# A Framework for Experimenting with Java 2D Image-Processing Filters

*Learn about the image-filtering classes of the Java 2D API. Also learn 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.*

**Published:** November 28, 2006
**By Richard G. Baldwin**

Java Programming, Notes # 450



---

# Preface

**An image-processing framework**

In the lesson entitled Processing Image Pixels using Java, Getting Started, I provided a framework program named **ImgMod02** that makes it easy to:

- Modify the pixels in an image
- Display the modified image
- Compare the modified image with the original image

I upgraded that framework a couple of times in subsequent lessons. *(See References below.)*

That framework and the updated framework named **ImgMod02a** that I provided in a subsequent lesson were based on the assumption that you would get right down in the mud and learn how to implement pixel-modification algorithms by working directly with the pixels.

## Sometimes there is an easier way

In some cases, writing your own pixel-modification or image-filtering program is the best and most effective way to go. However, in some cases, there is an easier way. As of the date of this writing, the Java 2D API provides the following classes that can be used to filter an image and modify the pixels in that image:

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

For those cases where it is possible to use one of the existing classes to satisfy your image-filtering needs, it is usually easier to write a program using one or more of the above classes than it is to write your own image-filtering program.

In this lesson, I will provide a framework program named **ImgMod05** that makes it easy to use the above classes to modify the pixels in an image and to display the modified image.

## The Java 2D API doesn't always provide an easier way

Before going any further, however, I need to point out that despite the fact that the image-filtering classes in the above list from the Java 2D API provide a great deal of capability, there are many interesting image-filtering needs that can't be satisfied by the capabilities of those classes.

> *(See, for example, the section entitled **Highlighting an image** in the earlier lesson entitled Processing Image Pixels using Java, Getting Started. Although I haven't given it a great deal of thought, I don't believe that any of the classes in the above list could be used to create the spotlight effect shown in that lesson. I will also explain later that the **ConvolveOp** class is very restrictive, and that you may find that you can do a better image-processing job by writing your own classes for this purpose than by using the **ConvolveOp** class.)*

However, for those cases where the classes from the Java 2D API will satisfy your needs, you should probably use those classes rather than to invent your own classes.

## Not a lesson on JAI

If you arrived at this lesson while searching for instructions on how to use the Java Advanced Imaging *(JAI)* API, you are certainly welcome to be here. However, that is not yet the purpose of the lessons in this series. *(Maybe I will add lessons on the JAI API later.)*

The JAI API provides image-processing capabilities that this series won't address for awhile. The next several lessons in this series will be limited to either the capabilities of the Java 2D API, or the capabilities of image-filtering programs that you write yourself.

## A new framework for the image-filtering classes of the Java 2D API

In this lesson, I will provide and explain a new framework program that makes it easy to:

- Use the image-filtering classes of the Java 2D API to write and evaluate image-filtering programs.
- Display the modified image along with the original image for easy comparison in a *before* and *after* sense.

## A driver program

This program is designed to be used as a driver that controls the execution of another program that actually processes the pixels.

By using this program as a driver, you can concentrate on writing and executing image-filtering algorithms without having to worry about many of the details involving image files, image display, etc.

## A simple image-filtering program

Also in this lesson, I will provide and explain the first of several image-filtering programs designed to teach you how to modify an image by using the image-filtering classes of the Java 2D API.

The image-filtering program provided in this lesson will be relatively simple with the intent being to get you started but not necessarily to produce a modified image that is especially interesting.

## More interesting image-filtering programs

Figure 1 through Figure 19 show examples of the use of the image-filtering classes from the earlier list without showing the code used to produce the images.

Future lessons will show you how to write image-filtering programs that use the image-filtering classes that were presented in the earlier list.

## 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 figures and listings while you are reading about them.

### A disclaimer

The programs that I will provide and explain in this series of lessons are not intended to be used for high-volume production work. Numerous integrated image-processing programs are available for that purpose.

However, most of the programs will have the capability to read an input image file, modify the image, and write the modified image into an output JPEG file. Therefore, feel free to use the programs to process your images if you find them useful.

# Image-Filtering Examples

This section provides examples of the input and output for images that were modified using the image-filtering classes of the Java 2D API and the **ImgMod05** framework program. This lesson will provide and explain the code for the framework program. The specific code used to produce these images under control of the **ImgMod05** framework will be provided and explained in future lessons.

## LookupOp Examples

### Color inversion

Figure 1 shows the result of using the **LookupOp** class to invert the color values of all the blue pixels in an image without modifying the red and green pixels.

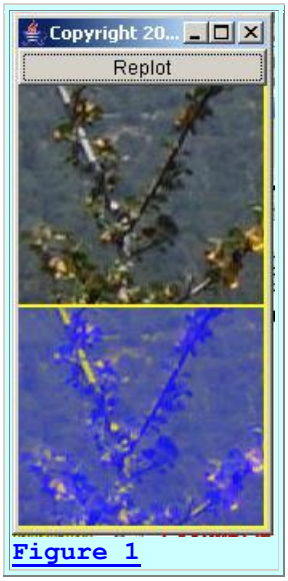


**Figure 1**

As is always the case with the framework program named **ImgMod05**, the input image is shown at the top and the output image is shown at the bottom.

## An independent look

Figure 2 shows an independent look at the output JPEG file produced by the program named **ImgMod05** for the case shown in Figure 1.

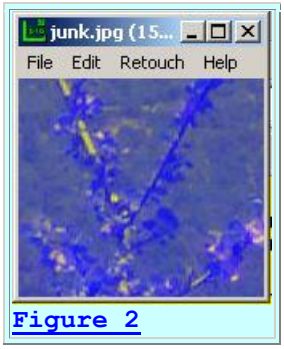

Figure 2

The image shown in Figure 2 was displayed using the commercial image-processing program named LView Pro.

## Invert all colors

Figure 3 shows the result of using the **LookupOp** class to invert the color values of all the red, green, and blue pixels in the same image.

Figure 3

## A few words about image color inversion

I will show you more than one way to use the classes in the above list to invert the colors in an image. Basically the process involves creating a new pixel color value by subtracting the old color value from 255.

Image inversion is an important concept for the following reasons:

- Every color value in the inverted image is guaranteed to be different from the original color value *(although values in the range of 127 and 128 aren't very different).*
- Every color value in the inverted image is guaranteed to be within the limits of 0 to 255 inclusive.
- The original color value can be easily recovered from the modified color value simply by subtracting the modified color value from 255.

For those reasons, many programs, *(such as the WYSIWYG HTML editor that I am using to write this lesson)*, use inverted image colors to represent the colors in an image that has been *selected* for editing. When the image is *deselected*, it is easy for the program to restore the original colors in the visual representation of the image.

## Posterizing an image

Figure 4 shows the result of using the **LookupOp** class to *posterize* the same image shown earlier.

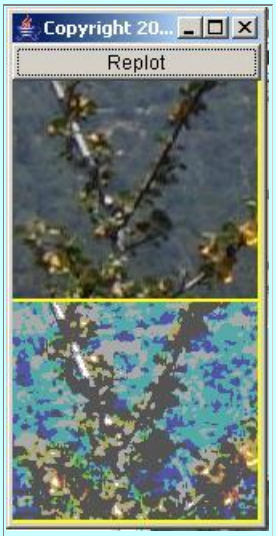

Figure 4

## What does it mean to posterize an image?

The best explanation that I can give for the process of *posterizing* an image is that it reduces the number of actual colors used to represent the image. This tends to make it look like it was painted using a *"paint by numbers"* set containing a limited number of paint colors.

*(If I did my calculations correctly, the bottom image in Figure 4 has twenty-seven actual colors, made up of three shades each of red, green, and blue.)*

## Custom transforms

Figure 5 shows the result of using the **LookupOp** class to modify the distribution of the red, green, and blue pixel colors in such a way as to cause the distribution to become somewhat narrower and to cause the mean value to be moved closer to 255.

Figure 5

You learned quite a lot about the impact of modifying the pixel color distributions in the earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness.

## Many custom transforms are possible

You can use the **LookupOp** class to implement a wide variety of custom transforms that convert the incoming image pixel colors to a different, but well-defined set of output pixel colors. In a future lesson, I will show you how to implement a *Log Transform* and a *Linear Transform*. *(For future reference, the bottom image in Figure 5 was produced using the Linear Transform with a Slope value of 0.85.)*

## Assessment

It would not be difficult to write your own image-filtering program to replicate the behavior of the **LookupOp** class. However, if the **LookupOp** class will satisfy your needs, it will require more work for you to write your own class than to simply use the existing class.

Therefore, in the spirit of the *"reuse, don't reinvent,"* principle of OOP, if the **LookupOp** class will meet your needs, by all means use it and don't reinvent your own class to perform the same task.

## AffineTransformOp Examples

In my opinion, *(with the possible exception of the **ColorConvertOp** class)*, the **AffineTransformOp** class is the most significant of all the image-filtering classes in the Java 2D API. I will explain my reasons for this opinion in the Assessment section. First, however, I will show you some examples of what you can accomplish using this class.

### Interpolation choices

Whenever you change the size, the location, or the orientation of an image, you usually need to recreate the color values for all of the new pixels on the basis of the color values contained in the original image. This is not a trivial computational task.

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

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

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

I will identify the choices that were used for each of the following examples.

### Scaling

Figure 6 and Figure 7 show the results of a simple scaling transform. For this transform, the width of the image was increased by a factor of 1.5 and the height was increased by a factor of 2.0. The *nearest neighbor* scheme was used to create the new pixel values for Figure 6. *Bicubic* interpolation was used for Figure 7.


Figure 6

**The image quality in Figure 7 is better**

If you compare the bottom image in Figure 6 above with the bottom image in Figure 7 below, you should be able to see that the image quality in Figure 7 is superior to that in Figure 6.

Figure 7

## Neither image has outstanding quality

Although neither image shows outstanding quality, *(which is a common result of enlarging images of this type)*, the image produced using the *nearest neighbor* scheme in Figure 6 is more grainy than the image produced using *bicubic* interpolation in Figure 7.

> *(For example, note the vertical stripes in the light gray stem in the upper-left of the image in Figure 6. Although there is some striping in this area of Figure 7, it isn't nearly as pronounced.)*

## Will use *bicubic* interpolation for remaining examples

Now that we have that issue out of the way, the remaining **AffineTransformOp** examples will use *bicubic* interpolation.

## Translation

Figure 8 shows the result of translating the image by 15.25 pixels to the right and 20.75 pixels down.


Figure 8

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

## Rotation

Figure 9 shows the result of first translating the image to the right and down a distance sufficient to give it room to rotate it about its center, and then rotating it by 31.5 degrees in a clockwise direction about its center.


Figure 9

### Mirror image

Figure 10 shows the result of using the **AffineTransformOp** class to effectively flip the image about a vertical line in the center of the image, thus creating a mirror image of the original image.

Figure 10

### Assessment

You can combine these basic transforms to produce a wide variety of other results.

As I stated earlier, in my opinion, the **AffineTransformOp** class is probably the most significant of all the image-filtering classes in the Java 2D API. Because of the interpolation issue, a great deal of programming effort would be required for you to write your own class that duplicates the behavior of the **AffineTransformOp** class. Therefore, if this class will serve your needs, this is clearly a case where you should *use* the existing class instead of *inventing* a new class.

## BandCombineOp Examples

### BufferedImage objects versus Raster objects

Unlike some of the other image-filtering classes in the Java 2D API that can operate either on **BufferedImage** objects or on **Raster** objects, the **BandCombineOp** filter can operate only on **Raster** objects.

> *(I will have more to say about **BufferedImage** objects and **Raster** objects in future lessons. For now, suffice it to say that by converting the **BufferedImage** objects to **Raster** objects, it is possible to operate on smaller rectangular areas of the image that are extracted from within the body of the entire image.)*

### Image-filtering methodology

For the **BandCombineOp** class, the red, green, and blue values of each pixel are treated as a column matrix. A 1 is appended onto the end of each column matrix producing a set of four-element column matrices that represents all of the pixels in the input **Raster** object.

Each pixel in the output **Raster** is produced by multiplying a user-specified 3x4 processing matrix by the 4x1 column matrix that represents the corresponding pixel in the input **Raster**. The same 3x4 processing matrix is applied to every input pixel.

This makes it possible to cause the intensity or shade of each of the three colors *(red, green, and blue)* in each pixel of the output **Raster** to be a function of the combined intensities of all three colors of the corresponding pixel in the input **Raster**, *(plus a constant that is equal to the rightmost value in the corresponding row of the processing matrix).*

## Potential arithmetic overflow

It is unclear in the documentation what happens to the output color value if the value resulting from the matrix multiplication and the addition of the constant falls outside the range from 0 to 255. However, observation of the results indicates that rather than clipping the value to force it to be within the range from 0 to 255, the value is allowed to overflow and become corrupt. Therefore, care must be exercised to avoid such overflow when setting the multiplicative values in the processing matrix.

> *(On the other hand, allowing the overflow to occur can lead to some interesting visual effects.)*

## A variety of interesting effects

This processing approach can lead to a variety of interesting effects. One author says that this class can be used to create cubist-style images. *(Given my limited knowledge of art, I will simply have to take the author's word on this.)*

## Color inversion

As with two of the other image-filtering classes, this class can easily be used to invert the colors, producing an output just like the output shown in Figure 3. *(Since I have already shown you an inverted image, I will simply refer back to Figure 3 in this section rather than repeating it here.)*

## Conversion to gray

Figure 11 shows the result of causing the red, green, and blue color values of each output pixel to be the average of all three color values of the corresponding input pixel.

Figure 11

Figure 11 also shows the result of extracting and processing a **Raster** from the original image that is 130 pixels wide and 120 pixels tall with the upper-left corner of the **Raster** being located at a horizontal position of 10 pixels and a vertical position of 15 pixels in the original image.

## Arithmetic overflow

Figure 12 shows the same thing as Figure 11, except that each output color value in Figure 12 was multiplied by 1.515 relative to the values in Figure 11.

Figure 12

As you can see, this caused the overall output image in Figure 12 to be brighter than the output image in Figure 11.

As you can also see, this resulted in arithmetic overflow for those output color values that exceeded a value of 255. As a result, light gray areas in Figure 11 became black areas outlined with white in Figure 12.

### A cubist-style image

For whatever it is worth, if I interpreted the previously-referenced article correctly, the output shown in Figure 13 is a *cubist-style* image. At least, it was produced using the same processing matrix as the one given in that article.

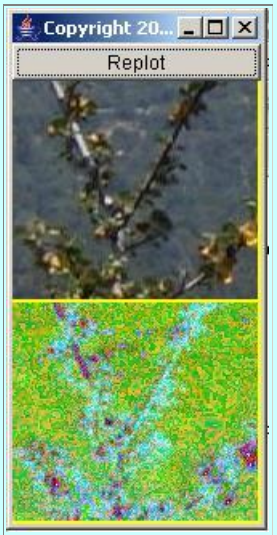

Figure 13

However, as I stated earlier, given my limited knowledge of art, I will simply have to take the author's word that this is a *cubist-style* image. Whatever it is, it illustrates that it is possible to use the **BandCombineOp** class to produce some weird and interesting effects.

## ConvolveOp Examples

In my opinion, the **ConvolveOp** class is the weakest of the image-filtering classes in the Java 2D API. I will explain my reasons for this opinion later in the section entitled Assessment. First, however, I will show you some examples of image convolution using the capabilities of the **ConvolveOp** class.

### Edge treatment

When performing image convolution, you must always decide how you are going to treat the edges of the image. The **ConvolveOp** class provides the following choices:

- Copy edge pixels in unmodified form
- Zero fill the edge pixels

Figure 14 and Figure 15 show the results of electing each of those two choices.

> *(The processed portion of each image was purposely blacked out or set to white to emphasize the treatment of the edges.)*

## Copy edge pixels in unmodified form

In Figure 14, the pixels at the edges of the input image were simply copied to the edges of the output image without modification. This is evidenced by the fact that the edges of the output image look just like the edges of the input image.

Figure 14

## Zero fill the edge pixels

In Figure 15, the pixels at the edge of the output image were set to zero, producing the black border around the output image.

Figure 15

The remaining examples in this section will use the first alternative and simply copy the pixels from the edges of the input image to the edges of the output image in unmodified form. I will

explain what controls the width of the border produced by that process in a future lesson on the **ConvolveOp** class.

## A flat 4x4 smoothing filter

Figure 16 shows the result of applying a flat 4x4 smoothing filter to the same image of a starfish that was used in the earlier lesson entitled Processing Image Pixels, Applying Image Convolution in Java.

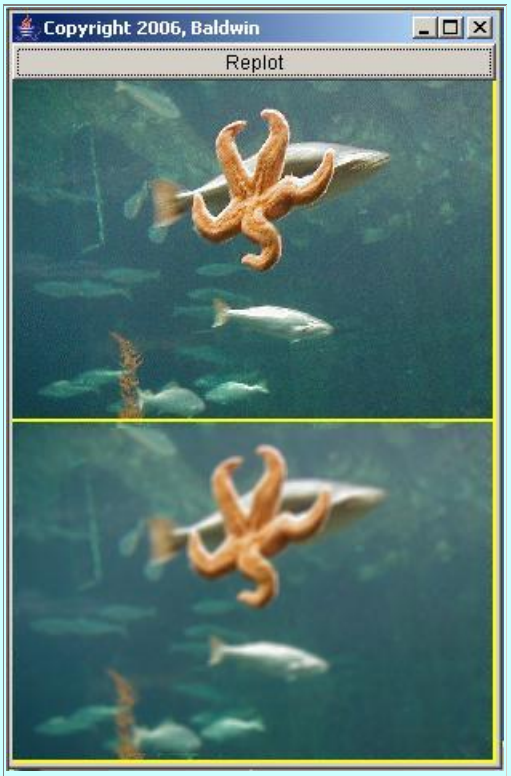


Figure 16

If you compare the output image in Figure 16 with the corresponding figure in the earlier lesson, you will see good agreement. This is because none of the output color values fell outside the range from 0 to 255 inclusive, and it was not necessary to deal with the normalization issue that I will discuss later in the Assessment section.

## An embossing filter

The earlier lesson presented the results of a convolution filtering process that was intended to produce an output that looks like embossed stationary. Figure 17 shows the same process implemented using the **ConvolveOp** class.

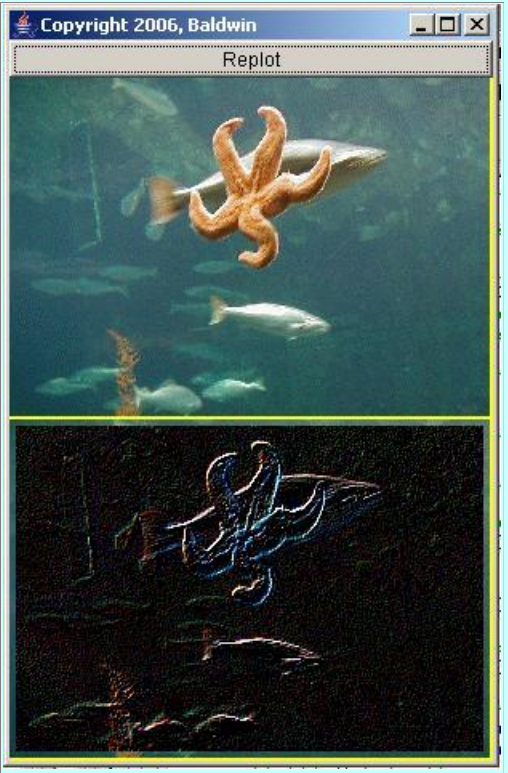

Figure 17

## The differences are striking

The difference between the results shown in Figure 17 and the results in the earlier lesson are striking. The difference lies solely in the *normalization* scheme used to deal with convolution output values that fall outside the allowable range from 0 through 255 inclusive.

## One approach is to simply clip the values

Apparently the **ConvolveOp** class simply clips those values at 0 and 255. Thus, this is not a *safe* scheme because it throws away some of the output values replacing them by either 0 or 255.

## A statistical scheme

The scheme used in the earlier lesson, on the other hand doesn't throw away any of the output values. Rather, the distribution of the output values is compressed, while maintaining its general shape, so as to cause all of the output values to fall within the allowable range. Thus, this is a *safe* scheme in that it doesn't throw away any information.

> *(When we do DSP in the world of signals, we usually try to avoid clipping the data. Clipping is a nonlinear, non-reversible process, which is usually considered to be a bad idea.)*

## Another statistical scheme

Another image in the earlier lesson shows the results for the same input image and the same convolution filter with still another normalization scheme. This scheme causes the mean and the standard deviation of the output to match the mean and the standard deviation of the input. As a result, some of the output values may still fall outside the acceptable limits. In that case, those values are simply clipped at 0 and 255, making this scheme less *safe* than the one described above, but probably more safe than simply clipping the output values at 0 and 255.

## Assessment

Earlier I indicated that in my opinion, the **ConvolveOp** class is the weakest of the image-filtering classes in the Java 2D API. Now I will explain my reasons for that opinion.

## Output normalization is the real issue

In the earlier lesson entitled Processing Image Pixels, Applying Image Convolution in Java, I explained that image convolution results can, and frequently do result in color values that fall outside the range from 0 through 255 inclusive. Thus, the real issue in image convolution is not how to do the convolution arithmetic. The arithmetic algorithm for image convolution is almost trivial. The real issue for the serious image processor has to do with how you normalize the output values to:

- Force them into the range from 0 through 255,
- Discard those that are outside the range from 0 through 255, or
- A combination of the two

## Apparently the ConvolveOp class simply clips the output

Although the documentation for the **ConvolveOp** class doesn't indicate how the results are normalized, observation of the results suggests that values outside the acceptable range are simply clipped to values of 0 and 255.

## Not necessarily the best approach

While that is the easiest approach to implement, it probably isn't the best approach from an image processing viewpoint. In fact, there probably isn't any one best approach. The normalization scheme that works best for one situation is likely to give way to a different normalization scheme for another situation. Unfortunately, simply clipping the output at 0 and 255 isn't likely to be the best approach for very many situations.

## Two normalization schemes were described earlier

In the earlier lesson entitled Processing Image Pixels, Applying Image Convolution in Java, I described two different normalization schemes that are based on the statistical distribution of the input and output color values. I showed that the choice of normalization scheme can have a dramatic effect on the visual results.

### Would like optional normalization schemes in the ConvolveOp class

I would like to have seen a choice among several such normalization schemes provided by the **ConvolveOp** class instead of simply clipping the results to values of 0 and 255. In my opinion, simply clipping the results seriously limits the value of the **ConvolveOp** class for the serious image processor.

## RescaleOp Examples

The **RescaleOp** class can be used to multiply the color value for each pixel by a user-specified scale factor, and then to add a user-specified constant to the product. Separate scale factors and additive constants are provided for each of the red, green, and blue colors.

Color values that fall outside the allowable range from 0 to 255 are simply clipped to 0 and 255.

### Color inversion

This is another class that makes it easy to invert the colors. In this case, the scale factor for all three bands would be set to -1 and the additive constant would be set to 255. This would produce an output image like that shown in Figure 3. Similarly, each color band can be inverted separately producing results like those shown in Figure 1.

### Adjusting the contrast and brightness

The **RescaleOp** class can also be used to adjust the contrast and brightness of an image using the concepts that I explained in the earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness.

Figure 18 shows the result of using the **RescaleOp** class to improve the contrast and brightness of the input image. In this case, each color value was multiplied by 3 and then a value of -160 was added to each product.

Figure 18

## The statistical changes

Multiplying the color values by a scale factor widens the distribution as shown in the earlier lesson. This increases the contrast. Adding the constant adjusts the mean value, thus modifying the brightness.

*(See the histogram in the earlier lesson.)*

Note that although the methodology isn't exactly the same, the results in Figure 18 compare favorably with the first figure in the earlier lesson indicating that the standard deviation and the mean for the output image in Figure 18 is probably very similar to the standard deviation and the mean for the output image shown in the first figure in the earlier lesson.

## Assessment

As demonstrated in the earlier lesson, it is not difficult to write your own program to replicate the behavior of the **RescaleOp** class. However, if the **RescaleOp** class will serve your needs, use it, don't reinvent it.

### One major difference

There is, however, one aspect of my implementation in the earlier lesson that I consider to be better than the implementation of the **RescaleOp** class, particularly when used for the purpose of adjusting the contrast and brightness of an image.

The mean value modification in my implementation is specified by the user as a multiplier, such as 1.25. This would, for example, cause the new mean value to be 1.25 times greater than the old mean value.

With the **RescaleOp** class, a constant must be added or subtracted from the product in order to move the mean value. You usually won't know what the actual mean value is, so you will have to do a lot of guesswork in order to determine the proper additive value.

On the other hand, the **RescaleOp** class can be used for other purposes *(such as color inversion)* where an additive constant is more appropriate than a multiplicative factor so it is a more general implementation.

## ColorConvertOp Example

The apparent purpose of this class is to make it possible for you to convert an image from one **ColorSpace** to another **ColorSpace**.

> *(I will leave it up to you to go to the Sun documentation to learn about the **ColorSpace** class.)*

In any event, when deciding upon a new color space, several are available. They are defined as constants in the **ColorSpace** class.

### Conversion to grayscale

Figure 19 coverts the color space of the input image to type **ColorSpace.CS_GRAY**. As you can see, this changed the image from a color image to what would probably be called a grayscale image.

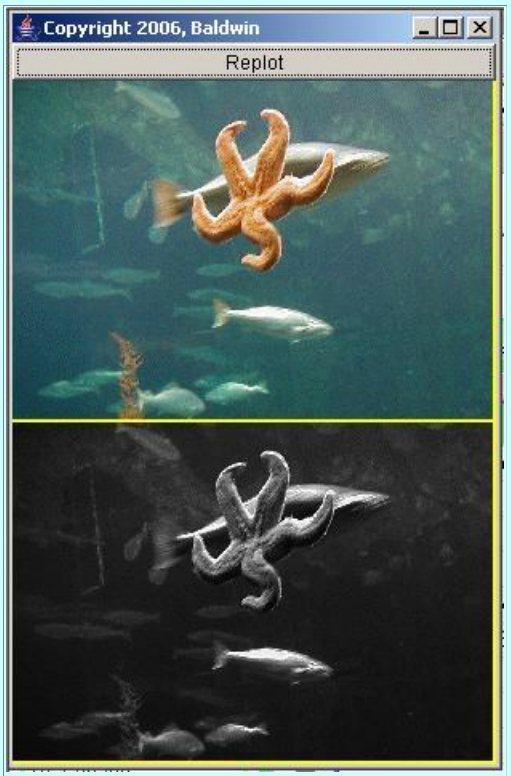

Figure 19

**Assessment**

In my opinion, writing your own program to replicate the behavior of the **ColorConvertOp** class would be very difficult. If you need this capability, by all means, use the class and don't attempt to reinvent it.

**A caveat**

This is the one case that I have found where programs that use the image-filtering classes of the Java 2D API are incompatible with the framework program named **ImgMod05**. If you modify the color space of an image, the code in **ImgMod05** that attempts to write the output image into a JPEG file will throw an error.

# Background Information

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 these 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.

### 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.

In addition, I recommend that you study the lessons that I have listed in the References section of this document.

## Preview

In this lesson, I will present and explain the three programs and one interface:

- ImgMod05a
- ImgMod05
- ProgramTest
- ImgIntfc05

### ImgMod05a

This is just about the simplest program that I know how to write that shows how to read, modify, and write an image file using Java2D image-filtering classes. This program isn't intended to be particularly useful in its own right. Rather, it is intended to teach you how to accomplish the steps described above so that you will understand those steps when you attack the larger program named **ImgMod05**.

### ImgMod05

This is the main program for which this lesson was written. This is the new image-processing framework program that I described in earlier sections of this lesson.

### ProgramTest

This is a test program designed to test the program named **ImgMod05** and also intended to show the essentials of writing an image-processing program that can be run under control of the **ImgMod05** framework.

### ImgIntfc05

This is an interface that must be implemented by any program that is designed to run under control of the **ImgMod05** framework.

## Discussion and Sample Programs

## The program named *ImgMod05a*

I will begin by explaining the program named **ImgMod05a**. I will explain the program in fragments. You can view a complete listing of the program in Listing 29 near the end of the lesson.

As mentioned earlier, the purpose of this program is to show you, in the simplest program practical, how to read, modify, and write an image file using Java2D image-filtering classes.

The program was tested using J2SE 5.0 under WinXP.

### The ImgMod05a class

The beginning of the class is shown in Listing 1.

```
class ImgMod05a{
  BufferedImage rawBufferedImage;
  BufferedImage processedImage;
  static String theImgFile =
"ImgMod05Test.jpg";
  MediaTracker tracker;

Listing 1
```

Listing 1 declares four instance variables, two of which are type **BufferedImage**. The **BufferedImage** type is relatively new to this lesson, and is crucial to the use of the image-filtering classes of the Java 2D API.

The types of the other two instance variables have been used in numerous previous lessons.

### The BufferedImage class

The **BufferedImage** class belongs to the **java.awt.image** package. Figure 20 contains part of what Sun has to say about this class.

> The **BufferedImage** subclass describes an **Image** with an accessible buffer of image data.
>
> A **BufferedImage** is comprised of a **ColorModel** and a **Raster** of image data.
>
> The number and types of bands in the **SampleModel** of the **Raster** must match the number and types required by the **ColorModel** to represent its color and alpha components.

All **BufferedImage** objects have an upper left corner coordinate of (0, 0).

Any **Raster** used to construct a **BufferedImage** must therefore have minX=0 and minY=0.

**Figure 20**

## Objects of the BufferedImage class are required

As you will see later, the framework program named **ImgMod05**:

- Reads and displays image data from an image file.
- Creates a **BufferedImage** object from the contents of the image file.
- Passes the **BufferedImage** object to the image-processing program for image processing.
- Receives a modified **BufferedImage** object back from the image-processing program and displays the modified image.
- Repeats the cycle each time the user clicks a **Replot** button.

## One class cannot operate directly on a BufferedImage object

All but one of the image-filtering classes in the Java 2D API can operate directly on **BufferedImage** objects, or on **Raster** objects. The one exception is the **BandCombineOp** image-filtering class, which can operate only on **Raster** objects.

## The main method

The **main** method for this class is shown in its entirety in Listing 2.

```
  public static void main(String[] args){
    //Instantiate an object of this class.
    ImgMod05a obj = new ImgMod05a();
  }//end main
```

**Listing 2**

The code in Listing 2 is straightforward and shouldn't require further explanation.

## The constructor for ImgMod05a

The constructor for the class is shown in its entirety in Listing 3.

```
  public ImgMod05a(){//constructor
    //Get an image from the specified image
file.
    rawBufferedImage = getTheImage();
```

```
    //Process the image.
    processedImage =
processImg(rawBufferedImage);

    //Write the modified image into a JPEG file
named
    // junk.jpg.
    writeJpegFile(processedImage);

  }//end ImgMod05a constructor

Listing 3
```

The code in Listing 3 is also straightforward since all it does is invoke three methods in succession. The purposes of the three methods are:

- **getTheImage** - Get a BufferedImage object that represents the image.
- **processImg** - Process the image.
- **writeJpegFile** - Write the modified image into an output JPEG file named **junk.jpg**.

All the real work is done by the three methods listed above.

## The method named getTheImage

The method named **getTheImage** begins in Listing 4.

```
  //This method reads an image from a specified
image file,
  // writes it into a BufferedImage object, and
returns a
  // reference to the BufferedImage object.
  //The name of the image file is contained in
an instance
  // variable of type String named theImgFile.
  BufferedImage getTheImage(){
    Image rawImage =
Toolkit.getDefaultToolkit().

getImage(theImgFile);

    //Use a MediaTracker object to block until
the image is
    // loaded or ten seconds has elapsed.
Terminate and
    // display an error message if ten seconds
elapse
    // without the image having been loaded.
Note that the
    // constructor for the MediaTracker
requires the
    // specification of a Component "on which
the images
```

```
    // will eventually be drawn" even if there
is no
    // intention for the program to actually
display the
    // image.  It is useful to have a media
tracker with a
    // timeout even if the image won't be drawn
by the
    // program.  Also, the media tracker is
needed to delay
    // execution until the image is fully
loaded.
    tracker = new MediaTracker(new Frame());
    tracker.addImage(rawImage,1);

    try{
      if(!tracker.waitForID(1,10000)){
        System.out.println("Timeout or Load
error.");
        System.exit(1);
      }//end if
    }catch(InterruptedException e){
      e.printStackTrace();
      System.exit(1);
    }//end catch

    //Make certain that the file was
successfully loaded.
    if((tracker.statusAll(false)
                          &
MediaTracker.ERRORED
                          &
MediaTracker.ABORTED) != 0){
      System.out.println("Load errored or
aborted");
      System.exit(1);
    }//end if

Listing 4
```

## Nothing new here

I have used and explained code similar to that shown in Listing 4 in numerous previous lessons so there is nothing new here. If you are unfamiliar with that code, just go to Google and search for the keywords

```
Baldwin MediaTracker.
```

## Create a BufferedImage object

The code in Listing 5 begins by creating an empty **BufferedImage** object using one of three overloaded constructors that exist for the **BufferedImage** class as of the date of this writing. This code is new to this lesson.

```
    BufferedImage buffImage = new
BufferedImage(

rawImage.getWidth(null),

rawImage.getHeight(null),

BufferedImage.TYPE_INT_RGB);

Listing 5
```

Note that the specified image type in Listing 5 is critical to the correct operation of the method named **processImg**, which will be invoked later. *(The **processImg** method may work correctly for other image types, but has been tested only for **TYPE_INT_RGB**.)*

The parameters to the **getWidth** and **getHeight** methods in Listing 5 are references to an **ImageObserver** object, or references to *"an object waiting for the image to be loaded."*

## Draw the image data into the BufferedImage object

The code in Listing 6 draws the image data from the input file into the **BufferedImage** object and returns a reference to that object.

```
    // Draw Image into BufferedImage
    Graphics g = buffImage.getGraphics();
    g.drawImage(rawImage, 0, 0, null);

    return buffImage;
  }//end getTheImage

Listing 6
```

You should have no problem following the code in Listing 6 if you simply look up the various methods in the Sun documentation.

## Process the image

Going back to Listing 3, the constructor next passes the **BufferedImage** object to the method named **processImg** for processing.

The version of the **processImg** method in this demonstration program uses the image-filtering class named **LookupOp** from the Java 2D API to process the image. However, a representative **processImg** method could have been written using any of the classes provided in the earlier list of image-filtering classes.

## Behavior of the processImg method

The behavior of the **processImg** method is to use the **LookupOp** class to invert all of the color values in the pixels as shown by the bottom image in Figure 3.

> *(Note that this demonstration program does not display the processed image. You will need to use some other program to display the output file named **junk.jpg** to see the processing results.)*

The alpha *(transparency)* value is not modified. As I explained earlier, the process of inverting the color values consists of subtracting each color value from 255.

**What does Sun have to say?**

Figure 21 contains part of what Sun has to say about the **LookupOp** class.

> This class implements a lookup operation from the source to the destination.
>
> The **LookupTable** object may contain a single array or multiple arrays, subject to the restrictions below.
>
> For **Rasters**, **...**
>
> For **BufferedImages**, the lookup operates on color and alpha components.
>
> The number of lookup arrays may be one, in which case the same array is applied to all color *(but not alpha)* components. Otherwise, the number of lookup arrays may equal the number of Source color components, in which case no lookup of the alpha component *(if present)* is performed.
>
> **Figure 21**

**To make a long story short ...**

The **filter** method that is later invoked on an object of the **LookupOp** class 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.

**The processImg method**

The **processImg** method begins in Listing 7.

```
  public BufferedImage processImg(BufferedImage
```

```
theImage){

    //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

Listing 7
```

The code in Listing 7 creates and populates a one-dimensional array of type **short** containing 256 values where the value in each element is equal to 255 minus the element index. This will satisfy the requirement for color inversion described earlier.

## Create the actual lookup table

Listing 8 creates the lookup table by instantiating an object of type **ShortLookupTable** and passing the array of lookup data as the second parameter to the constructor. *(The first parameter is an offset value that makes it possible to cause the early elements in the lookup data array to be ignored.)*

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

Listing 8
```

Figure 22 contains part of what Sun has to say about the **ShortLookupTable** class.

> This class defines a lookup table object. The output of a lookup operation using an object of this class is interpreted as an unsigned short quantity.
>
> The lookup table contains short data arrays for one or more bands *(or components)* of an image, and it contains an offset which will be subtracted from the input values before indexing the arrays. This allows an array smaller than the native data size to be provided for a constrained input. If there is only one array in the lookup table, it will be applied to all bands.
>
> **Figure 22**

## Create the filter object

Listing 9 creates the filter object by instantiating an object of the **LookupOp** class and saving its reference as type **BufferedImageOp**.

*(**BufferedImageOp** is an interface that is implemented by the **LookupOp** class.)*

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

Listing 9
```

The first parameter to the **LookupOp** constructor is a reference to the **ShortLookupTable** object. The second parameter, *(which may optionally be passed as null)*, is a reference to a **RenderingHints** object.

*(A **RenderingHints** object can be used to specify some of the fine details as to how the image is actually drawn on the final viewing surface.)*

## Apply the filter to the image

Listing 10 invokes the **filter** method belonging to the **LookupOp** object to apply the filter to the incoming image. It returns a reference to the resulting **BufferedImage** object that contains the modified image.

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

Listing 10
```

The **filter** method performs the table lookup operation and returns a reference to the modified image as type **BufferedImage**. The second parameter, *(which can optionally be passed as null)*, makes it possible to also cause the modified image to be written into an existing **BufferedImage** destination object.

Listing 10 also signals the end of the **processImg** method.

## Steps in processing an image

When using one of the image-filtering classes of the Java 2D API to directly modify an image stored in a **BufferedImage** object, the typical steps are:

- Create any necessary data and objects required to support the process, as in Listing 7 and Listing 8.
- Create the filter object as in Listing 9.
- Apply the filter object as in Listing 10.

*(If you elect to decompose the image and process individual **Raster** objects, additional steps are required. The **BandCombineOp** filter can operate only on*

*__Raster__ objects, but the other image-filtering classes can operate either on __BufferedImage__ objects or __Raster__ objects.)*

**Write the modified image into a JPEG file**

Returning to the constructor in Listing 3, you can see that the constructor invokes the method named **writeJpegFile** to cause the modified image to be written into an output file named **junk.jpg** in the current directory.

The method named **writeJpegFile** is shown in its entirety in Listing 11.

```
  void writeJpegFile(BufferedImage img){
    try{
      //Get a file output stream.
      FileOutputStream outStream =
                            new
FileOutputStream("junk.jpg");
      //Call the write method of the ImageIO
class to write
      // the contents of the BufferedImage
object to an
      // output file in JPEG format.
      ImageIO.write(img,"jpeg",outStream);
      outStream.close();
    }catch (Exception e) {
      e.printStackTrace();
    }//end catch
  }//end writeJpegFile

Listing 11
```

**There is very little that is new here**

There is very little, if anything, that is new in Listing 11. I have been publishing and using methods similar to this one to write JPEG output files since the earlier lesson entitled Processing Image Pixels, An Improved Image-Processing Framework in Java. The embedded comments in the method, along with the Sun documentation of the classes and methods involved, should suffice to explain the method and no further explanation should be required.

Listing 11 also signals the end of the explanation of the program named **ImgMod05a**.

## The program named *ImgMod05*

This program is an update of the earlier program named **ImgMod04a**.

*(The class named __ImgMod04a__ was first published in the earlier lesson entitled Processing Image Pixels, An Improved Image-Processing Framework in Java.)*

This program is designed to accommodate the use of the image-filtering classes of the Java 2D API.

## Sends and receives a BufferedImage object

This program sends and receives a **BufferedImage** object to an image-processing method of a compatible image-processing object instead of sending and receiving an array of pixel data as is the case for the program named **ImgMod04a**.

## Purpose

The purpose of this program is to make it easy to experiment with the modification of image data using the image-filtering classes of the Java 2D API and to display the modified version of the image along with the original image.

In addition to the graphic display of the modified image, the program 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

The program GUI 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 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**.

The **Replot** button is located at the top of the display to make it accessible when the display is too tall to fit on the screen.

> *(For purposes of seeing the entire display in that case, it can be moved up and down on the screen using the right mouse button and the up and down arrow keys.)*

## Input and output file format

The 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.

## A framework program

This program provides a framework that is designed to invoke another program to process an input image. The program reads the image from the input file and converts it to type **BufferedImage**. A second program is invoked to actually process the image.

Typically the image-processing program is based on the image-filtering classes of the Java 2D API, but that is not a requirement. The only requirement is that the image-processing program be capable of receiving the image as type **BufferedImage** and returning the processed image as type **BufferedImage**.

## Typical usage

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

```
java ImgMod05 ProcessingProgramName ImageFileName
```

## A self-contained test program

For test and illustration purposes, the source code includes a class definition for a sample image-processing program named **ProgramTest**. If the command-line parameters are omitted, the program will search for an image file named **ImgMod05Test.jpg** in the current directory and will process it using the sample image-processing program named **ProgramTest**.

The sample program named **ProgramTest** returns a reference to a **BufferedImage** object in which the colors in the modified image are inverted relative to the colors in the original image.

## The input image file

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

## Display of the images

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 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.

## The Replot button

A **Replot** button appears at the top of the frame. The behavior of the **Replot** button is as described above causing a newly processed version of the original image to replace the earlier processed version in the display.

## User data input

The processing program 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. *(The sample image-processing program that is built into this program does not provide that capability.)*

## The interface named ImgIntfc05

The image-processing program must implement the interface named **ImgIntfc05**. That interface declares a single image-processing method with the following signature:

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

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

A complete listing of the interface named **ImgIntfc05** is provided in Listing 31.

## Miscellaneous items

If the image-processing program has a **main** method, it will be ignored.

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

This program was tested using J2SE5.0 under WinXP.

## Will discuss in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 30.

The program begins in Listing 12 by declaring a pair of **BufferedImage** variables, just like in the program named **ImgMod05a**, shown in Listing 1.

```
class ImgMod05 extends Frame{
  BufferedImage rawBufferedImage;
  BufferedImage processedImage;

Listing 12
```

## Frame insets

Listing 13 declares a **Frame** object that is used to create the display shown in Figure 1.

```
  Frame displayFrame;//Frame to display the
images.
```

```
  int inLeft;//left inset
  int inTop;//top inset
  int inBottom;//bottom inset
  int buttonHeight;//Height of Replot button

Listing 13
```

Listing 13 also declares four **int** variables that will contain the **Frame** insets, which are used later to construct the display shown in Figure 1.

## Self-contained test program and default image file

Listing 14 declares a **String** variable and initializes it with the name of the self-contained image-processing program. Listing 14 also declares a **String** variable and initializes it with the name of the default image file.

```
  static String theProcessingClass =
"ProgramTest";

  static String theImgFile =
"ImgMod05Test.jpg";

Listing 14
```

The class identified by the string in Listing 14 is executed to process the image file identified in Listing 14 if the user fails to enter a pair of command-line parameters. You must provide this file in the current directory if it will be needed.

> *(You should be able to use any JPEG file for this purpose, provided you assign the correct file name to it.)*

The source code for the class identified by the string in Listing 14 is included later in this source code file.

## Miscellaneous instance variables

Listing 15 declares four additional instance variables, the purpose of which should become obvious as you study the code that follows.

```
  MediaTracker tracker;
  Display display = new Display();//A Canvas
object
  Button replotButton = new Button("Replot");

  //Reference to the image-processing object.
  ImgIntfc05 imageProcessingObject;

Listing 15
```

## The main method

The **main** method is shown in its entirety in Listing 16.

```
  public static void main(String[] args){
    //Get names for the image-processing class
and the
    // image file to be processed.  Program
reads gif
    // files and jpg files and possibly some
other file
    // types as well.
    if(args.length == 0){
      //Use default processing class and
default image
      // file.  Class and file names were
specified above.
    }else if(args.length == 2){
      theProcessingClass = args[0];
      theImgFile = args[1];
    }else{
      System.out.println("Invalid args");
      System.exit(1);
    }//end else

    //Display name of processing program and
image file.
    System.out.println(
              "Processing program: " +
theProcessingClass);
    System.out.println("Image file: " +
theImgFile);

    //Instantiate an object of this class.
    ImgMod05 obj = new ImgMod05();
  }//end main
```

**Listing 16**

Most of the code in Listing 16 is used to get the name of the image-processing class and the name of the image file to be processed.

Finally, the code in the **main** method in Listing 16 invokes the constructor to construct an object of the **ImgMod05** class.

All the code in Listing 16 is straightforward and shouldn't require further explanation.

## The constructor for the ImgMod05 class

The constructor begins in Listing 17 by invoking the method named **getTheImage** to get the image from the input file and to store it in an object of type **BufferedImage**.

```
  public ImgMod05(){//constructor
    //Get an image from the specified image
file.  Can be
    // in a different directory if the path was
entered
    // with the file name on the  command line.
    rawBufferedImage = getTheImage();
```

**Listing 17**

The method named **getTheImage** used in this program is essentially the same as the method having the same name that was explained in Listing 4, Listing 5, and Listing 6. Therefore, it should not be necessary to repeat that explanation.

You can view a complete listing of the method named **getTheImage** in Listing 30 near the end of the lesson.

## Construct the display object

Listing 18 contains code typical of that commonly used to construct an object of type **Frame** and to add a **Canvas** object and a **Button** object to the frame.

```
    //Construct the display object.
    this.setTitle("Copyright 2006, Baldwin");
    this.setBackground(Color.YELLOW);
    this.add(display);
    this.add(replotButton,BorderLayout.NORTH);

    //Make the frame visible to make it
possible to
    // get insets and the height of the button.
    setVisible(true);
    //Get and store inset data for the Frame
and the height
    // of the button.
    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;
    inBottom = this.getInsets().bottom;
    buttonHeight =
replotButton.getSize().height;

    //Save a reference to this Frame object for
use in
    // setting the size of the Frame later.
    displayFrame = this;
```

**Listing 18**

The code in Listing 18 along with the embedded comments should be self-explanatory and shouldn't require further explanation.

## An ActionListener on the Replot button

Continuing with the constructor, Listing 19 contains the beginning of an anonymous inner class listener for the **Replot** button. This **actionPerformed** method is invoked when the user clicks the **Replot** button. It is also invoked at startup when this program posts an **ActionEvent** to the system event queue attributing the event to the **Replot** button.

```
    replotButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent
e){
          //Process the image.
          System.out.println("\nProcess the
image");
          processedImage =

imageProcessingObject.processImg(

rawBufferedImage);
          System.out.println("Image
processed");

Listing 19
```

Listing 19 invokes the **processImg** method of the image-processing object, passing the **BufferedImage** object containing the original image to that method. The **processImg** method returns a reference to a **BufferedImage** object containing the modified image. The reference to the modified image is stored by the constructor in the variable named **processedImage**.

## Adjust the size of the display

The processed image may be larger than the original image. Therefore, the code in Listing 20 attempts to set the display size to accommodate the raw and processed images. In the event that the processed image won't fit in the display frame after the size adjustment is made by the code in Listing 20, the user can manually resize the frame.

```
          int maxWidth = 0;
          //Get max image width.
          if(processedImage.getWidth() >

rawBufferedImage.getWidth()){
            maxWidth =
processedImage.getWidth();
          }else{
            maxWidth =
rawBufferedImage.getWidth();
          }//end else
          int totalWidth = 2*inLeft + maxWidth
+ 2;

          //Get height of two images.
```

```
          int height =
rawBufferedImage.getHeight()
                                +
processedImage.getHeight();
          int totalHeight =
              inTop + inBottom + buttonHeight +
height + 4;

displayFrame.setSize(totalWidth,totalHeight);
```

**Listing 20**

Listing 20 sets the size of the display such that for non-rotated images, a tiny amount of the background color shows between the two images, to the right of the larger image, and below the bottom image as shown in Figure 1.

## Validate and repaint the display

Occasionally, *(on an intermittent basis)*, without the addition of the first statement in Listing 21, even though the **repaint** method is called, the operating system doesn't make a call to the overridden **paint** method. As a result, the original and processed images don't appear in the display frame and the program appears to be hung up in an indeterminate state.

```
          displayFrame.validate();

          System.out.println("Call repaint");
          //Repaint the image display frame
with the
          // original image at the top and the
modified
          // image at the bottom.
          display.repaint();
          System.out.println("Repaint call
complete");
```

**Listing 21**

## Why validate?

The Sun documentation for the **Container** class states

> *"If a component has been added to a container that has been displayed,* ***validate*** *must be called on that container to display the new component."*

Apparently the same thing holds true when the size of the container is changed. Inclusion of the first statement in Listing 21 seems to fix the intermittent problem.

Listing 21 also invokes the **repaint** method, asking the operating system to call the overridden **paint** method to repaint the display.

## Write the output image file

Listing 22 invokes the method named **writeJpegFile** to cause the modified image to be written into an output JPEG file named **junk.jpg**.

```
          System.out.println("Call
writeJpegFile");
          writeJpegFile(processedImage);
          System.out.println(
                              "writeJpegFile call
complete");
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener
    //End anonymous inner class registered on
the Replot
    // button.

Listing 22
```

This is essentially the same **writeJpegFile** method that was explained in Listing 11. Therefore I won't repeat that explanation. You can view the method named **writeJpegFile** in Listing 30.

Listing 22 also signals the end of the definition of the anonymous **ActionListener** class and the registration of an object of that class on the **Replot** button.

## Instantiate an image-processing object

Continuing with the constructor, Listing 23 instantiates a new object of the image-processing class.

```
    try{
      imageProcessingObject =
(ImgIntfc05)Class.forName(

theProcessingClass).newInstance();

Listing 23
```

Note that this object is instantiated using the **newInstance** method of the class named **Class**. This approach does not allow for the use of a parameterized constructor for the image-processing class.

## Post a counterfeit ActionEvent

Listing 24 posts a counterfeit **ActionEvent** to the system event queue and attributes it to the **Replot** button.

*(See the anonymous ActionListener class that registers an ActionListener object on the **Replot** button above.)*

Posting this event causes the image-processing method to be invoked and causes the modified image to be displayed at startup.

```
Toolkit.getDefaultToolkit().getSystemEventQueue().
        postEvent(
          new ActionEvent(replotButton,

ActionEvent.ACTION_PERFORMED,
                          "Replot")
        );//end postEvent method

    }catch(Exception e){
      e.printStackTrace();
      System.exit(1);
    }//end catch
```

**Listing 24**

## From this point forward ...

At this point, the image has been processed and the original image along with the modified image has been displayed.

From this point forward, each time the user clicks the **Replot** button:

- A new image-processing object will be instantiated.
- The original image will be processed again.
- The new modified image will be displayed along with the original image.
- The modified image will be written into an output JPEG file.

## Make everything visible

Still in the constructor, Listing 25 causes the composite of the frame, the canvas, and the button shown in Figure 1 to become visible.

```
    this.setVisible(true);
```

**Listing 25**

## Define and register an anonymous terminator class

Wrapping up the constructor, Listing 26 defines and registers a **WindowListener** object that causes the program to terminate when the user clicks the button with the X in the upper right corner of the frame.

```
    this.addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener

//=====================================================//

  }//end ImgMod05 constructor

//=====================================================//
```

**Listing 26**

## Define the Display class

Listing 27 defines the inner **Display** class. This class is used to instantiate a **Canvas** object on which the original and modified images are displayed.

```
  //Inner class for canvas object on which to
display the
  // two images.
  class Display extends Canvas{
    //Override the paint method to display the
raw image
    // and the modified image on the same
Canvas object,
    // separated by a couple of rows of pixels
in the
    // background color.
    public void paint(Graphics g){
      //First confirm that the image has been
completely
      // loaded and that none of the image
references are
      // null.
      if (tracker.statusID(1,false) ==

MediaTracker.COMPLETE){
        if((rawBufferedImage != null) &&
           (processedImage != null)){
          //Draw raw image at the top.
Terminate if the
          // the pixels are changing.
          boolean success = false;
          success =
g.drawImage(rawBufferedImage,0,0,this);
          if(!success){
            System.out.println("Unable to draw
top image");
            System.exit(1);
          }//end if
```

```
          //Draw processed image at the bottom.
          success =
g.drawImage(processedImage,0,

rawBufferedImage.getHeight() + 2,this);
          if(!success){
            System.out.println(
                              "Unable to draw
bottom image");
            System.exit(1);
          }//end if
        }//end if
      }//end if
    }//end paint()
  }//end class myCanvas

Listing 27
```

The code in Listing 27 is very similar to the code that I explained for the class having the same name in the earlier lesson entitled Processing Image Pixels using Java, Getting Started. Therefore, I won't repeat that explanation here.

## The ProgramTest class

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**. The beginning of the **ProgramTest** class is shown in Listing 28.

```
class ProgramTest implements ImgIntfc05{

  //The following method must be defined to
implement the
  // ImgIntfc05 interface.
  //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 ProgramTest

Listing 28
```

## Must implement ImgIntfc05

A compatible class 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**.

## A color-inverter program

The method named **processImg** in this sample program is a color inverter method.

The method 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 alpha values are not modified.

## The type of image is important

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 parameterized constructor is 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** class on the basis of the name of this class **as a String**. That process does not allow for constructor parameters for the class being instantiated.

## The processImg method

The **ProgramTest** class in Listing 28 consists of the single method named **processImg**. The **processImg** method is essentially the same as the method having the same name that I explained in conjunction with Listing 7 through Listing 10. Therefore, I won't repeat that explanation here.

Listing 28 also signals the end of the explanation of the program named **ImgMod05**.

# Run the Programs

I encourage you to copy the code from Listing 29, Listing 30, and Listing 31 into your text editor, compile and execute the code.

Make changes to the code and experiment with it. Above all, enjoy learning new things about object-oriented image processing using Java.

# Summary

In this lesson, I taught you a little about the image-filtering classes of the Java 2D API. I also showed 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.

# What's Next?

Future lessons will teach you more about the **LookupOp** class of the Java 2D API, and will also teach you how to use the following image-filtering classes from that API:

- **AffineTransformOp** class
- **BandCombineOp** class
- **ConvolveOp** class
- **RescaleOp** class
- **ColorConvertOp** class

# 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


# Complete Program Listings

Complete listings of the programs discussed in this lesson are shown in Listing 29, Listing 30, and Listing 31 below.

```
/*File ImgMod05a.java
Copyright 2006, R.G.Baldwin

The purpose of this program is to show how to read, modify,
and write an image file using Java2D image-processing
operations.  The program was simplified as much as
practical.

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

import java.awt.*;
import java.io.*;
import javax.imageio.*;
import java.awt.image.*;

class ImgMod05a{
  BufferedImage rawBufferedImage;
  BufferedImage processedImage;
  static String theImgFile = "ImgMod05Test.jpg";
  MediaTracker tracker;

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

  public static void main(String[] args){
    //Instantiate an object of this class.
    ImgMod05a obj = new ImgMod05a();
  }//end main
  //--------------------------------------------//

  public ImgMod05a(){//constructor
    //Get an image from the specified image file.
    rawBufferedImage = getTheImage();

    //Process the image.
    processedImage = processImg(rawBufferedImage);

    //Write the modified image into a JPEG file named
    // junk.jpg.
    writeJpegFile(processedImage);

  }//end ImgMod05a constructor
  //==================================================//

  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
  //=====================================================//

  //Write the contents of a BufferedImage object to a JPEG
  // file named junk.jpg.
  void writeJpegFile(BufferedImage img){
    try{
      //Get a file output stream.
      FileOutputStream outStream =
                            new FileOutputStream("junk.jpg");
      //Call the write method of the ImageIO class to write
      // the contents of the BufferedImage object to an
      // output file in JPEG format.
      ImageIO.write(img,"jpeg",outStream);
      outStream.close();
    }catch (Exception e) {
      e.printStackTrace();
    }//end catch
  }//end writeJpegFile
  //-----------------------------------------------------//

  //This method reads an image from a specified image file,
  // writes it into a BufferedImage object, and returns a
  // reference to the BufferedImage object.
  //The name of the image file is contained in an instance
  // variable of type String named theImgFile.
  BufferedImage getTheImage(){
    Image rawImage = Toolkit.getDefaultToolkit().
                                      getImage(theImgFile);

    //Use a MediaTracker object to block until the image is
    // loaded or ten seconds has elapsed.  Terminate and
    // display an error message if ten seconds elapse
    // without the image having been loaded.  Note that the
    // constructor for the MediaTracker requires the
    // specification of a Component "on which the images
    // will eventually be drawn" even if there is no
    // intention for the program to actually display the
    // image.  It is useful to have a media tracker with a
    // timeout even if the image won't be drawn by the
    // program.  Also, the media tracker is needed to delay
```

```
    // execution until the image is fully loaded.
    tracker = new MediaTracker(new Frame());
    tracker.addImage(rawImage,1);

    try{
      if(!tracker.waitForID(1,10000)){
        System.out.println("Timeout or Load error.");
        System.exit(1);
      }//end if
    }catch(InterruptedException e){
      e.printStackTrace();
      System.exit(1);
    }//end catch

    //Make certain that the file was successfully loaded.
    if((tracker.statusAll(false)
                           & MediaTracker.ERRORED
                           & MediaTracker.ABORTED) != 0){
      System.out.println("Load errored or aborted");
      System.exit(1);
    }//end if

    //Create an empty BufferedImage object.  Note that the
    // specified image type is critical to the correct
    // operation of the image-processing method. The method
    // may work correctly for other image types, but has
    // been tested only for TYPE_INT_RGB.  The
    // parameters to the getWidth and getHeight methods are
    // references to ImageObserver objects, or references
    // to "an object waiting for the image to be loaded."

    BufferedImage buffImage = new BufferedImage(
                               rawImage.getWidth(null),
                               rawImage.getHeight(null),
                               BufferedImage.TYPE_INT_RGB);

    // Draw Image into BufferedImage
    Graphics g = buffImage.getGraphics();
    g.drawImage(rawImage, 0, 0, null);

    return buffImage;
  }//end getTheImage
  //-----------------------------------------------------//
}//end ImgMod05a.java class
//=======================================================//
```

**Listing 29**

Listing 30

```
/*File ImgMod05.java
Copyright 2006, R.G.Baldwin

This is an update of the class named ImgMod04a designed to
accommodate the use of the image-processing operations of
```

```
the Java 2D API.  When this class is run as a program, it
sends and receives a BufferedImage object to an image
processing method of a compatible image-processing object
instead of sending and receiving an array of pixel data as
is the case in the class named ImgMod04a.

The purpose of this program is to make it easy to
experiment with the modification of image data using the
image-processing operations of the Java 2D API and to
display a modified version of the image along with the
original image.

This program 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 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.

The Replot button is located at the top of the display to
make it accessible when the display is too tall to fit
on the screen.  (For purposes of seeing the entire
display in that case, it can be moved up and down on the
screen using the right mouse button and the up and down
arrow keys.)

The 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.

This program provides a framework that is designed to
invoke another program to process an input image.  This
program reads the image from the input file and converts it
to type BufferedImage.  A second program is invoked to
actually process the image.

Typically the image-processing program is based on the
image-processing operations of the Java 2D API, but that is
not a requirement.  The only requirement is that the image
processing program be capable of receiving the image as
type BufferedImage and returning the processed image as
type BufferedImage.

Typical usage is as follows:

java ImgMod05 ProcessingProgramName ImageFileName

For test and illustration purposes, the source code
includes a class definition for a sample image-processing
program named ProgramTest.
```

```
If the command-line parameters are omitted, the program
will search for an image file in the current directory
named ImgMod05Test.jpg and will process it using the sample
image-processing program named ProgramTest.  The sample
program returns a reference to a BufferedImage object in
which the colors in the modified image are inverted
relative to the colors in the original image.

The image file must be provided by the user in all cases.
However, it doesn't have to be in the current directory if
a path to the file is specified 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
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.

A Replot button appears at the top of the frame.  The
behavior of the Replot button is as described above
causing a newly processed version of the original image to
replace the earlier processed version in the display.

The processing program 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.  (The sample image-processing program does not
provide that capability.)

The image-processing program must implement the
interface named ImgIntfc05.  That interface declares an
image-processing method with the following signature:

public BufferedImage processImg(BufferedImage input);

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

The image-processing method must return a reference to a
BufferedImage object containing the processed image.

If the image-processing program has a main method, it will
be ignored.

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.*;
import java.awt.event.*;
import java.io.*;
import javax.imageio.*;
```

```
import java.awt.image.*;

class ImgMod05 extends Frame{
  BufferedImage rawBufferedImage;
  BufferedImage processedImage;

  Frame displayFrame;//Frame to display the images.
  int inLeft;//left inset
  int inTop;//top inset
  int inBottom;//bottom inset
  int buttonHeight;//Height of Replot button

  //This is the name of the default image-processing
  // program.  This class will be executed to process the
  // image if there aren't two command-line parameters.
  // The source code for this class file is included in
  // this source code file.
  static String theProcessingClass = "ProgramTest";

  //This is the name of the default image file.  This image
  // file will be processed if there aren't two command-
  // line parameters.  You must provide this file in the
  // current directory if it will be needed.
  static String theImgFile = "ImgMod05Test.jpg";

  MediaTracker tracker;
  Display display = new Display();//A Canvas object
  Button replotButton = new Button("Replot");

  //Reference to the image-processing object.
  ImgIntfc05 imageProcessingObject;
  //-----------------------------------------------------//

  public static void main(String[] args){
    //Get names for the image-processing class and the
    // image file to be processed.  Program reads gif
    // files and jpg files and possibly some other file
    // types as well.
    if(args.length == 0){
      //Use default processing class and default image
      // file.  Class and file names were specified above.
    }else if(args.length == 2){
      theProcessingClass = args[0];
      theImgFile = args[1];
    }else{
      System.out.println("Invalid args");
      System.exit(1);
    }//end else

    //Display name of processing program and image file.
    System.out.println(
              "Processing program: " + theProcessingClass);
    System.out.println("Image file: " + theImgFile);

    //Instantiate an object of this class.
    ImgMod05 obj = new ImgMod05();
```

```
  }//end main
  //---------------------------------------//

  public ImgMod05(){//constructor
    //Get an image from the specified image file.  Can be
    // in a different directory if the path was entered
    // with the file name on the  command line.
    rawBufferedImage = getTheImage();

    //Construct the display object.
    this.setTitle("Copyright 2006, Baldwin");
    this.setBackground(Color.YELLOW);
    this.add(display);
    this.add(replotButton,BorderLayout.NORTH);

    //Make the frame visible to make it possible to
    // get insets and the height of the button.
    setVisible(true);
    //Get and store inset data for the Frame and the height
    // of the button.
    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;
    inBottom = this.getInsets().bottom;
    buttonHeight = replotButton.getSize().height;

    //Save a reference to this Frame object for use in
    // setting the size of the Frame later.
    displayFrame = this;

    //================================================//
    //Anonymous inner class listener for Replot button.
    // This actionPerformed method is invoked when the user
    // clicks the Replot button.  It is also invoked at
    // startup when this program posts an ActionEvent to
    // the system event queue attributing the event to the
    // Replot button.
    replotButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          //Process the image.
          System.out.println("\nProcess the image");
          processedImage =
                          imageProcessingObject.processImg(
                                            rawBufferedImage);
          System.out.println("Image processed");
          //Set the display size to accommodate the raw and
          // processed images.  In the event that the
          // processed image won't fit in the display
          // frame, the user can manually resize the frame.
          // Set the size such that for non-rotated images,
          // a tiny amount of the background color shows
          // between the two images, to the right of the
          // larger image, and below the bottom image.
          int maxWidth = 0;
          //Get max image width.
          if(processedImage.getWidth() >
```

```
                              rawBufferedImage.getWidth()){
          maxWidth = processedImage.getWidth();
        }else{
          maxWidth = rawBufferedImage.getWidth();
        }//end else
        int totalWidth = 2*inLeft + maxWidth + 2;

        //Get height of two images.
        int height = rawBufferedImage.getHeight()
                           + processedImage.getHeight();
        int totalHeight =
            inTop + inBottom + buttonHeight + height + 4;
        displayFrame.setSize(totalWidth,totalHeight);

        //Occasionally on an intermittent basis, without
        // the addition of the following statement, even
        // though the repaint method is called, the OS
        // doesn't make a call to the overridden paint
        // method.  As a result, the original and
        // processed images don't appear in the display
        // frame and it appears to be hung up in an
        // intermediate state.  The Sun documentation for
        // the Container class states "If a component has
        // been added to a container that has been
        // displayed, validate must be called on that
        // container to display the new component."
        // Apparently the same thing holds true when the
        // size of the container is changed.  Inclusion
        // of the following statement seems to fix the
        // intermittent problem.
        displayFrame.validate();

        System.out.println("Call repaint");
        //Repaint the image display frame with the
        // original image at the top and the modified
        // image at the bottom.
        display.repaint();
        System.out.println("Repaint call complete");
        //Write the modified image into a JPEG file named
        // junk.jpg.
        System.out.println("Call writeJpegFile");
        writeJpegFile(processedImage);
        System.out.println(
                           "writeJpegFile call complete");
      }//end actionPerformed
    }//end ActionListener
  );//end addActionListener
  //End anonymous inner class registered on the Replot
  // button.
  //================================================//

  //Continuing with the constructor code ...

  //Instantiate a new object of the image-processing
  // class.  Note that this object is instantiated using
  // the newInstance method of the class named Class.
```

```
    // This approach does not allow for the use of a
    // parameterized constructor.
    try{
      imageProcessingObject = (ImgIntfc05)Class.forName(
                          theProcessingClass).newInstance();

       //Post a counterfeit ActionEvent to the system event
       // queue and attribute it to the Replot button.
       // (See the anonymous ActionListener class that
       // registers an ActionListener object on the Replot
       // button above.)  Posting this event causes the
       // image-processing method to be invoked at startup
       // and causes the modified image to be displayed.
      Toolkit.getDefaultToolkit().getSystemEventQueue().
        postEvent(
          new ActionEvent(replotButton,
                          ActionEvent.ACTION_PERFORMED,
                          "Replot")
        );//end postEvent method

      //At this point, the image has been processed.  The
      // original image and the modified image have been
      // displayed.  From this point forward, each time the
      // user clicks the Replot button, a new image
      // processing will be instantiated, the image will be
      // processed again, and the new modified image will
      // be displayed along with the original image.

    }catch(Exception e){
      e.printStackTrace();
      System.exit(1);
    }//end catch

    //Cause the composite of the frame, the canvas, and the
    // button to become visible.
    this.setVisible(true);

    //================================================//

    //Anonymous inner class listener to terminate
    // program.
    this.addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
    //================================================//

  }//end ImgMod05 constructor
  //================================================//

  //Inner class for canvas object on which to display the
  // two images.
  class Display extends Canvas{
```

```
    //Override the paint method to display the raw image
    // and the modified image on the same Canvas object,
    // separated by a couple of rows of pixels in the
    // background color.
    public void paint(Graphics g){
      //First confirm that the image has been completely
      // loaded and that none of the image references are
      // null.
      if (tracker.statusID(1,false) ==
                                      MediaTracker.COMPLETE){
        if((rawBufferedImage != null) &&
           (processedImage != null)){
          //Draw raw image at the top.  Terminate if the
          // the pixels are changing.
          boolean success = false;
          success = g.drawImage(rawBufferedImage,0,0,this);
          if(!success){
            System.out.println("Unable to draw top image");
            System.exit(1);
          }//end if
          //Draw processed image at the bottom.
          success = g.drawImage(processedImage,0,
                    rawBufferedImage.getHeight() + 2,this);
          if(!success){
            System.out.println(
                           "Unable to draw bottom image");
            System.exit(1);
          }//end if
        }//end if
      }//end if
    }//end paint()
  }//end class myCanvas
//=============================================================//

  //Write the contents of a BufferedImage object to a JPEG
  // file named junk.jpg.
  void writeJpegFile(BufferedImage img){
    try{
      //Get a file output stream.
      FileOutputStream outStream =
                           new FileOutputStream("junk.jpg");
      //Call the write method of the ImageIO class to write
      // the contents of the BufferedImage object to an
      // output file in JPEG format.
      ImageIO.write(img,"jpeg",outStream);
      outStream.close();
    }catch (Exception e) {
      e.printStackTrace();
    }//end catch
  }//end writeJpegFile
  //-------------------------------------------------------//

  //This method reads an image from a specified image file,
  // writes it into a BufferedImage object, and returns a
  // reference to the BufferedImage object.
  //The name of the image file is contained in an instance
```

```
  // variable of type String named theImgFile.
  BufferedImage getTheImage(){
    Image rawImage = Toolkit.getDefaultToolkit().
                                      getImage(theImgFile);

    //Use a MediaTracker object to block until the image is
    // loaded or ten seconds has elapsed.  Terminate and
    // display an error message if ten seconds elapse
    // without the image having been loaded.
    tracker = new MediaTracker(this);
    tracker.addImage(rawImage,1);

    try{
      if(!tracker.waitForID(1,10000)){
        System.out.println("Load error.");
        System.exit(1);
      }//end if
    }catch(InterruptedException e){
      e.printStackTrace();
      System.exit(1);
    }//end catch

    //Make certain that the file was successfully loaded.
    if((tracker.statusAll(false)
                             & MediaTracker.ERRORED
                             & MediaTracker.ABORTED) != 0){
      System.out.println("Load errored or aborted");
      System.exit(1);
    }//end if

    //Create an empty BufferedImage object.  Note that the
    // specified image type is critical to the correct
    // operation of the image-processing method. The method
    // may work correctly for other image types, but has
    // been tested only for TYPE_INT_RGB.
    BufferedImage buffImage = new BufferedImage(
                               rawImage.getWidth(this),
                               rawImage.getHeight(this),
                               BufferedImage.TYPE_INT_RGB);

    // Draw Image into BufferedImage
    Graphics g = buffImage.getGraphics();
    g.drawImage(rawImage, 0, 0, null);

    return buffImage;
  }//end getTheImage
  //-----------------------------------------------------//
}//end ImgMod05.java class
//=======================================================//

//The ProgramTest class

//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.  A compatible class
// 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 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 alpha values
// are not modified.

//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 on the basis of the name of this
class
// as a String.  That process
// does not allow for constructor parameters for the class
// being instantiated.
class ProgramTest implements ImgIntfc05{

  //The following method must be defined to implement the
  // ImgIntfc05 interface.
  //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 ProgramTest


Listing 30
```

Listing 31

```
/*File ImgIntfc05.java
Copyright 2006, R.G.Baldwin

The purpose of this interface is to declare the method
required by image-processing classes that are
compatible with the program named ImgMod05.java.

Tested using J2SE 5.0 under WinXP
**********************************************************/

import java.awt.image.BufferedImage;

interface ImgIntfc05{
  public BufferedImage processImg(BufferedImage input);
}//end ImgIntfc05
//========================================================//

Listing 31
```

---



**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

-end-