Using the Java 2D LookupOp Filter Class to Scramble and Unscramble Images

*Learn how to use the **LookupOp** image-filtering class from the Java 2D API, along with the **Random** class from the **java.util** package to write a pair of easy-to-use programs to scramble and unscramble images in a reasonably secure fashion.*

Published: June 5, 2007
By Richard G. Baldwin

Java Programming Notes # 456

- Preface
- Preview
- Discussion and Sample Code
  - The Program Named ImgMod47
  - The Program Named ImgMod46a
  - The Program Named ImgMod46b
- Run the Program
- Summary
- References
- Complete Program Listing

Preface

The scenario

Pretend for a moment that you are a fashion designer, or perhaps an automotive designer. You and your colleagues around the world frequently need to exchange high quality images on a timely basis via email. Because the design world is highly competitive, you would prefer to exchange those images in a reasonably secure manner to prevent them from falling into the hands of your competitors who might steal your ideas. On the other hand, you would like the effort required to achieve a reasonable degree of security to be minimal.

Scrambling and unscrambling images

In this lesson, I will show you how to use the **LookupOp** image-filtering class from the Java 2D API, along with the **Random** class from the **java.util** package to write a pair of easy-to-use programs to scramble and unscramble images in a reasonably secure fashion.

How secure are the images?

Clearly, this method of protecting images wouldn't survive the code breakers at the National Security Agency. It probably also wouldn't survive the code breakers at a number of universities and research consortiums that specialize in cryptography. However, I believe that it would survive for quite some time against less accomplished members of the industrial espionage community, particularly if a different scrambling key is used for every image that is exchanged.

A disclaimer

On the other hand, **I have no expertise in cryptography**, so I may be surprised to learn that the images can be unscrambled by the teenager next door using a hand calculator. Therefore, I am providing these programs for **educational use only**. If you use them for any purpose, you are using them at your own risk. I will not be responsible for any damages that you may incur through the use of these programs.

A useful educational exercise

In any event, regardless of the degree of security that these two programs may or may not provide, they do provide a useful learning experience in the use of the **LookupOp** image-filtering class of the Java 2D API.

Viewing tip

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

Supplementary material

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

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

Preview

In this lesson, I will provide the following three programs:

- **ImgMod47** - Image display program.
- **ImgMod46a** - Image scrambling program.
- **ImgMod46b** - Image unscrambling program.

I will explain the code for the last two programs in the above list.

A utility program

The first program in the list, *(named **ImgMod47**)***,** is a utility program that I wrote because I didn't already have a utility program *(other than a web browser)* that would display image files in the Portable Network Graphics (PNG) format. I needed to be able to display those files in order to write this lesson. I decided to write the program so that it would display two images in the same frame as shown in Figure 1.

This program should be able to display JPEG, GIF, BMP, and PNG files, and possibly some other file types as well. Although I am providing this program for your use, *(see Listing 11 near the end of the lesson)*, I won't attempt to explain the code. Hopefully the comments in the code will suffice for that purpose.

What is the PNG format?

According to the W3C:

> *"PNG is an extensible file format for the lossless, portable, well-compressed storage of raster images. PNG provides a patent-free replacement for GIF and can also replace many common uses of TIFF. Indexed-color, grayscale, and truecolor images are supported, plus an optional alpha channel for transparency. Sample depths range from 1 to 16 bits per component (up to 48bit images for RGB, or 64bit for RGBA)."*

The PNG format seems to be reasonably well-supported by Java. I elected to use it because I needed to be able to store scrambled images in a lossless file format.

Scrambling and unscrambling programs

The program named **ImgMod46a** can be used to scramble an image and to store the scrambled image in an output PNG file. The program named **ImgMod46b** can be used to read the PNG file, to unscramble the image, and to write the unscrambled image into an output JPEG file. *(See Figure 1 for an example of a scrambled image and the unscrambled version of the same image.)*

Scrambling quality

As you will see in the examples that follow, if the input image is a high quality image containing lots of different colors *(such as 24-bit RGB with more than 16-million colors)*, the scrambled image has very little resemblance to the original image. On the other hand, if the input image is a low quality image *(such a GIF image with only 256 colors)*, the major features of the original image are often revealed by the scrambled image.

A shared key

A user-provided key, *(which is a **long** integer)*, is used to seed the random number generator that is used to scramble the image. As a **long** integer, the value of the key can range:

From -9223372036854775808 to 9223372036854775807

Given the large number of possible keys, the likelihood of someone guessing the value of the key is rather remote.

Not like horseshoes and hand grenades...

Unlike playing horseshoes and tossing hand grenades, guessing close to the actual key value doesn't count. As you will see later, a key value that differs from the actual key by only two units in the least significant digit is of no value in an attempt to unscramble the image.

The scrambling algorithm

The scrambling algorithm is very simple. However, because of the use of the shared key to scramble and unscramble the image, simply knowing the algorithm that was used to scramble the image should be of no particular value to an attacker who is attempting to unscramble the image without knowing the key.

Although I can certainly be proven wrong, I believe that it would be extremely difficult, if not impossible, to unscramble the image without knowing both the algorithm and the key that was used to scramble the original image. Once again, this assumes that a different key is used to scramble every image, thereby eliminating the possibility of making comparisons between different images that were scrambled using the same key.

The random number generator.
This all hinges on the quality of the pseudo-random number sequences produced by the Random class of the java.util package.

A brute force attack

Of course, given enough time and computer resources, an attacker could use a brute force approach and try every possible key within the allowable range of keys given above. *(There are a lot of possible keys within the allowable range.)* However, the only way for an attacker to know if an attack was successful would be to view the unscrambled image to see if it makes any sense. This would require the attacker to examine every resulting image to determine if the attack was successful. Given the wide range of allowable keys, this would probably require more time, effort, and eyestrain than most people would be willing to invest.

No expertise in cryptography.
Once again, however, I have no expertise in cryptography, so I may be surprised to learn that the images can be unscrambled by the teenager next door using a hand calculator, so if you use these programs for any purpose, you are using them at your own risk.

Key management

As with all security systems that make use of shared keys, key management could be a problem. That is the main reason that public key cryptography was invented. In my hypothetical scenario involving fashion designers, the group of designers who are collaborating on the

swimwear design for the next summer season could simply exchange several hundred numbered keys via postal mail in advance. Then they could use and identify one of the numbered keys each time an image is scrambled and shared among them via email. *(Each time an image is scrambled and shared, all members of the group would scratch that key off the list to make certain that it isn't used again later.)*

Format for example images

In the following examples, I am going to show you the scrambled image at the top and the unscrambled image at the bottom as shown in Figure 1. To avoid redundancy, I won't show you the original image. I will simply ask you to trust me when I tell you that the unscrambled image matches the original.

Some example images

Figure 1 shows an example for which this image-scrambling technique is well suited.

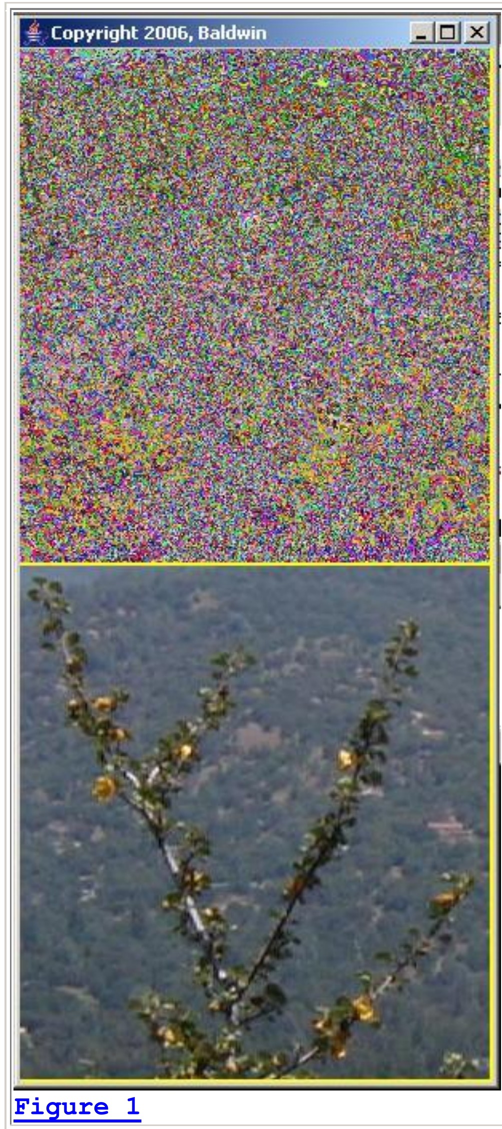

Figure 1

The top panel in Figure 1 shows the scrambled image and the bottom panel shows the unscrambled image. The unscrambled image is a good visual match for the original image.

A digital photograph

This image is the result of a digital photograph that I took somewhere while on vacation and extracted from the camera as a JPEG file. The color depth for this image is large with the possibility of more than sixteen-million different colors.

The following key was used both to scramble and to unscramble the image:

```
Scrambling Key:   -9223372036854775808
Unscrambling Key: -9223372036854775808
```

A slightly different unscrambling key

The bottom panel in Figure 2 shows the result of attempting to unscramble the same scrambled image using an unscrambling key that differs from the actual scrambling key by only two units in the least significant digit.

Figure 2

The scrambling and unscrambling keys in this case were:

```
Scrambling Key:   -9223372036854775808
Unscrambling Key: -9223372036854775806
```

Despite the similarity of the two keys, the bottom image in Figure 2 doesn't reveal much if anything about the original image.

The effect of color depth on the scrambling process

Figure 3 compares the scrambling and unscrambling of a starfish image in a JPEG file *(on the left)* with the scrambling and unscrambling of a GIF version of the same starfish image *(on the right). (Note that I purposely reduced the size of both pairs of images so that I could display them side-by-side in this narrow publication format.)*

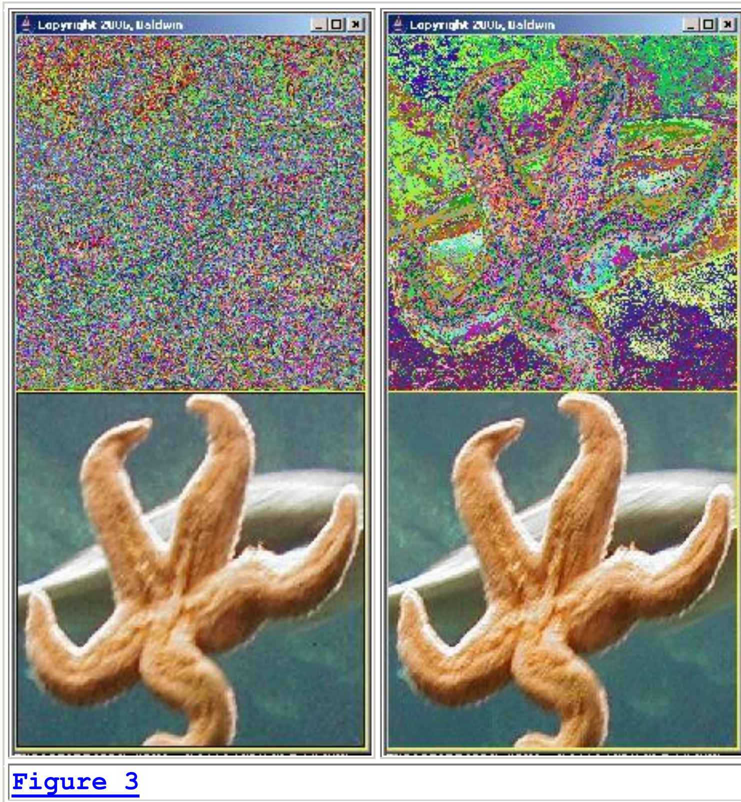

Figure 3

As you can see, the scrambled JPEG image on the left reveals very little about the original image. On the other hand, the scrambled GIF image on the right reveals quite a lot about the original image.

Even though every color in the original image was replaced by a different color *(chosen randomly)* in the scrambled image, because the number of colors in the GIF image was small, the scrambled GIF image is still recognizable as a starfish.

Large areas with pixels of the exactly same color

When the number of actual colors in the image is small, *(such as in a GIF image)*, there are many areas in the image where large numbers of adjacent pixels are exactly the same color. Simply changing the colors of the sets of same-colored adjacent pixels leaves patterns in the scrambled image that the human eye/mind combination is able to recognize. *(This speaks well of the pattern-recognition capability of the human eye/mind.)* The result of scrambling the GIF image *(with shallow color depth)*, shown in the top-right image of Figure 3, is simply a multi-colored starfish.

Very few areas with pixels of exactly the same color

On the other hand, when the number of actual colors in the image is large *(such as a digital photograph with more than sixteen-million colors)*, there are very few areas in the image where any significant number of adjacent pixels are exactly the same color. *(Adjacent pixels may be almost the same color but they are not exactly the same color.)*

Therefore, using a randomly-based substitution process to change each of the sixteen-million colors to a different color results in very few patterns that the human eye/mind combination is able to recognize, as illustrated by the top-left image in Figure 3.

In other words, because there are few if any groupings of colors in the scrambled image, the human eye/mind combination is simply overwhelmed in its attempts to recognize patterns. Thus, the top-left image in Figure 3 appears to be simply a bunch of randomly colored pixels. The important point, however, is that by reversing the color substitution process, the original image can be recovered as shown by the bottom-left image in Figure 3.

A caveat

Be aware, however, that just because an image is stored in a format that supports a large color depth, the image may actually have very shallow color depth. For example, if you were to use a digital camera to take a photograph of a black and white line drawing, the number of actual colors in the image would probably be quite small. A scrambled version of the resulting image would probably reveal quite a lot about the line drawing.

Artwork anyone?

Even if you aren't interested in scrambling images for security purposes, the technique can be used to produce some interesting artwork. Figure 4 shows the result of scrambling two versions of a color wheel. The original image on the left was a JPEG image, while the original image on the right was a GIF version of the JPEG image.

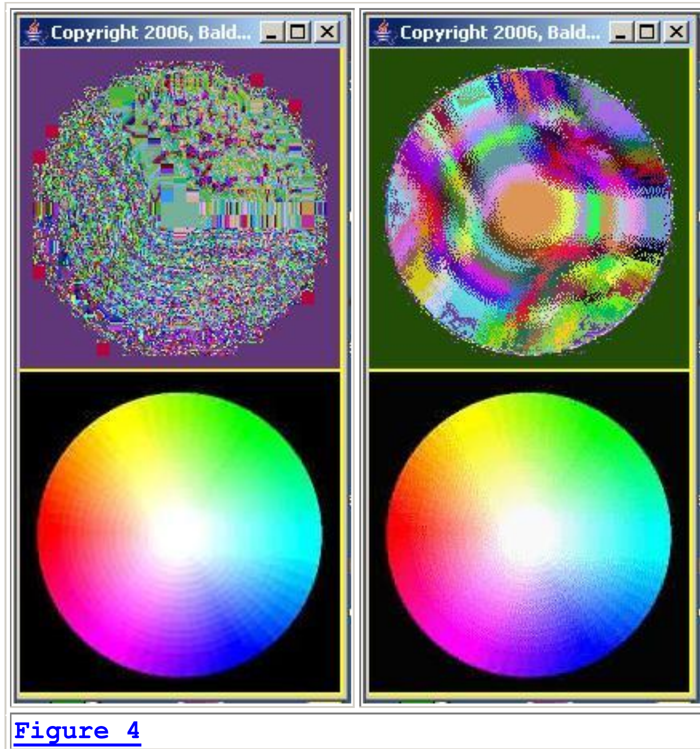

Figure 4

To the unpracticed eye, two different versions of the color wheel in the bottom of Figure 4 aren't terribly different. However, when scrambled in this manner, the top two images clearly show that the color wheel on the left contains a lot more actual colors than the color wheel on the right.

The downside - larger image files

Because this approach to scrambling and unscrambling images involves a simple color substitution process, it is absolutely necessary to preserve the scrambled image without modification. Otherwise, the image resulting from the unscrambling process won't match the original image. This eliminates the use of lossy compression schemes *(such as JPEG)* for the storage and exchange of scrambled images.

For this reason, the program named **ImgMod46a** produces an output PNG file containing the scrambled image. As discussed earlier, the PNG format is lossless. *(Other lossless image file formats could have been used as well.)*

An estimate.
I refer to this as an estimate because a compressed JPEG file is incapable of reproducing the original image. The uncompressed image is simply an estimate of the original image. Wikipedia estimates

The downside is that a PNG file is typically much larger than a JPEG file containing *(an estimate)* of the same image.

that a PNG file is likely to be five to ten times larger than a JPEG file containing the same original image.

Creating your own examples

The examples shown above were produced using a combination of all three programs in the above list of programs. To reduce the amount of typing required to run the three programs in sequence for every example, I created a batch file named **ImgMod46.bat** containing the commands shown in Figure 5.

```
echo off
echo Usage: enter ImgMod46 followed by a space
echo and the name of the image file.
java ImgMod46a -9223372036854775807 %1
java ImgMod46b -9223372036854775807
java ImgMod47 junk.png junk.jpg
```
Figure 5

Having created the batch file, all that was necessary for me to run an example was to enter a command such as the following at the command line:

`ImgMod46 colorwheel.jpg`

where **colorwheel.jpg** is the name of the image file to be processed.

If you are running Windows, you should be able to do the same thing. For other operating systems, you will either need to run the sequence of three programs directly from the command line, or create your own script.

Discussion and Sample Code

The Program Named ImgMod47

A complete listing of this program is provided in Listing 11 near the end of the lesson.

Description of the program

As mentioned earlier, I don't plan to explain the program code for this program in this lesson. However, I will provide an operational description.

This program reads and displays two image files one above the other for comparison purposes *(as shown in Figure 1)*. The images do not have to be the same size or to be of the same type. The program will read and display gif, jpg, bmp, and png files and possibly some other file types as well.

Typical usage

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

```
java ImgMod47 TopImageFileName BottomImageFileName
```

where the two command-line parameters specify the names of the image files to be displayed.

If the command-line parameters are omitted, the program will search for an image file named **ImgMod47Test.jpg** in the current directory and will display it in both the top and bottom display locations. This file must be provided in the current directory if it will be needed.

Display a single file.
To display a single file, just enter the name of the file twice on the command line. The file will be displayed in both the top and bottom locations.

The image files must be provided by the user but they don't have to be in the current directory if a path to the files is specified on the command line.

Display format

The two images are displayed in a frame with one image above the other. The program attempts to set the size of the frame so as to accommodate both images. However, if both images are not totally visible, the user can manually resize the frame in an attempt to make them both visible.

Error conditions

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

The program was tested using J2SE5.0 under WinXP. J2SE 5.0 or a later version is required due to the use of generics.

The Program Named ImgMod46a

Purpose

The purpose of this program is to scramble an image using a random number generator and to write the scrambled image into an output PNG file named **junk.png**.

User input

The seed value.
The seed value is the same as the shared key discussed earlier.

The value used to seed the random number generator and the name of the image file are specified by the user on the command line.

The program defaults to a fixed seed value and to an image file named **imgmod46test.jpg** if the user fails to specify both the seed value and the image file name on the command line.

A lossless output file format

A PNG file is used as the output file because it is necessary to avoid lossy compression in the output file. For example, if the output file were a JPEG file, *(which is a lossy compression scheme)*, it would not be possible to unscramble the image later.

Usage instructions

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

```
java ImgMod46a RandomSeedValue ImageFileName
```

The first parameter is a **long** value that is used to seed the random number generator. The same seed value must be used to scramble the image using this program and to later unscramble the image using the program named **ImgMod46b**.

As a **long** integer, the seed value can range:

From -9223372036854775808 to 9223372036854775807.

Input file types

The program can read and process jpg, gif, bmp, and png image files, and possibly some other file types as well.

Unscrambling the image file

Run the program named **ImgMod46b** to read the PNG file produced by this program and to unscramble the image that it contains.

Program testing

This program was tested using J2SE 5.0 under WinXP. J2SE 5.0 or later is required due to the use of generics.

Will discuss in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 12 near the end of the lesson.

Begin class definition

The class definition begins in Listing 1.

```
class ImgMod46a{
  BufferedImage rawBufferedImage;
```

```
  BufferedImage processedImage;
  static String defaultImgFile =
"imgmod46test.jpg";
  static String theImgFile = null;//Input image
file
  static long defaultSeed =
1234567890;//Default seed
  static long seed;
  MediaTracker tracker;
```

Listing 1

The code in Listing 1 simply declares some instance variables that are used later in the program.

The main method

The main method is shown in its entirety in Listing 2.

```
  public static void main(String[] args){

    //Get the seed and the input image file
name from the
    // command line, or use the default seed
and image
    // file name instead.
    if(args.length == 2){
      //Get the seed value.
      seed = Long.parseLong(args[0]);
      //Get the input file name
      theImgFile = args[1];
    }else{
      seed = defaultSeed;
      theImgFile = defaultImgFile;
    }//end else

    System.out.println("Scrambling Key: " +
seed);

    //Instantiate an object of this class.
    ImgMod46a obj = new ImgMod46a();
  }//end main
```

Listing 2

Listing 2 is straightforward and shouldn't require additional explanation. Note that the last statement in the **main** method invokes the constructor to instantiate an object of the class.

The constructor

The constructor is shown in its entirety in Listing 3.

```
  public ImgMod46a(){//constructor
    //Get an image from the specified image
file.
    rawBufferedImage = getTheImage();

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

    //Write the modified image into a file
named
    // junk.png.
    writeOutputFile(processedImage);

  }//end ImgMod46a constructor
```

Listing 3

The constructor is also straightforward. In invokes the following three methods in succession to perform the tasks shown:

- **getTheImage** - read the image from the input file into an object of type **BufferedImage**.
- **processImg** - scramble the image.
- **writeOutputFile** - write the scrambled image into an output file of type PNG named **junk.png**.

The method named getTheImage

This method is essentially the same as a method having the same name that I explained in the earlier lesson entitled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see References)*. Rather than to repeat that explanation here, I will refer you back to the earlier lesson.

The method named writeOutputFile

With the exception of the fact that this program writes a PNG file instead of a JPEG file, this method is essentially the same as the method named **writeJpegFile** that I explained in the same earlier lesson entitled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see References)*. Once again, rather than repeating that explanation here, I will refer you back to the earlier lesson.

That leaves us with the method named **processImg**.

The method named processImg

The method named **processImg** begins in Listing 4. This method uses the **LookupOp** class from the Java 2D API along with the **Random** class from the **java.util** package to scramble all of the color values in the pixels in the image identified by the parameter named **theImage**.

```
  public BufferedImage processImg(BufferedImage
theImage){
    Random randomGenerator = new Random(seed);

    ArrayList <Short>redList = new
ArrayList<Short>(256);
    ArrayList <Short>greenList = new
ArrayList<Short>(256);
    ArrayList <Short>blueList = new
ArrayList<Short>(256);
```

Listing 4

A random number generator and three ArrayList objects

The **processImg** method begins by instantiating a random number generator object of type **Random** using the seed that was provided as input by the user *(or the default seed if the user failed to provide a seed).*

Then the method creates three **ArrayList** objects, each containing 256 unique unsigned 8-bit values formatted into the least significant eight bits of 256 values of type **short**. The three empty **ArrayList** objects are instantiated in Listing 4. *(Note the use of generics in Listing 4, requiring the use of J2SE 5.0 or later.)*

Populate the ArrayList objects

Listing 5 shows the beginning of a **for** loop that is used to independently populate each of the three **ArrayList** objects with unique unsigned 8-bit values.

```
    for(int cnt = 0;cnt < 256;cnt++){
      //Get a priming value for the redList.
      short value =

(short)(randomGenerator.nextInt() & 0xFF);
      while(redList.contains(value)){
        //Try another value.  This one was
already used.
        value =
(short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      redList.add(value);//Add unique value to
the list.
```

Listing 5

Listing 5 contains code that obtains a value from the random number generator that is not contained in the **ArrayList** object referred to by **redList** and adds that value to the **ArrayList** object.

(If the value of the first random number is already contained in the ***ArrayList*** *object, it is discarded and replaced by another random value. This process is continued until a random value is found that is not already contained in the* ***ArrayList*** *object.)*

The comments in Listing 5 should be sufficient to more fully explain the operation of the code in that listing.

The green and blue ArrayList objects

Listing 6 shows the remainder of the **for** loop that began in Listing 5. The code in Listing 6 populates the green and blue **ArrayList** objects.

```
      //Get a priming value for the greenList.
      value = (short)(randomGenerator.nextInt()
& 0xFF);
      while(greenList.contains(value)){
        //Try another value.  This one was
already used.
        value =
(short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      greenList.add(value);//Add unique value
to the list.

      //Get a priming value for the blueList.
      value = (short)(randomGenerator.nextInt()
& 0xFF);
      while(blueList.contains(value)){
        //Try another value.  This one was
already used.
        value =
(short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      blueList.add(value);//Add unique value to
the list.

    }//end for loop
```

Listing 6

All three ArrayList objects are populated

When the **for** loop in Listing 6 terminates, *(after 256 iterations)*, each of the **ArrayList** objects have been independently populated with random sequences of unique unsigned 8-bit values formatted into the least significant eight bits of 256 values of type **short**.

These are the values that will be substituted for the red, green, and blue color values for each pixel in the image that is being scrambled.

Use of the LookupOp image-filtering class

I explained the use of the class named **LookupOp** for processing images in the method named **processImageForThePage** in the earlier lesson entitled "Using the Java 2D LookupOp Filter Class to Process Images" *(see References)*. I won't repeat that explanation in this lesson. Rather, I will simply refer you back to the earlier lesson for those details.

The ArrayList objects are a temporary convenience

What we really need to be able to use the **LookupOp** image-filtering class are three array objects of type **short** containing the substitution values. The **ArrayList** objects were simply used as a convenient way to create those values. Listing 7 creates the three array objects and populates them by copying the values from the **ArrayList** objects into the three array objects referred to as **red**, **green**, and **blue**.

```
    //Create the data for the lookup table.
    short[] red = new short[256];
    short[] green = new short[256];
    short[] blue = new short[256];

    for (int cnt = 0; cnt < 256; cnt++){
      red[cnt] = redList.get(cnt);
      green[cnt] = greenList.get(cnt);
      blue[cnt] = blueList.get(cnt);
    }//end for loop
```

Listing 7

Wrapping it up

The remaining program code is shown in Listing 8.

```
    //Create the 2D array that will be used to
create the
    // lookup table.
    short[][] lookupData = new
short[][]{red,green,blue};

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

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

    //Apply the filter to the incoming image
and return
    // a reference to the resulting
BufferedImage object.
    return filterObj.filter(theImage, null);
```

```
  }//end processImg
  //------------------------------------------
----------//
}//end ImgMod46a.java class
```

Listing 8

Same as the code in the earlier lesson

The code in Listing 8 is essentially the same as the code that I explained in the method named **processImageForThePage** in the earlier lesson entitled "Using the Java 2D LookupOp Filter Class to Process Images" *(see References)*. Please see that lesson for an explanation of the remaining code in Listing 8.

The end of the program

Listing 8 signals the end of the method named **processImg**, and also signals the end of the class named **ImgMod46a**. Therefore, this ends the explanation of the program named **ImgMod46a**.

The Program Named ImgMod46b

The program named **ImgMod46a** is very similar to the program named **ImgMod46a** that I explained above. If you haven't done so already, you should go back and study my explanation of that program.

Purpose of the program

The purpose of the program named **ImgMod46b** is:

- To unscramble the image in a PNG file that was scrambled and written by the program named **ImgMod46a**.
- To write the unscrambled image into an output JPEG file.

More specifically, this program:

- Reads a scrambled image from a file named **junk.png**.
- Unscrambles the image.
- Writes the unscrambled image into an output JPEG file named **junk.jpg**.

Display the unscrambled image

The program named **ImgMod47** can be used to display the scrambled image along with the unscrambled image for comparison purposes as shown in Figure 1.

The same seed must be used

A seed value is required to instantiate a random number generator object that is used to unscramble the image. The same seed value must be used to unscramble the image as was used to scramble the image using the program named **ImgMod46a**.

The seed value is specified by the user on the command line. The program defaults to the same fixed seed value as the default seed value used in the program named **ImgMod46a** if the user fails to specify the seed value on the command line.

Usage instructions

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

```
java ImgMod46b RandomSeedValue
```

The single parameter is a **long** integer that is used to seed the random number generator. As mentioned above, the same seed value must be used to unscramble the image as was used by the program named **ImgMod46a** to scramble the image.

The range of allowable seed values

The seed value that is specified by the user is the same as the shared key discussed earlier.

As a long integer, the value of the seed may range:

From -9223372036854775808 to 9223372036854775807.

This program was tested using J2SE 5.0 under WinXP. J2SE 5.0 or a later version is required due to the use of generics.

What's new in the program named ImgMod46b?

A complete listing of the program named **ImgMod46b** can be viewed in Listing 13 near the end of the lesson. This program is essentially the same as the program named **ImgMod46a** with two exceptions:

- This program writes a JPEG output file instead of a PNG output file.
- This program populates the filtering object differently.

Writing a JPEG file

Listing 9 shows the method named **writeOutputFile** in its entirety. This method writes the contents of a **BufferedImage** object into a file named **junk.jpg**.

```
void writeOutputFile(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 jpg format.
      ImageIO.write(img,"jpeg",outStream);
      outStream.close();
    }catch (Exception e) {
      e.printStackTrace();
    }//end catch
  }//end writeOutputFile
```

Listing 9

The code that is different from the code in the program named **ImgMod46a** is highlighted in boldface in Listing 9.

Populating the filtering object

Listing 10 shows the creation of three array objects along with a **for** loop that was extracted from the method named **processImg**. You can view this code in context in Listing 13. *(Compare the code in Listing 10 with the code in Listing 7.)*

```
    //Create the data for the lookup table.
    short[] red = new short[256];
    short[] green = new short[256];
    short[] blue = new short[256];

    for (int cnt = 0; cnt < 256; cnt++){
      red[cnt] =
           (short)(redList.indexOf(new
Short((short)cnt)));
      green[cnt] =
         (short)(greenList.indexOf(new
Short((short)cnt)));
      blue[cnt] =
          (short)(blueList.indexOf(new
Short((short)cnt)));
    }//end for loop
```

Listing 10

Listing 10 populates the **red**, **green**, and **blue** array objects with a set of substitution values that will reverse the substitution process implemented by the values used to populate the corresponding array objects in Listing 7.

A pencil and paper may help here

Unless your ability to analyze code in your mind is better than mine, you may need to use a pencil and paper and work through some of the values to understand the difference between the boldface expressions in Listing 10 and the corresponding boldface expressions in Listing 7.

A description of the unscrambling process

The code in Listing 7 simply copies the substitution values from the **ArrayList** objects into the array objects at the same index values.

The code in Listing 10 uses the substitution values in the **ArrayList** objects as indexes into the array objects and assigns the corresponding index values from the **ArrayList** objects as substitution values in the array objects.

As a result, when the **red**, **green**, and **blue** array objects in Listing 10 are used to filter an image that was previously filtered using the **red**, **green**, and **blue** array objects in Listing 7, the original substitution process is reversed and the scrambled image is unscrambled as shown in Figure 1.

And that completes the explanation of the unscrambling program named **ImgMod46b**.

Run the Program

I encourage you to copy the code from Listing 11, Listing 12, and Listing 13 into your text editor. Compile the code and execute it. Experiment with it, making changes, and observing the results of your changes.

If you are running under Windows, you may want to consider creating a batch file similar to that shown in Figure 5 to make it a little easier to execute the three programs in sequence. If you are running under a different operating system, you may want to consider creating a similar script for use with that operating system.

Summary

In this lesson, I taught you how to use the **LookupOp** image-filtering class from the Java 2D API, along with the **Random** class from the **java.util** package to write a pair of easy-to-use programs to scramble and unscramble images in a reasonably secure fashion.

References

- 400 Processing Image Pixels using Java, Getting Started
- 402 Processing Image Pixels using Java, Creating a Spotlight
- 404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
- 406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
- 408 Processing Image Pixels, Performing Convolution on Images
- 410 Processing Image Pixels, Understanding Image Convolution in Java
- 412 Processing Image Pixels, Applying Image Convolution in Java, Part 1

- 414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
- 416 Processing Image Pixels, An Improved Image-Processing Framework in Java
- 450 A Framework for Experimenting with Java 2D Image-Processing Filters
- 452 Using the Java 2D LookupOp Filter Class to Process Images

Complete Program Listing

Complete listings of the programs discussed in this lesson are shown in Listing 11, Listing 12, and Listing 13 below.

```
/*File ImgMod47.java
Copyright 2006, R.G.Baldwin

This program reads and displays two image files, one above
the other for comparison purposes.  The images do not have
to be the same size or the same type.

The program will read gif, jpg, and png files and possibly
some other input file types as well.

Typical usage is as follows:

java ImgMod47 TopImageFileName BottomImageFileName

If the command-line parameters are omitted, the program
will search for an image file in the current directory
named ImgMod47Test.jpg and will display it in both the
top and bottom display locations.  This file must be
provided in the current directory if it will be needed.

The image files must be provided by the user in all cases.
However, they don't have to be in the current directory if
a path to the files is specified on the command line.

The two images are displayed in a frame with one above the
other.  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.

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

Tested using J2SE5.0 under WinXP.  Requires J2SE 5.0 or
later version due to the use of generics.
**********************************************************/

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

```
class ImgMod47 extends Frame{
  //References to top and bottom images.
  BufferedImage topImage;
  BufferedImage bottomImage;

  Frame displayFrame;//Frame to display the images.
  int inLeft;//left inset
  int inTop;//top inset
  int inBottom;//bottom inset

  //This is the name of the default image file.  This image
  // file will be displayed in both the top and bottom
  // locations if the user fails to enter two command-line
  // parameters.  You must provide this file in the current
  // directory if it will be needed.
  static String topFile = "ImgMod47Test.jpg";
  static String bottomFile = "ImgMod47Test.jpg";

  MediaTracker tracker;
  Display display = new Display();//A Canvas object
  //-----------------------------------------------------//

  public static void main(String[] args){
    //Get input file names.  Program reads gif, png, and
    // jpg files and possibly some other file types as
    // well.
    if(args.length == 0){
      //Use default image file specified above.
    }else if(args.length == 2){
      topFile = args[0];
      bottomFile = args[1];
    }else{
      System.out.println("Invalid args");
      System.exit(1);
    }//end else

    //Display names of top and bottom image files.
    System.out.println("Top File: " + topFile);
    System.out.println("Bottom File: " + bottomFile);

    //Instantiate an object of this class.
    ImgMod47 obj = new ImgMod47();
  }//end main
  //-------------------------------------------//

  public ImgMod47(){//constructor
    //Get the top 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.
    topImage = getTheImage(topFile);

    //Get the bottom image from the specified image file.
    bottomImage = getTheImage(bottomFile);

    //Construct the display object.
    this.setTitle("Copyright 2006, Baldwin");
```

```
    this.setBackground(Color.YELLOW);
    this.add(display);

    //Make the frame visible so as to make it possible to
    // get insets.
    setVisible(true);
    //Get and store inset data for the Frame.
    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;
    inBottom = this.getInsets().bottom;
    setVisible(false);

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

    //Set the display size to accommodate the top and
    // bottom images. Set the size such that  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(bottomImage.getWidth() > topImage.getWidth()){
      maxWidth = bottomImage.getWidth();
    }else{
      maxWidth = topImage.getWidth();
    }//end else
    int totalWidth = 2*inLeft + maxWidth + 2;

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

    //Repaint the image display frame.
    display.repaint();

    //Cause the composite of the frame and the canvas 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 ImgMod47 constructor
  //=====================================================//

  //Inner class for canvas object on which to display the
  // two images.
  class Display extends Canvas{
    //Override the paint method to display two images 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((topImage != null) && (bottomImage != null)){

          //Draw the top image.  Terminate if the pixels
          // are changing.
          boolean success = false;
          success = g.drawImage(topImage,0,0,this);
          if(!success){
            System.out.println("Unable to draw top image");
            System.exit(1);
          }//end if

          //Draw the bottom image.
          success = g.drawImage(bottomImage,0,
                            topImage.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
  //=====================================================//

  //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 received as an incoming
  // parameter.
  BufferedImage getTheImage(String fileName){
    Image rawImage = Toolkit.getDefaultToolkit().
                                      getImage(fileName);

    //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.  This program
    // 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 ImgMod47.java class
//=====================================================//
```

Listing 11

Listing 12

```
/*File ImgMod46a.java
Copyright 2006, R.G.Baldwin

The purpose of this program is to scramble an image using
a random number generator and to write the scrambled image
into an output png file named junk.png.  The random seed
value and the name of the image file are specified by the
user on the command line.  The program defaults to a fixed
seed value and to an image file named imgmod46test.jpg if
the user fails to specify both the seed value and the image
file name on the command line.

A png file is used as the output file because it is
necessary to avoid lossy compression in the output file.
```

```
For example, if the output file were a JPEG file, which is
a lossy compression scheme, it would not be possible to
unscramble the image later.

Usage:  Enter the following at the command-line:

java ImgMod46a RandomSeedValue ImageFileName

The first parameter is a long value that is used to seed
the random number generator.  The same seed value must
be used to scramble and to unscramble the image.  The seed
is a long integer, which may range from
-9223372036854775808 to 9223372036854775807

Can read jpg, gif, and png image files, and possibly some
other file types as well.  Note, however, that because of
the relatively small number of actual colors in a gif
image, scrambling a gif image often results in a scrambled
image in which the shapes of the items in the image can
be easily seen.

Use the program named ImgMod46b to read the png file
produced by this program and to unscramble the image that
it contains.  The program named ImgMod46b writes the
unscrambled image into a JPEG output file named junk.jpg.

The program named ImgMod47 can be used to display the
scrambled image along with the unscrambled image for
comparison purposes.

Tested using J2SE 5.0 under WinXP.  A batch file containing
the following commands was used to test this program and
its companion programs named ImgMod46b and ImgMod47:

echo off
echo Usage: enter ImgMod46 followed by a space
echo and the name of the image file.
java ImgMod46a -9223372036854775808 %1
java ImgMod46b -9223372036854775808
java ImgMod47 junk.png junk.jpg
**********************************************************/

import java.awt.*;
import java.io.*;
import javax.imageio.*;
import java.awt.image.*;
import java.util.Random;
import java.util.ArrayList;

class ImgMod46a{
  BufferedImage rawBufferedImage;
  BufferedImage processedImage;
  static String defaultImgFile = "imgmod46test.jpg";
  static String theImgFile = null;//Input image file
  static long defaultSeed = 1234567890;//Default seed
  static long seed;
```

```
MediaTracker tracker;

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

public static void main(String[] args){

  //Get the seed and the input image file name from the
  // command line, or use the default seed and image
  // file name instead.
  if(args.length == 2){
    //Get the seed value.
    seed = Long.parseLong(args[0]);
    //Get the input file name
    theImgFile = args[1];
  }else{
    seed = defaultSeed;
    theImgFile = defaultImgFile;
  }//end else

  System.out.println("Scrambling Key: " + seed);

  //Instantiate an object of this class.
  ImgMod46a obj = new ImgMod46a();
}//end main
//---------------------------------------------//

public ImgMod46a(){//constructor
  //Get an image from the specified image file.
  rawBufferedImage = getTheImage();

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

  //Write the modified image into a file named
  // junk.png.
  writeOutputFile(processedImage);

}//end ImgMod46a constructor
//======================================================//

//Use the LookupOp class from the Java 2D API to
// scramble all of the color values in the pixels.  The
// alpha value is not modified.
public BufferedImage processImg(BufferedImage theImage){

  //Create three ArrayList objects, each containing 256
  // unique unsigned 8-bit values.  It is required that
  // the same seed value be used to scramble the image
  // and to unscramble the image.
  Random randomGenerator = new Random(seed);
  ArrayList <Short>redList = new ArrayList<Short>(256);
  ArrayList <Short>greenList = new ArrayList<Short>(256);
  ArrayList <Short>blueList = new ArrayList<Short>(256);

  for(int cnt = 0;cnt < 256;cnt++){
    //Get a priming value for the redList.
```

```
      short value =
                 (short)(randomGenerator.nextInt() & 0xFF);
      while(redList.contains(value)){
        //Try another value.  This one was already used.
        value = (short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      redList.add(value);//Add unique value to the list.

      //Get a priming value for the greenList.
      value = (short)(randomGenerator.nextInt() & 0xFF);
      while(greenList.contains(value)){
        //Try another value.  This one was already used.
        value = (short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      greenList.add(value);//Add unique value to the list.

      //Get a priming value for the blueList.
      value = (short)(randomGenerator.nextInt() & 0xFF);
      while(blueList.contains(value)){
        //Try another value.  This one was already used.
        value = (short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      blueList.add(value);//Add unique value to the list.

    }//end for loop

    //Create the data for the lookup table.
    short[] red = new short[256];
    short[] green = new short[256];
    short[] blue = new short[256];

    for (int cnt = 0; cnt < 256; cnt++){
      red[cnt] = redList.get(cnt);
      green[cnt] = greenList.get(cnt);
      blue[cnt] = blueList.get(cnt);
    }//end for loop

    //Create the 2D array that will be used to create the
    // lookup table.
    short[][] lookupData = new short[][]{red,green,blue};

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

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

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

  }//end processImg
  //=========================================================//
```

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

//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 ImgMod46a.java class
//=======================================================//
```

Listing 12

Listing 13

```
/*File ImgMod46b.java
Copyright 2006, R.G.Baldwin

See comments in the companion program named ImgMod46a.

The purpose of this program is to unscramble the image in
a png file that was scrambled and written by the program
named ImgMod46a, and to write the unscrambled image into an
output JPEG file.

This program reads a scrambled image from a file named
junk.png, unscrambles the image, and writes the unscrambled
image into an output JPEG file named junk.jpg.

The program named ImgMod47 can be used to display the
scrambled image along with the unscrambled image for
comparison purposes.

A random seed value is required to unscramble the image.
The same seed value must be used to unscramble the image
as was used to scramble the image.  The seed value is
specified by the user on the command line.  The program
```

```
defaults to the same fixed seed value as the default seed
value used in the program named ImgMod46a if the user fails
to specify the seed value on the command line.

Usage:  Enter the following at the command-line:

java ImgMod46b RandomSeedValue

The parameter is a long integer that is used to seed the
random number generator.  The same seed value must be used
to unscramble the image as was used by the program named
ImgMod46a to scramble the image.  As a long integer, the
value of the seed may range from -9223372036854775808
to 9223372036854775807

Tested using J2SE 5.0 under WinXP.  A batch file containing
the following commands was used to test this program and
its companion programs named ImgMod46a and ImgMod47:

echo off
echo Usage: enter ImgMod46 followed by a space
echo and the name of the image file.
java ImgMod46a -9223372036854775808 %1
java ImgMod46b -9223372036854775808
java ImgMod47 junk.png junk.jpg
**********************************************************/

import java.awt.*;
import java.io.*;
import javax.imageio.*;
import java.awt.image.*;
import java.util.Random;
import java.util.ArrayList;

class ImgMod46b{
  BufferedImage rawBufferedImage;
  BufferedImage processedImage;
  static String theImgFile = "junk.png";
  static long defaultSeed = 1234567890;//Default seed
  static long seed;
  MediaTracker tracker;

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

  public static void main(String[] args){
    //Get the seed value from the command line, or use the
    // default seed value instead.
    if(args.length == 1){
      //Get the seed value.
      seed = Long.parseLong(args[0]);
    }else{
      seed = defaultSeed;
    }//end else

    System.out.println("Unscrambling Key: " + seed);
```

```
    //Instantiate an object of this class.
    ImgMod46b obj = new ImgMod46b();
  }//end main
  //-------------------------------------------//

  public ImgMod46b(){//constructor
    //Get an image from the specified image file.
    rawBufferedImage = getTheImage();

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

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

  }//end ImgMod46b constructor
  //===================================================//

  //Use the LookupOp class from the Java 2D API to
  // unscramble the color values in the pixels.  The
  // alpha value is not modified.
  public BufferedImage processImg(BufferedImage theImage){

    //Create three ArrayList objects, each containing 256
    // unique unsigned 8-bit values.  It is required that
    // the same seed value be used to unscramble the image
    // as was originally used to scramble the image.
    Random randomGenerator = new Random(seed);
    ArrayList <Short>redList = new ArrayList<Short>(256);
    ArrayList <Short>greenList = new ArrayList<Short>(256);
    ArrayList <Short>blueList = new ArrayList<Short>(256);

    for(int cnt = 0;cnt < 256;cnt++){
      //Get a priming value for the red list.
      short value =
                 (short)(randomGenerator.nextInt() & 0xFF);
      while(redList.contains(value)){
        //This value was used earlier. Try another value.
        value = (short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      redList.add(value);//Add unique value to the list.

      //Get a priming value for the green list.
      value = (short)(randomGenerator.nextInt() & 0xFF);
      while(greenList.contains(value)){
        //Try another value
        value = (short)(randomGenerator.nextInt() & 0xFF);
      }//end while
      greenList.add(value);//Add unique value to the list.

      //Get a priming value for the blue list.
      value = (short)(randomGenerator.nextInt() & 0xFF);
      while(blueList.contains(value)){
        //Try another value
        value = (short)(randomGenerator.nextInt() & 0xFF);
```

```
    }//end while
    blueList.add(value);//Add unique value to the list.

  }//end for loop

  //Create the data for the lookup table.
  short[] red = new short[256];
  short[] green = new short[256];
  short[] blue = new short[256];
  for (int cnt = 0; cnt < 256; cnt++){
    red[cnt] =
         (short)(redList.indexOf(new Short((short)cnt)));
    green[cnt] =
       (short)(greenList.indexOf(new Short((short)cnt)));
    blue[cnt] =
        (short)(blueList.indexOf(new Short((short)cnt)));
  }//end for loop

  //Create the 2D array that will be used to create the
  // lookup table.
  short[][] lookupData = new short[][]{red,green,blue};

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

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

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

}//end processImg
//=====================================================//

//Write the contents of a BufferedImage object to a
// file named junk.jpg.
void writeOutputFile(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 jpg format.
    ImageIO.write(img,"jpeg",outStream);
    outStream.close();
  }catch (Exception e) {
    e.printStackTrace();
  }//end catch
}//end writeOutputFile
//-----------------------------------------------------//

//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 ImgMod46b.java class
//====================================================//
```

Listing 13

Copyright 2007, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

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 filter LookupOp

-end-