // log of 1 instead.  (Note that with J2SE 5.0,
          // I could have used a static import directive
          // in order to eliminate the explicit reference
          // to the Math class in the following
          // expressions.)
          value =
              (short)(Math.log10(1.0)*255/Math.log10(255));
        }else{
          value =
              (short)(Math.log10(cnt)*255/Math.log10(255));
        }//end else
      }else{//Linear conversion must have been selected
        //Perform a linear conversion
        double slope = 0;
        try{//Get the slope from the text field.
          slope =
             Double.parseDouble(page02TextField.getText());
        }catch(java.lang.NumberFormatException e){
          page02TextField.setText("Bad Input");
          slope = 0.0;//Override user input on bad input.
        }//end catch

        //Guarantee that the slope is positive and <= 1.0.
        if((slope < 0.0) || (slope > 1.0)){
          page02TextField.setText("Bad Input");
          slope = 0.0;//Override user input on bad input.
        }//end if

        //Compute the intercept of a straight line with the
        // y-axis using the slope provided by the user.
        // Cause the line to go through a y-value of 255
        // at an x-value of 255.
        int yIntercept = (int)(255.0 - 255.0*slope);

        //Compute the value of y for each value of x(cnt)
        // using the equation of a straight line, which
        // is, y = slope*x + yIntercept
        value = (short)(slope*cnt + yIntercept);

        //Guard against roundoff errors that might cause
        // the color values to go slightly outside their
        // allowed range of 0 through 255.
        if(value < 0) value = 0;
        if(value > 255) value = 255;
      }//end else

      masterData[cnt] = value;
    }//end for loop

    //Examine the check boxes.  If any checkbox has been
```

```
    // checked, reset the corresponding array to point it
    // to the array containing the master data.  Otherwise,
    // it will contain all zero values by default.
    if(page02RedCkBx.getState() == true){
      red = masterData;
    }//end if
    if(page02GreenCkBx.getState() == true){
      green = masterData;
    }//end if
    if(page02BlueCkBx.getState() == true){
      blue = masterData;
    }//end if

    //Process the image and return the processed result.
    return processImageForThePage(theImage,red,green,blue);

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

  //Use the LookupOp class from the Java 2D API along
  // with three separate data arrays to process the
  // color values in the selected color bands.  The
  // alpha value is not modified.  This is a common method
  // that is called by the code that processes each
  // individual page in the tabbed pane.
  BufferedImage processImageForThePage(
                                  BufferedImage theImage,
                                  short[] red,
                                  short[] green,
                                  short[] blue){
    //Create and populate a 2D array with data for the
    // lookup table.  Note that this is a 2D array, rather
    // than a 1D array, which is the case when a single
    // data array is used to process all three color bands.
    short[][] lookupData = new short[][]{red,green,blue};

    //Create the lookup table.  The first parameter is an
    // offset for extracting data from the array object.
    //In this case, all of the data is extracted from the
    // array object beginning at an index of 0.
    ShortLookupTable lookupTable =
                        new ShortLookupTable(0,lookupData);

    //Create the filter object. The second parameter
    // provides the opportunity to use RenderingHints.
    BufferedImageOp filterObject =
                            new LookupOp(lookupTable,null);

    //For illustration purposes only, work backwards from
    // the filterObject to get and display some data
    // from the lookup table.  Note that this is not an
    // image-processing requirement.
    displayTableData(filterObject);

    //Apply the filter to the incoming image and return
    // a reference to the resulting BufferedImage object.
```

```
    // The second parameter can optionally specify an
    // existing BufferedImage object to serve as a
    // destination for the processed image.
    return filterObject.filter(theImage, null);
  }//end processImageForThePage
  //-----------------------------------------------------//

  //Print some column headers followed by data from the
  // lookup table.  Print every 32nd row beginning with
  // row 0.  Then print the data for row 255.
  void displayTableData(BufferedImageOp filterObject){

    //First, get a 2D array containing data from the lookup
    // table.
    ShortLookupTable theTable = ((ShortLookupTable)(
                        (LookupOp)filterObject).getTable());
    short[][] tableData = theTable.getTable();

    System.out.println("Row\tRed\tGreen\tBlue");
    for(int row = 0;
        row<tableData[0].length;
        row += 32){
        System.out.print((row) + "\t");
      for(int col = 0;
          col<tableData.length;
          col++){
        System.out.print(tableData[col][row] + "\t");
      }//end inner loop
      System.out.println();
    }//end outer loop
    System.out.println(255 + "\t" + tableData[0][255]
                       + "\t" + tableData[1][255] + "\t"
                       + tableData[2][255]);

  }//end displayTableData

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

  //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;
    }//end switch
```

```
    return outputImage;
  }//end processImg

}//end class ImgMod39
```

Listing 38

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 LookupOp

-end-