Processing Image Pixels, An Improved Image-Processing Framework in Java

Learn about a new image processing framework that provides the ability to perform two sequential processes on an image, to display the modified images resulting from each process, (in addition to the original image), and to write the modified images into output files in JPEG format.

Published: October 31, 2006
By Richard G. Baldwin

Java Programming Notes # 416

Preface

In this lesson, you will Learn about a new image processing framework that provides the ability to perform two sequential processes on an image, to display the modified image resulting from each process, *(in addition to the original image),* and to write the modified images into output files in JPEG format.

The earlier programs named ImgMod02 and ImgMod02a

In the Fall of 2004, I published a tutorial lesson entitled Processing Image Pixels using Java, Getting Started, which contained a program named **ImgMod02**. That program was designed to make it *"easy to experiment with the modification of pixel data in an image and to display the modified version of the image along with the original version of the image."*

A couple of months later in the lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness, I published an updated version of the program. The updated version was named **ImgMod02a**. The purpose of that update was rather minor. Basically it improved the code that reads an input image file.

Many subsequent lessons

Since then, I have published about a dozen tutorial lessons on various image processing and digital signal processing *(DSP)* topics that make use of one or the other of those two programs. During the publication of those lessons, I have identified several features that I wished I had included in the original version. Therefore, it is time for a major new update.

The new program named ImgMod04

This lesson will present a new program named **ImgMod04** that incorporates the following new features in the framework:

- The ability to perform two sequential pixel-modification processes *(instead of just one)* on an image and to display the modified image resulting from each process in addition to displaying the original image.
- The ability to write two modified images into output files in JPEG format. *(The names of the output files are **junk1.jpg** and **junk2.jpg**. They are written into the current directory.)*
- The elimination of several conversions back and forth between type **double** and type **int**. All computations can now be performed as type **double**, and the data is maintained as type **double** from the initial conversion from **int** to **double** to the point where the data is ready to be displayed and written into a JPEG file.
- The **Replot** button *(see Figure 1)* was moved to the top of the display *(see Figure 2)* to make it accessible when the display is too long to fit on the screen. *(For purposes of viewing the display in that case, the frame can be moved up and down on the screen using the right mouse button and the up and down arrow keys, but the lower portion of the bottom display still gets chopped off.)*

Differences between the two programs

The most obvious differences between the two programs are illustrated by differences between Figure 1 and Figure 2. Figure 1 shows an output from the earlier program named **ImgMod02a**. The two images in Figure 1 were first published in Figure 57 in the earlier lesson entitled Processing Image Pixels, Applying Image Convolution in Java.

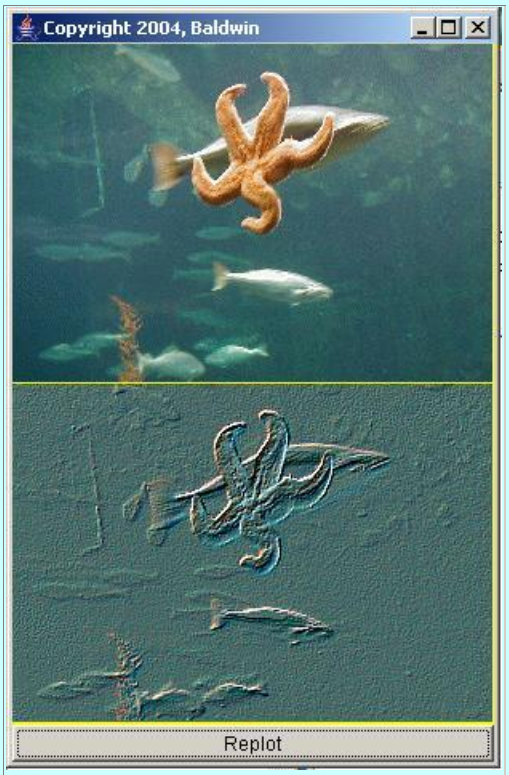

Figure 1

An embossing filter

If you have studied the earlier lesson, you are aware that the bottom image in Figure 1 was produced by applying a two-dimensional embossing convolution filter to the image of the starfish in the top of Figure 1.

A hidden watermark example

Figure 2 shows two output displays from the new program named **ImgMod04** arranged side-by-side for comparison purposes. The primary display for **ImgMod04** contains three images plus a **Replot** button in a vertical stack in a Java **Frame** object.

The six images contained in the two displays shown in Figure 2 *(along with some other images as well)* will be published later in a future lesson on creating hidden watermarks.

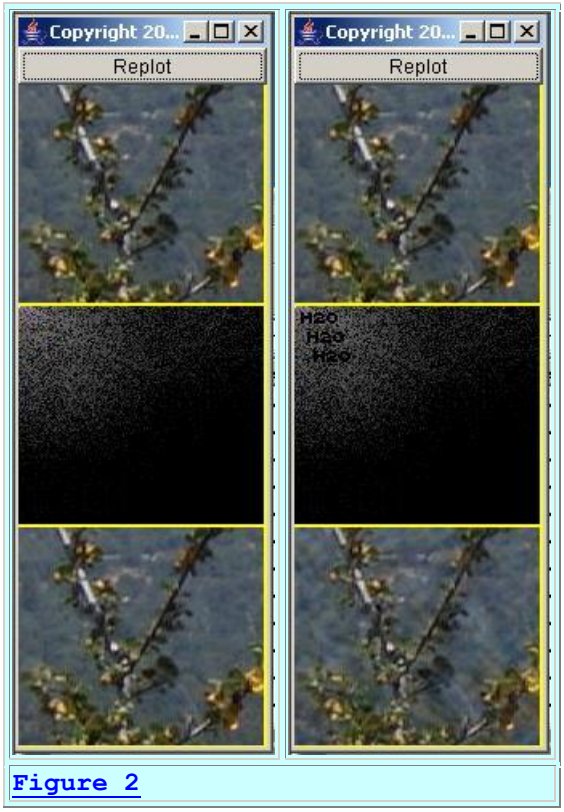

Figure 2

Brief description of Figure 2

Briefly, the two input images shown in the top of the left and right panels in Figure 2 are the same. The middle image in the left panel shows the result of performing a two-dimensional Discrete Cosine Transform *(2D-DCT)* on the input image. *(I explained the 2D-DCT in the earlier lesson entitled Understanding the 2D Discrete Cosine Transform in Java.)*

Recovering the visual image

The bottom image in the left panel of Figure 2 shows the result of performing an inverse 2D-DCT on the middle image for the purpose of recovering the visual image from the spectral data. Ideally, this image should be an exact match for the top image.

Adding watermarks to the spectral data

The middle image in the right panel of Figure 2 shows the result of performing a 2D-DCT on the input image and then adding three instances of a watermark consisting of the characters **H2O**.

> *(You should be able to see the watermarks in the upper-left corner of the middle image in the right panel of Figure 2. Obviously, they are not hidden in the spectral data, but they are hidden in the recovered visual image discussed below.)*

The visual image with the hidden watermarks

The bottom image in the right panel of Figure 2 shows the result of performing an inverse 2D-DCT on the middle image for the purpose of recovering the visual image from the spectral data that includes the watermark. Ideally, the bottom image should be an exact match for the top image. Also, ideally, the bottom image on the right should be an exact match for the bottom image on the left. A comparison of the two bottom images indicates the visual impact, if any, that the addition of the hidden watermarks had on the visual image.

I will explain digital watermarking in some detail in a future lesson. In that lesson, I will show that the hidden watermark can be exposed *(for the purpose of attempting to establish ownership of the image)* by performing a forward 2D-DCT on the image containing the hidden watermark in the bottom-right panel. *(I may perform some additional DSP processes in addition to the 2D-DCT.)* I will use the JPEG file produced by the program to show that.

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.

A new class named ImgMod04a

While I was at it, I decided to go ahead and provide a class named **ImgMod04a** that contains all of the features of **ImgMod04** except that it supports only one image processing method.

The most notable thing about this new class is that it replaces the earlier class named **ImgMod02a** with a class that writes the modified image into an output file in JPEG format. The name of the output file is **junk.jpg**.

I will have more to say about this class later.

Preview

The program named ImgMod04

A complete listing of this program is provided in Listing 28 near the end of the lesson. The purpose of this program is to make it easy to experiment with the modification of pixel data in an image and to display two different modified versions of the image along with the original image.

Program updates

This program is an update of the earlier program named **ImgMod02a**. This update provides the following new features:

- The ability to perform two sequential pixel-modification processes *(instead of just one)* on an image and to display the modified image resulting from each process in addition to displaying the original image.
- The ability to write two modified images into output files in JPEG format. *(The names of the output files are **junk1.jpg** and **junk2.jpg**. They are written into the current directory.)*
- The elimination of several conversions back and forth between type **double** and type **int**. All computations can now be performed as type **double**, and the data is maintained as type **double** from the initial conversion from **int** to **double** to the point where the data is ready to be displayed and written into a JPEG file.
- The **Replot** button *(see Figure 1)* was moved to the top of the display *(see Figure 2)* to make it accessible when the display is too long to fit on the screen. *(For purposes of viewing the display in that case, the frame can be moved up and down on the screen using the right mouse button and the up and down arrow keys, but the lower portion of the bottom display still gets chopped off.)*

Input from an image file

The program reads the pixel data from an image file into a 3D array of type:

```
double[row][column][depth]
```

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

Contents of the 3D array

The first two dimensions of the array correspond to the rows and columns of pixels in the image. The third dimension always has a value of 4 and contains the following values by index:

- 0 - alpha *(transparency)*
- 1 - red
- 2 - green
- 3 - blue

Note that these values are stored as type **double** rather than as type *unsigned byte* which is the standard format of pixel data in the original image file. This type conversion eliminates many problems involving the requirement to perform unsigned arithmetic on unsigned byte data.

(Recall that Java doesn't support unsigned numeric data or unsigned arithmetic.)

Image file types

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

Operation

This program provides a framework that is designed to invoke another program to process the pixels extracted from an image. In other words, this program extracts the pixels and puts them in a format that is relatively easy to work with. A second program is invoked by the program named **ImgMod04** to actually process the pixels.

Thus, this program performs the following major tasks:

- Reads the original image data from an input file.
- Provides the image data to a pair of processing methods initially and again each time the **Replot** button is clicked.
- Displays the data returned by the two image processing methods along with the original image data.
- Writes the image data that is displayed into output files in JPEG format.

Typical usage is as follows:

```
java ImgMod04 ProcessingProgramName ImageFileName
```

A built-in test program

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

If the **ImageFileName** is not specified on the command line as shown above, the program will search for an image file in the current directory named **ImgMod04Test.jpg** and will process it using the processing program specified by **ProcessingProgramName**.

If both command-line arguments are omitted, the program will search for an image file in the current directory named **ImgMod04Test.jpg** and will process it using the built-in processing program named **ProgramTest**.

I will provide a complete description of ProgramTest later in this lesson.

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 on the command line.

Behavior at startup

When the program is started, the original image and two processed versions of the image are displayed in a frame with the original image above the two processed images as shown in either the left or right panel of Figure 2.

Behavior of the Replot button

A **Replot** button appears at the top of the frame. If the user clicks the **Replot** button, the two image processing methods are re-run using the original image as input to the first processing program, and using one of the array objects output from the first processing program as input to the second processing program.

The images are reprocessed and the newly processed versions of the images replace the old versions in the display.

A GUI for user input

The processing program may provide a GUI for user input making it possible for the user to modify the behavior of the image processing methods each time they are run. This capability is illustrated by the built-in processing program named **ProgramTest**.

(The user input GUI for ProgramTest is shown in Figure 3.)

The ImgIntfc04 interface

The image processing program must implement the interface named **ImgIntfc04**. A complete listing of the interface is provided in Listing 29. That interface declares two image processing methods with the following signatures:

```
ImgIntfc04Method01Output processImg01(double[][][] input);
double[][][] processImg02(double[][][] input);
```

The image processing method named processImg01

The first image processing method named **processImg01** must return references to two3D array objects of type **double** encapsulated in an object of the class **ImgIntfc04Method01Output**. *(This is simply a wrapper class designed to encapsulate the two references in a single returned object.)* The definition for the **ImgIntfc04Method01Output** class is also provided in Listing 29.

The contents of the array referred to by one of the returned references are displayed as the middle image of the main display. *(See Figure 2 for two examples of the main display.)* This image data is also written into an output file in JPEG format named **junk1.jpg**.

The other reference returned from the method named **processImg01** is passed as a parameter to the second processing method named **processImg02**.

Depending on the processing objectives, the two references may point to the same or to different array objects. In Figure 2, the two references pointed to different array objects. For the output from the built-in program named **ProgramTest**, shown in Figure 4, the two references point to the same array object.

The image processing method named processImg02

The second image processing method that is declared in the interface named **ImgIntfc04** must return a reference to a single 3D array object of type **double**. The contents of this array object are displayed in the bottom image of the main display. *(Once again, see Figure 2 for two examples of the main display.)* The contents of the array object returned by the method named **processImg02** are also written into an output file in JPEG format named **junk2.jpg**.

No parameterized constructor

The image processing program cannot use a parameterized constructor. This is because an object of the class is instantiated by invoking the **newInstance** method of the class named **Class** on the name of the image processing program provided as a **String** on the command line. This approach to object instantiation does not support parameterized constructors.

No main method needed

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

Programming guidelines

The image processing methods named **processImg01** and **processImg02** each receive a 3D array containing pixel data as type **double**. Each method should probably make a copy of the incoming array and modify the copy rather than modifying the original. Then the programs should return references to the modified copies of the 3D pixel arrays.

The two image processing methods are free to modify the values in the incoming arrays in any manner whatsoever before returning references to the modified arrays. Note however that native pixel data consists of four unsigned bytes. If the modification of the data produces negative values or positive values greater than 255, this should be dealt with before returning the modified data that is to be displayed and written into the output JPEG files. Otherwise, the results of displaying the modified data may not be as expected.

Two ways to deal with the issue

There are at least two ways to deal with this situation. One way is to simply clamp all negative values at zero and to clamp all values greater than 255 at 255. The other way is to perform a further modification of the data so as to map the range from -x to +y into the range from 0 to 255. This amounts to changing the color distribution for the image, which may or may not be appropriate in a specific situation. The previous lesson entitled Processing Image Pixels Using

Java: Controlling Contrast and Brightness explains many aspects of dealing with the color distribution of an image.

There is no single correct way to deal with the issue for all situations.

The built-in test class named ProgramTest

The purpose of this class is to provide a simple example of an image processing program that is compatible with the class named **ImgMod04** and the interface named **ImgIntfc04**.

The constructor

The constructor for the class displays a small frame on the screen with a single text field as shown in Figure 3.

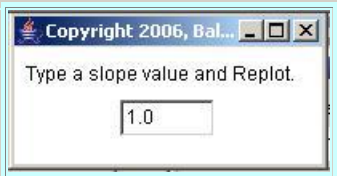

Figure 3

The purpose of the text field is to allow the user to enter a value that represents the slope of a line.

Enter a slope and click Replot

In operation, the user types a new value into the text field and then clicks the **Replot** button shown at the top of the main image display frame in either the left panel or the right panel of Figure 4. This causes the slope of the line shown in the middle and bottom images to match the value in the text field and causes the images to be repainted.

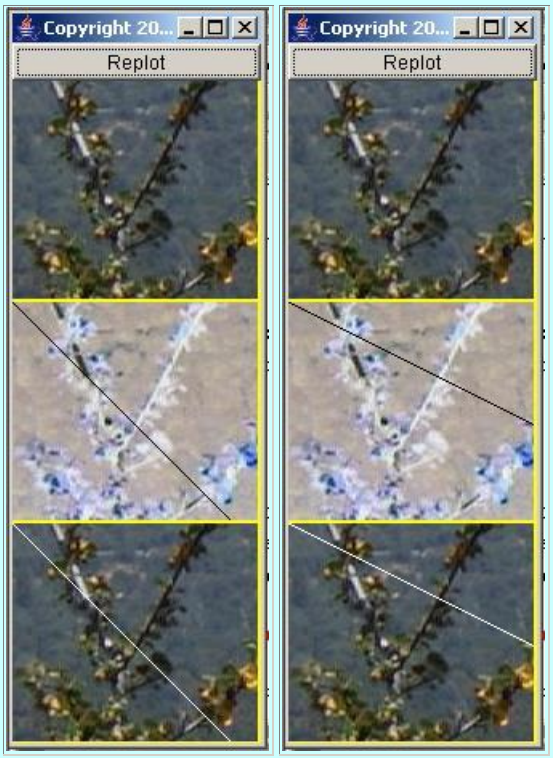

Figure 4

The user is not required to press the Enter key after typing the new value, but it doesn't do any harm to do so.

Negative slopes are not supported. An attempt to use a negative slope will cause the program to abort with an error.

The method named processImage01

The method named **processImage01** receives a 3D array of type **double** containing alpha, red, green, and blue values for the image shown at the top of either panel in Figure 4.

Draw a line on the image

The image that is received by the **processImg01** method is modified by the method to cause a white diagonal line to be drawn down and to the right from the upper left corner of the image. The slope of the line is controlled by the value in the text field.

> *(Because the positive vertical direction is down the screen instead of up, the actual slope of the line that is drawn is the negative of what we typically think of as the slope of a line.)*

Initially, the value in the text field is 1.0 as shown in Figure 3, but the value can be modified by the user.

*(If the characters in the text field cannot be converted to a numeric type **double**, the program will abort with an error.)*

The line shown in the middle and bottom images of the left panel of Figure 4 has the initial slope of 1.0. The lines shown in the images in the right panel have a user-specified slope of 0.5.

Invert the colors in the image

After drawing the line, the **processImg01** method inverts the colors in the image. This results in a black line on an image with inverted colors as shown by the middle images in the two panels of Figure 4.

Return references to two image arrays

The method named **processImg01** is required to return references to two 3D array objects of type **double**. The two references are encapsulated in an object of type **ImgIntfc04Method01Output**.

The version of the **processImg01** method in the built-in class named **ProgramTest** returns two references to the same modified pixel array, but that is not necessary. The method can return references to two different 3D arrays of type **double** as is the case for the hidden watermark example shown in Figure 2.

Data to be displayed

The data in the 3D array that is to be displayed as the middle image is referenced by the value stored in the variable named **objectA** of the returned object of type **ImgIntfc04Method01Output**. This data is displayed as the middle image by the code in the class named **ImgMod04**.

Data to be passed to the processImg02 method

The reference stored in the variable named **objectB** of the returned object is passed as a parameter to the method named **processImg02** by the code in the class named **ImgMod04**.

As mentioned above, in this version of the method, both references point to the same 3D array object, so the same reference is used to produce the middle image and also to be passed to the method named **processImg02**.

The method named processImage02

The method named **processImg02** receives an incoming 3D array of type **double** and returns one reference to a 3D array of type **double** containing pixel data.

*(There is no requirement that the incoming array to the **processImage02** method contain pixel data. It can be any kind of data so long as it is properly stored in a*

*3D array of type **double**. In fact, for the hidden watermark case shown in Figure 2, the incoming data to the **processImage02** method isn't pixel data at all. Rather, it is data that describes the 2D wave-number spectrum of the image at the top of Figure 2 obtained by performing a 2D-DCT on the image. Converting that spectral data of type **double** to eight-bit pixel data before passing it to the method would seriously degrade the quality of the spectral data.)*

Perform a color inversion

This version of the **processImg02** method simply performs a color inversion on the incoming data and returns a reference to the array of modified pixel data. This effectively reverses the color inversion performed earlier by the **processImg01** method, causing the bottom images in Figure 4 to match the top images except that they have a white diagonal line drawn on them.

Drawing a new line

To cause a new line to be drawn, type a new slope value into the text field and click the **Replot** button at the top of the main image display frame. As mentioned above, the images in the right panel of Figure 4 were produced with a user input slope of 0.5.

Must implement ImgIntfc04

The **ProgramTest** class extends **Frame** and implements **ImgIntfc04**. A compatible class is not required to extend the **Frame** class but it is required to implement the **ImgIntfc04** interface. This example extends **Frame** because it provides a GUI for user input.

Discussion and Sample Code

The Class Named ImgMod04

As is my custom, I will discuss and explain the code in fragments. Listing 1 shows the beginning of the class named **ImgMod04**.

```
class ImgMod04 extends Frame{
  Image rawImg;//A reference to the raw image.
  int imgCols;//Number of horizontal pixels
  int imgRows;//Number of rows of pixels
  Image modImgA;//Reference to first modified
image
  Image modImgC;//Reference to third modified
image

  //Default image processing program.  This
class will be
  // executed to process the image if the name
of another
  // class is not entered on the command line.
Note that
```

```
  // the source code for this class file is
included in
  // this source code file.
  static String theProcessingClass =
"ProgramTest";

  //Default image file name.  This image file
will be
  // processed if another file name is not
entered on the
  // command line.  You must provide this file
in the
  // current directory if it is going to be
processed.
  static String theImgFile =
"ImgMod04Test.jpg";

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

  //Reference to the image processing object.
  ImgIntfc04 imageProcessingObject;
```

Listing 1

The code in Listing 1 is straightforward and shouldn't require any explanation beyond the comments embedded in the code.

The main method

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

```
  public static void main(String[] args){
    //Get names for the image processing class
and the
    // image file to be processed.  Program
supports 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 == 1){
      theProcessingClass = args[0];
      //Use default image file along with this
class.
    }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.
    ImgMod04 obj = new ImgMod04();
  }//end main

Listing 2
```

The comments in Listing 2 should suffice to explain everything about the **main** method.

The constructor

The constructor begins in Listing 3.

```
  public ImgMod04(){//constructor
    final double[][][] threeDPix =
getTheImage();

Listing 3
```

The constructor begins by invoking the method named **getTheImage** to get an image from the specified image file. The image file can be in a different directory if the path was entered with the file name on the command line.

The local variable named **threeDPix** must be declared **final** because it is accessed from within an anonymous inner class. Once the reference is returned from the method named **getTheImage** and assigned to the variable, the contents of the variable cannot be changed for the duration of the program.

At this point, I will set the constructor aside discuss the method named **getTheImage**.

The method named getTheImage

The method named **getTheImage** reads an image from a specified image file and converts the pixel data into a 3D array of type **double**. The name of the image file is specified by the contents of the **String** variable named **theImgFile** in Listing 1. The method named **getTheImage** is shown in its entirety in Listing 4.

```
  double[][][] getTheImage(){
    rawImg = Toolkit.getDefaultToolkit().

getImage(theImgFile);

    //Use a MediaTracker object to block until
the image is
    // loaded or ten seconds has elapsed.
    tracker = new MediaTracker(this);
    tracker.addImage(rawImg,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

    //Raw image has been loaded.  Get width and
height of
    // the raw image.
    imgCols = rawImg.getWidth(this);
    imgRows = rawImg.getHeight(this);

    //Create a 1D array object to receive the
pixel
    // representation of the image
    int[] oneDPix = new int[imgCols * imgRows];

    //Create an empty BufferedImage object
    BufferedImage buffImage = new
BufferedImage(
                            imgCols,
                            imgRows,

BufferedImage.TYPE_INT_ARGB);

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

```
    //Convert the BufferedImage to numeric
pixel
    // representation.
    DataBufferInt dataBufferInt =

(DataBufferInt)buffImage.getRaster().

getDataBuffer();
    oneDPix = dataBufferInt.getData();

    //Convert the pixel byte data in the 1D
array to
    // double data in a 3D array to make it
easier to work
    // with the pixel data later.  Recall that
pixel data
    // is unsigned byte data and Java does not
support
    // unsigned arithmetic. Performing unsigned
arithmetic
    // on byte data is particularly cumbersome.
    return
convertTo3D(oneDPix,imgCols,imgRows);
  }//end getTheImage
```

Listing 4

I explained code very similar to the code in Listing 4 in the earlier lesson entitled The AWT Package, Graphics - Overview of Advanced Image Processing Capabilities so I won't repeat that explanation here.

Convert to 3D double data

Listing 4 invokes the **convertTo3D** method to convert the pixel byte data in the 1D array of type **int** to a 3D array of type **double**. This was done to make it easier to work with the pixel data later.

> *(Recall that pixel data is unsigned byte data and Java does not support unsigned arithmetic. Therefore, performing unsigned arithmetic on byte data in Java is particularly cumbersome.)*

You can view the method named **convertTo3D** in Listing 28. If you understand the use of bitwise operators, the code in the **convertTo3D** method shouldn't require further explanation. If not, you can learn a little about bitwise operators in the earlier lesson entitled Operators.

Construct the display object

Returning now to the discussion of the constructor, the code in Listing 5 performs several routine operations necessary to construct the display objects shown earlier in the panels of Figure 4.

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

Listing 5
```

Set the frame size

Listing 6 sets the frame size so that a small amount of yellow background will show on the right, between the images, and on the bottom when all three images are displayed, one above the other as shown by the images in the panels in Figure 4.

```
    //Make the frame visible so as 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.
    int inTop = this.getInsets().top;
    int inLeft = this.getInsets().left;
    int buttonHeight =
replotButton.getSize().height;

    this.setSize(2*inLeft+imgCols + 2,inTop
                           + buttonHeight +
3*imgRows + 9);

Listing 6
```

Note that it is necessary to make the frame visible before it is possible to get valid values for the insets and the height of the button.

Register ActionListener on the Replot button

Listing 7 begins the definition and instantiation of an anonymous inner class listener that is registered on the **Replot** button.

```
    replotButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent
e){
          ImgIntfc04Method01Output output =

imageProcessingObject.processImg01(threeDPix);

Listing 7
```

(If you don't know about anonymous inner classes, see my earlier lesson entitled The Essence of OOP using Java, Anonymous Classes.)

The actionPerformed method

The **actionPerformed** method that begins in Listing 7 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.

Listing 7 passes a 3D array of pixel data to the first processing method named **processImg01**. This method returns references to two 3D arrays encapsulated in the variables named **outputA** and **outputB** in an object of type **ImgIntfc04Method01Output**.

The contents of the array referred to by **outputA** will be displayed as the middle image in the displays shown in Figure 2 and Figure 4. The reference contained in **outputB** will be passed as a parameter to the second processing method named **processImg02**. The contents of the array referred to by **outputA** will also be written into an output JPEG file named **junk1.jpg**.

The second processing method named **processImg02** returns a reference to a single 3D array of pixel data as type **double**. The contents of that array will be displayed as the bottom image in the displays shown in Figure 2 and Figure 4. The contents of that array will also be written into an output JPEG file named **junk2.jpg**.

Extract the array object references

Listing 8 extracts the references to the two 3D array objects of type **double** from the returned object of type **ImgIntfc04Method01Output**.

```
          double[][][] threeDPixModA =
output.outputA;
          double[][][] threeDPixModB =
output.outputB;

Listing 8
```

Pass object's reference to second processing method

Listing 9 passes the second array object's reference returned by the first processing method to the second processing method named **processImg02**.

```
          double[][][] threeDPixModC =

imageProcessingObject.processImg02(

threeDPixModB);

Listing 9
```

Listing 9 also saves the reference to an array object returned by the second processing method named **processImg02**.

Prepare the data for display

At this point, the time has come to display the contents of two 3D array objects as the middle and bottom images in one of the panels shown in Figure 2 and Figure 4. However, the data is not in a form suitable for display at this point in the program, so the data must be prepared to be displayed.

Listing 10 invokes the method named **convertTo1D** to convert one 3D array of modified pixel data of type **double** to a 1D array of pixel data of type **int**. This 1D array is in a standard pixel format.

```
          int[] oneDPixA =
convertTo1D(threeDPixModA);

Listing 10
```

The method named convertTo1D

At this point, I will set the discussion of the **actionPerformed** method aside and discuss the method named **convertTo1D**, which is shown in its entirety in Listing 11.

```
  int[] convertTo1D(double[][][] data){
    int imgRows = data.length;
    int imgCols = data[0].length;

    //Create the 1D array of type int to be
populated with
    // pixel data, one int value per pixel,
with four
    // color and alpha bytes per int value.
    int[] oneDPix = new int[imgCols * imgRows *
4];

    //Move the data into the 1D array.  Note
the use of the
    // bitwise OR operator and the bitwise
left-shift
    // operators to put the four 8-bit bytes
into each int.
    // Also note that the values are clamped at
0 and 255.
    for(int row = 0,cnt = 0;row <
imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        if(data[row][col][0] < 0)
data[row][col][0] = 0;
        if(data[row][col][0] > 255)
data[row][col][0]=255;
```

```
        if(data[row][col][1] < 0)
data[row][col][1] = 0;
        if(data[row][col][1] > 255)
data[row][col][1]=255;
        if(data[row][col][2] < 0)
data[row][col][2] = 0;
        if(data[row][col][2] > 255)
data[row][col][2]=255;
        if(data[row][col][3] < 0)
data[row][col][3] = 0;
        if(data[row][col][3] > 255)
data[row][col][3]=255;

        oneDPix[cnt] = (((int)data[row][col][0]
<< 24)
                                                &
0xFF000000)
                     | (((int)data[row][col][1]
<< 16)
                                                &
0x00FF0000)
                      |
(((int)data[row][col][2] << 8)
                                                &
0x0000FF00)
                           |
(((int)data[row][col][3])
                                                &
0x000000FF);
        cnt++;
      }//end for loop on col
    }//end for loop on row

    return oneDPix;
  }//end convertTo1D
```

Listing 11

Purpose of the convertTo1D method

The purpose of the **convertTo1D** method in Listing 11 is to convert the data in the 3D array of type **double** back into a 1d array of type **int** in the same format as the format produced by the invocation of the **getData** of the **DataBufferInt** class in Listing 4.

The int array data format

Each element of type **int** in the 1D array returned by the **convertTo1D** method contains the four unsigned data bytes that represent a single pixel. The most significant byte contains the alpha or transparency data. Moving from most to least significant, the remaining bytes contain the unsigned values for red, green, and blue in that order.

If the image has N columns of pixels in each row, the first N elements in the array contain the data for the first row of pixels, the second N elements contain the data for the second row of pixels, etc.

*(Note that the data values are clamped at 0 and 255 before casting to type **int**.)*

Reverse the process

This is the reverse of the process implemented by the method named **convertTo3D** that was invoked in Listing 4 to convert the image data from type **int** to type **double**.

(If you understand the use of bitwise operators, the code in Listing 11 shouldn't require further explanation. If not, you can learn about bitwise operators in the earlier lesson entitled Operators.)

Create the Image object

Returning to the **actionPerformed** method, Listing 12 invokes the **createImage** method of the **Component** class to create a new **Image** object from the 1D array of pixel data.

```
          modImgA = createImage(new
MemoryImageSource(

imgCols,imgRows,oneDPixA,0,imgCols));

Listing 12
```

Note that the **MemoryImageSource** class implements the **ImageProducer** interface and therefore satisfies one of the overloaded versions of the **createImage** method. If this is new to you, you can read about it in the Sun documentation. You can also read about it in my earlier lesson entitled The AWT Package, Graphics - Overview of Advanced Image Processing Capabilities.

At this point, the image data has been converted into an object of type **Image**, which is suitable for being passed as a parameter to the **drawImage** method of the **Graphics** class later in the overridden **paint** method.

Prepare the other image

Listing 13 uses the same methodology to convert the other 3D array of pixel data into an object of type **Image**.

```
          int[] oneDPixC =
convertTo1D(threeDPixModC);
          modImgC = createImage(new
MemoryImageSource(
```

```
imgCols,imgRows,oneDPixC,0,imgCols));
```

Listing 13

Repaint the display

Listing 14 invokes the **repaint** method. This sends a message to the operating system requesting that the overridden **paint** method be executed to cause the main display shown in Figure 2 and Figure 4 to be repainted.

```
          display.repaint();
```

Listing 14

The overridden paint method

This program invokes the **drawImage** method of the **Graphics** class to draw the three images on an object instantiated from a subclass of the **Canvas** class. The **Canvas** class is extended into a new inner class named **Display** to make it possible to override the **paint** method.

> *(If you don't know about inner classes, see my earlier lessons numbered 1636, 1638, and 1640.)*

You can view the code for the **Display** class and its overridden **paint** method in Listing 28. If you already know about the use of the overridden **paint** method in a callback sense to draw on the computer screen in Java, no explanation of the overridden **paint** method beyond the embedded comments should be needed. If you don't have that knowledge, go to Google and search for the following keywords:

baldwin java overridden paint

You will probably find that I have published more material on this topic than you will have the time to read.

Write the JPEG files

Listing 15 invokes the method named **writeJpegFile** twice to cause the images shown in the middle and the bottom of the displays in Figure 2 and Figure 4 to be written into output files in JPEG format. The names of the files are hard-coded into the program. However, it would be easy for you to make the file names input parameters to the program if you wished to do so. The middle image is written into the file named **junk1.jpg**, and the bottom image is written into the file named **junk2.jpg**.

```
writeJpegFile(modImgA,imgCols,imgRows,

"junk1.jpg");
```

```
writeJpegFile(modImgC,imgCols,imgRows,

"junk2.jpg");
```

Listing 15

You can view the entire method named **writeJpegFile** in Listing 28. Although the code is not completely straightforward, the embedded comments in the method, along with the Sun documentation of the classes and methods involved, should suffice to explain the method.

End anonymous listener class definition

The nested curly braces along with the right-parenthesis and the semicolon in Listing 16 signal the end of the definition of the anonymous listener class.

```
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener
    //End anonymous inner class registered on
the Replot
    // button.
```

Listing 16

When the code in Listing 16 has been executed, the anonymous class has been defined, an anonymous object of the anonymous class has been instantiated, and the anonymous object has been registered as an **ActionListener** on the **Replot** button shown at the top of each of the displays in Figure 2 and Figure 4.

Instantiate an image processing object

Continuing with the constructor code, Listing 17 instantiates a new object of the image processing class.

```
    try{
      imageProcessingObject =
(ImgIntfc04)Class.forName(

theProcessingClass).newInstance();
```

Listing 17

Recall that the name of the image processing class is provided as command-line parameter when the program is started. Listing 17 invokes the **forName** method of the class named **Class** to get an object of the class named **Class** that represents the image processing class.

If you are unfamiliar with the use of the **forName** method for this purpose, go to Google and search for the following keywords:

```
Baldwin java forName
```

The newInstance method

Then Listing 17 invokes the **newInstance** method of the class named **Class** to create a new object of the image processing class that is represented by the **Class** object.

At the risk of seeming redundant, if you are unfamiliar with the use of the **newInstance** method to create objects, go to Google and search for the following keywords:

```
Baldwin java forName newInstance
```

> *(Because I have published hundreds of tutorial lessons on hundreds of Java programming topics, you can find what I have had to say on most basic Java programming topics by searching for the appropriate keywords at Google.)*

Cast to type ImgIntfc04

Note that in Listing 17, the object of the image processing class is cast to the interface type, **ImgIntfc04** to make it compatible for storage in the instance variable named **imageProcessingObject** that was declared in Listing 1.

Cannot use a parameterized constructor

Also note that when you create a new object using the **newInstance** method, you do not have access to a parameterized constructor for the class. Therefore, when you define your image processing classes, there will be no point in defining parameterized constructors for the classes. I'll have more to say about this later in conjunction with the discussion of the ProgramTest class.

Post a counterfeit ActionEvent

Listing 18 posts a counterfeit **ActionEvent** to the system event queue and attributes it to the **Replot** button.

> *(See Event Handling in JDK 1.1, Posting Synthetic Events to the System Event Queue for an explanation of the use of the system event queue.)*

Posting this event causes the two image processing methods to be invoked in sequence at startup and causes the modified images to be displayed. The effect is exactly as if the user had clicked the **Replot** button on 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 18

Status at this point in the execution

At this point, the original image has been processed. The original image and both of modified images have been displayed as shown in the panels of Figure 2 and Figure 4.

From this point forward, each time the user clicks the **Replot** button, the image will be processed again and the images returned from the image processing methods will be displayed along with the original image.

The catch block

Listing 18 also contains a **catch** block that matches up with the **try** block that began in Listing 17.

Make everything visible

Although the frame itself was made visible earlier to facilitate the reading of the inset values, it is still necessary to cause the composite of the frame, the canvas, the button, and the images to become visible. This is accomplished in Listing 19.

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

Listing 19

Activate the close button

Listing 20 defines, instantiates, and registers a **WindowListener** object that causes the program to terminate when the user clicks the X-button in the upper right of the frames shown in Figure 2 and Figure 4.

```
    this.addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
```

```
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener

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

  }//end constructor

Listing 20
```

Listing 20 also signals the end of the constructor and the end of the class named **ImgMod04**.

The Class Named ImgMod04a

As mentioned earlier, while I was at it, I decided to go ahead and provide a class named **ImgMod04a** that contains all of the features of **ImgMod04** except that it supports only one image processing method.

The most notable thing about this new class is that it replaces the earlier class named **ImgMod02a** with a class that writes the modified image into an output file in JPEG format. The name of the output file is **junk.jpg**.

A complete listing of the class is provided in Listing 30. If you understand the class named **ImgMod04**, you should have no problem understanding the class named **ImgMod04a**.

Usage

Typical usage of this new class is as follows:

```
java ImgMod04a ProcessingProgramName ImageFileName
```

A built-in test program

For test and illustration purposes, the source code includes a class definition for an image processing program named **ProgramTestA**. This test program is very similar to the test program named **ProgramTest**, which I will explain later. If you understand the test program named **ProgramTest**, you should have no difficulty understanding the test program named **ProgramTestA**.

The interface named ImgIntfc04a

A program that is compatible with being driven by the new class named **ImgMod04a** must implement the interface named **ImgIntfc04a**. A complete listing of the interface is provided in Listing 31.

If you understand the interface named **ImgIntfc04**, you should have no difficulty understanding the interface named **ImgIntfc04a**.

The Class Named ProgramTest

As mentioned earlier, for test and illustration purposes, the source code includes a class definition for an image processing program named **ProgramTest**. A complete description of the class was provided earlier.

The class definition, including the declaration of some instance variables, begins in Listing 21.

```
class ProgramTest extends Frame implements
ImgIntfc04{

  double slope;//Controls the slope of the line
  String inputData;//Obtained via the TextField
  TextField inputField;//Reference to TextField

Listing 21
```

Must implement ImgIntfc04

This is an *image processing program* in the sense of the command-line parameter indicated by **ProcessingProgramName** earlier. For example, this image processing program can be run by entering the following at the command line:

```
java ImgMod04 ProgramTest ImgMod04Test.jpg
```

Perhaps the most important thing about Listing 21 is the fact that the class implements the interface named **ImgIntfc04**. All image processing classes must implement that interface to make them capable of being driven by the class named **ImgMod04**.

The constructor

The constructor for the class is shown in its entirety in Listing 22.

```
  ProgramTest(){
    //Create and display the user-input GUI.
    setLayout(new FlowLayout());

    Label instructions = new Label(
                         "Type a slope value
and Replot.");
    add(instructions);

    inputField = new TextField("1.0",5);
    add(inputField);

    setTitle("Copyright 2006, Baldwin");
    setBounds(400,0,200,100);
    setVisible(true);
  }//end constructor
```

Listing 22

The purpose of the constructor is to construct the simple GUI shown in Figure 3. The code in Listing 22 is straightforward and shouldn't require further explanation.

The method named processImg01

The method named **processImg01** must be defined to implement the **ImgIntfc04** interface. This is the first image processing method that is applied to the original image file that is specified on the command line.

Note that this method must return references to two 3D array objects of type **double** encapsulated in an object of the **ImgIntfc04Method01Output** class. That class is defined in the same source file that defines the interface named **ImgIntfc04** *(see Listing 29)*. As you learned earlier, the data in one of the 3D array objects is displayed as the middle image in the displays shown in Figure 4. The reference to the other array is passed as a parameter to the second image processing method named **processImg02**.

Can be simple or complex

The method named **processImg01** can be as simple or as complex as needed. This test version is relatively simple. The version that I have developed to do the watermark work shown in Figure 2 is much more complex. *(I will publish and explain it in a future lesson.)*

Listing 23 shows the beginning of the method named **processImg01**.

```
  public ImgIntfc04Method01Output
                     processImg01(double[][][]
threeDPix){

    //Determine number of rows and cols
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    //Display some interesting information
    System.out.println("Program test");
    System.out.println("Width = " + imgCols);
    System.out.println("Height = " + imgRows);

    //Make a working copy of the 3D array to
avoid making
    // permanent changes to the image data.
    double[][][] temp3D =
copy3DArray(threeDPix);
```

Listing 23

The code in Listing 23 gets the dimensions of the image, displays some interesting information, and then invokes the method named **copy3DArray** to make a working copy of the 3D array to

avoid making permanent changes to the original image data. The code in the method named **copy3DArray** is straightforward and shouldn't require further explanation. You can view that code in Listing 28.

Draw a white diagonal line

Next, the code in Listing 24 gets a value for the slope of a line from the text field shown in Figure 3 and uses that slope to draw a sloping white diagonal line on the image as shown by the middle image in the displays in Figure 4.

```
    //Get slope value from the TextField
    slope =
Double.parseDouble(inputField.getText());

    //Draw a white diagonal line on the image.
    for(int col = 0;col < imgCols;col++){
      int row = (int)(slope * col);
      if(row > imgRows -1) break;
      //Set values for alpha, red, green, and
      // blue colors.
      temp3D[row][col][0] = 255.0;
      temp3D[row][col][1] = 255.0;
      temp3D[row][col][2] = 255.0;
      temp3D[row][col][3] = 255.0;

    }//end for loop
```

Listing 24

The three images in the left panel of Figure 4 were drawn with the default slope value of 1.0. The three images in the right panel of Figure 4 were drawn with a user-defined slope value of 0.5.

Invert the colors

You may have noticed that the line in the middle images of Figure 4 is black instead of white. This is because the code in Listing 25 invokes the method named **invertColors** to invert the colors in the image. Thus, white becomes black and the other colors are replaced by their inverted values.

```
    invertColors(temp3D);
```

Listing 25

I explained color inversion in the earlier lesson entitled Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion. Once you understand the general concept of color inversion, the method named **invertColors** is straightforward and shouldn't require further explanation. The method can be viewed in its entirety in Listing 28. *(Note that the method does not invert the alpha values because that would cause an opaque image to become transparent.)*

Return the required references

Listing 26 instantiates a new object of the **ImgIntfc04Method01Output** class encapsulating references to two 3D arrays of type **double**.

```
    return new
ImgIntfc04Method01Output(temp3D,temp3D);
  }//end processImg01

Listing 26
```

Both references point to the same array object

In this case, both of the references passed to the constructor for the **ImgIntfc04Method01Output** object refer to the same 3D array. This causes the array to be displayed in the middle images of Figure 4 and also to be passed as input to the second processing method.

A second alternative

A second alternative is to apply two different processes to the original image within the first processing method and to return the results of those two processes as the two required references. I will illustrate that alternative in conjunction with the hidden watermark work *(see Figure 2)* that I will explain in a future lesson.

A third alternative

A third alternative would be to return a reference to a 3D array containing the data from the original image as one of the two required references. This would make it possible to cause the two image processing methods to apply different processes to the same original image data. I may also illustrate that alternative in a future lesson.

Listing 26 signals the end of the image processing method named **processImg01**.

The image processing method named processImg02

The method named **processImg02** must also be defined to satisfy the interface named **ImgIntfc04**. This method is shown in its entirety in Listing 27.

```
  public double[][][] processImg02(double[][][]
threeDPix){

    //Make a working copy of the 3D array to
avoid making
    // permanent changes to the image data.
    double[][][] temp3D =
copy3DArray(threeDPix);
```

```
    //Invert the colors.  Don't invert the
alpha value.
    invertColors(temp3D);

    return temp3D;
  }//end processImg02
```

Listing 27

The method begins by making a working copy of the 3D array received as an incoming parameter.

Then it invokes the same method named **invertColors** described above to invert the colors in the image. This effectively cancels out the inversion performed earlier by the method named **processImg01** causing the returned image to consist of a white diagonal line on a background that matches the original image. This is illustrated by the two bottom images in Figure 4. The bottom image on the left shows a line with the default slope of 1.0. The bottom image on the right shows a line with a user-defined slope of 0.5.

And that concludes the discussion of the class named **ProgramTest**.

Run the Program

I encourage you to copy the code from Listing 28, Listing 29, Listing 30, and Listing 31 into your text editor. Compile the code and execute it. Experiment with it, making changes, and observing the results of your changes.

For example, it might be useful to modify the display to cause the three images to be displayed in a side-by-side format across the screen rather than up and down the screen.

Another useful alternative would be to cause the three images to be displayed in independent frames to make it possible to move them around the screen independently of one another.

You might also want to try adding one or more additional processing stages so that more than two different processes can be applied to an image with the output from all processing stages being displayed.

You might want to modify the program to support the specification of two or more input images to make it possible to define processes that require more than one image *(such as morphing one image into another image for example)*.

Above all, have fun and learn as much as you can about image processing.

Summary

In this lesson I presented and explained a new image processing framework named **ImgMod04** that incorporates the following features:

- The ability to perform two sequential pixel-modification processes *(instead of just one)* on an image and to display the modified image resulting from each process in addition to displaying the original image.
- The ability to write two modified images into output files in JPEG format. *(The names of the output files are **junk1.jpg** and **junk2.jpg**. They are written into the current directory.)*
- The elimination of several conversions back and forth between type **double** and type **int**. All computations can now be performed as type **double**, and the data is maintained as type **double** from the initial conversion from **int** to **double** to the point where the data is ready to be displayed and written into a JPEG file.
- The **Replot** button *(see Figure 1)* was moved to the top of the display *(see Figure 2)* to make it accessible when the display is too long to fit on the screen. *(For purposes of viewing the display in that case, the frame can be moved up and down on the screen using the right mouse button and the up and down arrow keys, but the lower portion of the bottom display still gets chopped off.)*

In addition, I provided another new image processing framework named **ImgMod04a** that contains all of the features of **ImgMod04** except that it supports only one image processing method.

The most notable thing about this new class is that it replaces the earlier class named **ImgMod02a** with a class that writes the modified image into an output file in JPEG format. The name of the output file is **junk.jpg**.

What's Next?

I plan to use the new image processing frameworks to continue publishing Java programming tutorials on interesting image processing topics. This will include lessons on the creation of both visible and hidden watermarks in the images.

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
2444 Understanding the Discrete Cosine Transform in Java
2446 Understanding the 2D Discrete Cosine Transform in Java
2448 Understanding the 2D Discrete Cosine Transform in Java, Part 2

Complete Program Listings

Complete listings of the programs discussed in this lesson are shown in Listing 28, Listing 29, Listing 30, and Listing 31 below.

```
/*File ImgMod04.java
Copyright 2006, R.G.Baldwin

The purpose of this program is to make it easy to
experiment with the modification of pixel data in an image
and to display two modified versions of the image along
with the original image.

This program is an update of the earlier program named
ImgMod02a.  This update supports the following new
features:

The ability to perform two sequential processes on an image
and to display the modified image resulting from each
process in addition to the original image.

The ability to write two modified images into output files
in JPEG format.  The names of the output files are
junk1.jpg and junk2.jpg.  They are written into the current
directory.

The elimination of several conversions back and forth
between type double and type int.  All computations are now
performed as type double, and the data is maintained as
type double from the initial conversion from int to double
to the point where it is ready to be displayed or written
into a JPEG file.

The Replot button was moved to the top of the display to
make it accessible when the display is too long 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 extracts the pixel data from an image file into
a 3D array of type:

double[row][column][depth].

The first two dimensions of the array correspond to the
rows and columns of pixels in the image.  The third
dimension always has a value of 4 and contains the
following values by index:

0 alpha
1 red
2 green
3 blue
```

```
Note that these values are stored as type double rather
than type unsigned byte which is the format of pixel data
in the original image file.  This type conversion
eliminates many problems involving the requirement to
perform unsigned arithmetic on unsigned byte data.

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

Operation:  This program provides a framework that is
designed to invoke another program to process the pixels
extracted from an image.  In other words, this program
extracts the pixels and puts them in a format that is
relatively easy to work with.  A second program is invoked
to actually process the pixels.  Typical usage is as
follows:

java ImgMod04 ProcessingProgramName ImageFileName

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

If the ImageFileName is not specified on the command line,
the program will search for an image file in the current
directory named ImgMod04Test.jpg and will process it using
the processing program specified by the second command-
line argument.

If both command-line arguments are omitted, the program
will search for an image file in the current directory
named ImgMod04Test.jpg and will process it using the built-
in processing program named ProgramTest.  A complete
description of the behavior of the test program is provided
by comments in the source code for the class named
ProgramTest.

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 two
processed versions of the image are displayed in a frame
with the original image above the two processed images.  A
Replot button appears at the top of the frame.  If the
user clicks the Replot button, the image processing methods
are re-run on the original image.  The original image is
reprocessed and the newly processed versions of the images
replace the old versions.

The processing program may provide a  GUI for data input
making it possible for the user to modify the behavior of
the image processing methods each time they are run.  This
capability is illustrated by the built-in processing
```

program named ProgramTest.

The image processing programming must implement the interface named ImgIntfc04. That interface declares two image processing methods with the following signatures:

```
ImgIntfc04Method01Output processImg01(double[][][] input);

double[][][] processImg02(double[][][] input);
```

The first processing method must return references to two 3D double array objects encapsulated in an object of the class ImgIntfc04Method01Output. The class definition for this class is contained in the same source code file as the source code for the interface named ImgIntfc04.

The contents of the array referred to by one of the returned references is displayed as the middle image on the main display panel. The other reference is passed as a parameter to the second processing method. Depending on the objectives, the two references may point to the same or to different array objects.

The second processing method must return a reference to a single 3D double array object. The contents of the array object are displayed in the bottom image, and are also written into an output file in JPEG format.

Both processing methods receive a reference to a 3D double array object containing image pixel data in the format described earlier.

The image processing program cannot have a parameterized constructor. This is because an object of the class is instantiated by invoking the newInstance method of the class named Class on the name of the image processing program provided as a String on the command line. This approach to object instantiation does not allow parameterized constructors.

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

The processImg methods receive a 3D array containing pixel data. They should make a copy of the incoming array and modify the copy rather than modifying the original. Then the programs should return references to the modified copies of the 3D pixel arrays.

The processImg methods are free to modify the values of the pixels in the incoming array in any manner before returning reference to the modified arrays. Note however that native pixel data consists of four unsigned bytes. If the modification of the pixel data produces negative values or positive value greater than 255, this should be dealt with before returning the modified pixel data. Otherwise, the

```
results of displaying the modified pixel data may not be as
 expected.

There are at least two ways to deal with this situation.
One way is to simply clamp all negative values at zero and
to clamp all values greater than 255 at 255.  The other way
is to perform a further modification so as to map the
range from -x to +y into the range from 0 to 255.  There is
no single correct way for all situations.

When the processImg methods return, this program causes the
original image and the modified images to be displayed in a
frame on the screen with the original image above the
modified images.

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

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

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

class ImgMod04 extends Frame{
  Image rawImg;//A reference to the raw image.
  int imgCols;//Number of horizontal pixels
  int imgRows;//Number of rows of pixels
  Image modImgA;//Reference to first modified image
  Image modImgC;//Reference to third modified image

  //Default image processing program.  This class will be
  // executed to process the image if the name of another
  // class is not entered on the command line.  Note that
  // the source code for this class file is included in
  // this source code file.
  static String theProcessingClass = "ProgramTest";

  //Default image file name.  This image file will be
  // processed if another file name is not entered on the
  // command line.  You must provide this file in the
  // current directory if it is going to be processed.
  static String theImgFile = "ImgMod04Test.jpg";

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

  //Reference to the image processing object.
  ImgIntfc04 imageProcessingObject;
  //-----------------------------------------------------//

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

```
    //Get names for the image processing class and the
    // image file to be processed.  Program supports 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 == 1){
      theProcessingClass = args[0];
      //Use default image file along with this class.
    }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.
    ImgMod04 obj = new ImgMod04();
  }//end main
  //----------------------------------------//

  public ImgMod04(){//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.  This local
    // variable must be declared final because it is
    // accessed from within an anonymous inner class.
    final double[][][] threeDPix = 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 so as 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.
    int inTop = this.getInsets().top;
    int inLeft = this.getInsets().left;
    int buttonHeight = replotButton.getSize().height;

    //Size the frame so that a small amount of yellow
    // background will show on the right, between the
    // images, and on the bottom when all three images are
    // displayed, one above the other.
    this.setSize(2*inLeft+imgCols + 2,inTop
```

```
                          + buttonHeight + 3*imgRows + 9);

    //=================================================//
    //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){
          //Pass a 3D array of pixel data to the first
          // processing method.  This method returns two 3D
          // arrays of pixel data, outputA and outputB
          // encapsulated in an object of type
          // ImgIntfc04Method01Output.
          //The contents of outputA are displayed as the
          // middle image in the display.  The contents of
          // outputB are passed along to the second
          // processing method, which returns a single 3D
          // array of pixel data.
          //The contents of outputA are also written into
          // an output JPEG file named junk1.jpg.
          //The contents of the array returned by the
          // second processing method are displayed as the
          // bottom image in the display.
          //The contents of that array are also written
          // into an output JPEG file named junk2.jpg.
          //To cause the same 3D array to be displayed in
          // the middle and also to be passed into the
          // second processing method, write the first
          // processing method return two references to the
          // same 3D array.
          ImgIntfc04Method01Output output =
             imageProcessingObject.processImg01(threeDPix);
          //Extract the references to the two 3D array
          // objects of type double from the returned
          // object.
          double[][][] threeDPixModA = output.outputA;
          double[][][] threeDPixModB = output.outputB;

          //Pass the contents of outputB to the second
          // processing method.
          double[][][] threeDPixModC =
                        imageProcessingObject.processImg02(
                                              threeDPixModB);

          //Now prepare the two pixel arrays for display.

          //Convert one 3D array of modified pixel data of
          // type double to a 1D array of pixel data of
          // type int.  This 1D array is in a standard
          // pixel format.  See a brief description of the
          // format in the comments in the method named
          // convertTo1D.
```

```
        int[] oneDPixA = convertTo1D(threeDPixModA);
        //Use the createImage() method of the Component
        // class to create a new Image object from the
        // 1D array of pixel data.  Note that
        // MemoryImageSource implements the
        // ImageProducer interface and therefore
        // satisfies one of the overloaded versions of
        // the createImage method.
        modImgA = createImage(new MemoryImageSource(
                    imgCols,imgRows,oneDPixA,0,imgCols));

        //Convert the other array of modified pixel data
        // to a 1D array of pixel data using the same
        // methodology.
        int[] oneDPixC = convertTo1D(threeDPixModC);
        modImgC = createImage(new MemoryImageSource(
                    imgCols,imgRows,oneDPixC,0,imgCols));

        //Repaint the image display frame with the
        // original image at the top and the modified
        // images in the center and at the bottom.
        display.repaint();

        //Write the middle modified image into a JPEG
        // file named junk1.jpg
        writeJpegFile(modImgA,imgCols,imgRows,
                                              "junk1.jpg");

        //Write the final modified image into a JPEG
        // file named junk2.jpg.
        writeJpegFile(modImgC,imgCols,imgRows,
                                              "junk2.jpg");

      }//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 = (ImgIntfc04)Class.forName(
                      theProcessingClass).newInstance();

     //Post a counterfit 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
```

```
      // two image processing methods to be invoked in
      // sequence at startup and causes the modified
      // images 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 both modified images have been
     // displayed.  From this point forward, each time the
     // user clicks the Replot button, the image will be
     // processed again and the new modified images 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 constructor
 //=====================================================//

 //Inner class for canvas object on which to display the
 // three images.
 class Display extends Canvas{
   //Override the paint method to display the rawImg and
   // the two modified 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((rawImg != null) &&
          (modImgA != null) &&
```

```
           (modImgC != null)){
          g.drawImage(rawImg,0,0,this);
          g.drawImage(modImgA,0,imgRows + 2,this);
          g.drawImage(modImgC,0,2*imgRows + 4, this);
        }//end if
      }//end if
    }//end paint()
  }//end class myCanvas
//=========================================================//

  //Save pixel values as type double to make
  // arithmetic easier later.

  //The purpose of this method is to convert the data in
  // the int oneDPix array into a 3D array of type double.
  //The dimensions of the 3D array are row, col, and color
  // in that order.
  //Row and col correspond to the rows and columns of the
  // pixels in the image.  Color corresponds to
  // transparency and color information at the following
  // index levels in the third dimension:
  // 0 alpha (tranparency)
  // 1 red
  // 2 green
  // 3 blue
  // The structure of this code is determined by the way
  // that the pixel data is formatted into the 1D array of
  // pixel data of type int when the image file is read.
  double[][][] convertTo3D(
                   int[] oneDPix,int imgCols,int imgRows){
    //Create the new 3D array to be populated with pixwl
    // data.
    double[][][] data = new double[imgRows][imgCols][4];

    for(int row = 0;row < imgRows;row++){
      //Extract a row of pixel data into a temporary array
      // of ints
      int[] aRow = new int[imgCols];
      for(int col = 0; col < imgCols;col++){
        int element = row * imgCols + col;
        aRow[col] = oneDPix[element];
      }//end for loop on col

      //Move the data into the 3D array.  Note the use of
      // bitwise AND and bitwise right shift operations to
      // mask all but the correct set of eight bits.
      for(int col = 0;col < imgCols;col++){
        //Alpha data
        data[row][col][0] = (aRow[col] >> 24) & 0xFF;
        //Red data
        data[row][col][1] = (aRow[col] >> 16) & 0xFF;
        //Green data
        data[row][col][2] = (aRow[col] >> 8)  & 0xFF;
        //Blue data
        data[row][col][3] = (aRow[col]) & 0xFF;
      }//end for loop on col
```

```
    }//end for loop on row
    return data;
  }//end convertTo3D
  //----------------------------------------//

  //The purpose of this method is to convert the data in
  // the 3D array of type double back into the a 1d array
  // of type int.  This is the reverse of the method named
  // convertTo3D.  Note that the data values are clamped at
  // 0 and 255 before casting to type int.
  int[] convertTo1D(double[][][] data){
    int imgRows = data.length;
    int imgCols = data[0].length;

    //Create the 1D array of type int to be populated with
    // pixel data, one int value per pixel, with four
    // color and alpha bytes per int value.
    int[] oneDPix = new int[imgCols * imgRows * 4];

    //Move the data into the 1D array.  Note the use of the
    // bitwise OR operator and the bitwise left-shift
    // operators to put the four 8-bit bytes into each int.
    // Also note that the values are clamped at 0 and 255.
    for(int row = 0,cnt = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        if(data[row][col][0] < 0) data[row][col][0] = 0;
        if(data[row][col][0] > 255) data[row][col][0]=255;
        if(data[row][col][1] < 0) data[row][col][1] = 0;
        if(data[row][col][1] > 255) data[row][col][1]=255;
        if(data[row][col][2] < 0) data[row][col][2] = 0;
        if(data[row][col][2] > 255) data[row][col][2]=255;
        if(data[row][col][3] < 0) data[row][col][3] = 0;
        if(data[row][col][3] > 255) data[row][col][3]=255;

        oneDPix[cnt] = (((int)data[row][col][0] << 24)
                                              & 0xFF000000)
                     | (((int)data[row][col][1] << 16)
                                              & 0x00FF0000)
                      | (((int)data[row][col][2] << 8)
                                              & 0x0000FF00)
                           | (((int)data[row][col][3])
                                              & 0x000000FF);
        cnt++;
      }//end for loop on col
    }//end for loop on row

    return oneDPix;
  }//end convertTo1D
  //----------------------------------------------------//

  //Write the contents of an Image object to a JPEG file
  // with the name specified by the incoming parameter
  // fileName.
  void writeJpegFile(Image img,int width,int height,
                                          String fileName){
    //Create an off-screen drawable image by calling the
```

```
    // createImage method of the Component class.  Cast
    // it to type BufferedImage.
    BufferedImage bufferedImg =
                  (BufferedImage)createImage(width,height);

    //Call the createGraphics method of the BufferedImage
    // class to create a Graphics2D object, which can be
    // used to draw into the BufferedImage object.
    Graphics2D bufferedGraphics =
                              bufferedImg.createGraphics();

    //Call the drawImage method of the Graphics class to
    // draw the image into the off-screen buffer.  Pass
    // null as the ImageObserver.
    bufferedGraphics.drawImage(img,0,0,null);

    try{
      //Get a file output stream.
      FileOutputStream outStream =
                             new FileOutputStream(fileName);
      //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(bufferedImg,"jpeg",outStream);
      outStream.close();
    }catch (Exception e) {
      e.printStackTrace();
    }//end catch
  }//end writeJpegFile
  //-----------------------------------------------------//

  //This method reads an image from a specified image file
  // and converts the pixel data into a 3D array of type
  // double.  The name of the image file is specified as
  // a String by theImgFile.
  double[][][] getTheImage(){
    rawImg = Toolkit.getDefaultToolkit().
                                       getImage(theImgFile);

    //Use a MediaTracker object to block until the image is
    // loaded or ten seconds has elapsed.
    tracker = new MediaTracker(this);
    tracker.addImage(rawImg,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

    //Raw image has been loaded.  Get width and height of
    // the raw image.
    imgCols = rawImg.getWidth(this);
    imgRows = rawImg.getHeight(this);

    //Create a 1D array object to receive the pixel
    // representation of the image
    int[] oneDPix = new int[imgCols * imgRows];

    //Create an empty BufferedImage object
    BufferedImage buffImage = new BufferedImage(
                               imgCols,
                               imgRows,
                               BufferedImage.TYPE_INT_ARGB);

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

    //Convert the BufferedImage to numeric pixel
    // representation.
    DataBufferInt dataBufferInt =
                      (DataBufferInt)buffImage.getRaster().
                                              getDataBuffer();
    oneDPix = dataBufferInt.getData();

    //Convert the pixel byte data in the 1D array to
    // double data in a 3D array to make it easier to work
    // with the pixel data later.  Recall that pixel data
    // is unsigned byte data and Java does not support
    // unsigned arithmetic. Performing unsigned arithmetic
    // on byte data is particularly cumbersome.
    return convertTo3D(oneDPix,imgCols,imgRows);
  }//end getTheImage
  //-----------------------------------------------------//
}//end ImgMod04.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 class program named ImgMod04 and the
// interface named ImgIntfc04.

//The constructor for the class displays a small frame on
// the screen with a single textfield. The purpose of the
// text field is to allow the user to enter a value that
// represents the slope of a line.  In operation, the user
// types a value into the text field and then clicks the
```

```
// Replot button on the main image display frame.  The user
// is not required to press the Enter key after typing the
// new value, but it doesn't do any harm to do so.
// Negative slopes are not supported.  An attempt to use
// a negative slope will cause the program to abort with
// an error.

//The method named processImage01 receives a 3D array of
// type double containing alpha, red, green, and blue
// values for an image.

// The 3D array that is received by processImg01 is
// modified by the method to cause a white diagonal line to
// be drawn down and to the right from the upper left
// corner of the image.  The slope of the line is
// controlled by the value in the text field.  Initially,
// this value is 1.0, but the value can be modified by the
// user.  (If the characters in the text field cannot be
// converted to a numeric type double, the program will
// abort with an error.)
//After drawing the line, the method inverts the colors in
// the image.  This results in a black line on an image
// with inverted colors.

//The method named processImg01 is required to return
// references to two 3D array objects of type double
// encapsulated in an object of type
// ImgIntfc04Method01Output.  This version of the method
// returns two references to the same modified pixel array,
// but that is not necessary.  The method could return
// references to two different pixel arrays.

//The data in the 3D array referenced by the value stored
// in the variable named objectA of the returned object is
// displayed as the center image of the main image display
// by the code in the class named ImgMod04. The reference
// stored in the variable named objectB of the returned
// object is passed as a parameter to the method named
// processImg02 by the code in the class named ImgMod04.
// In this version of the method,both references point to
// the same 3D array object, so the same reference is
// displayed and also passed to the processImg02 method.

//The method named processImg02 receives an incoming 3D
// array of type double containing pixel data and returns
// one reference to a 3D array of type double containing
// pixel data.  This version of the method performs a
// color inversion on the data and returns a reference to
// the modified array.  This reverses the color inversion
// performed by processImg01, causing the bottom image to
// match the original image except that it has a white
// diagonal line drawn on it.

//To cause a new line to be drawn, type a new slope
// value into the text field and click the Replot button
// at the top of the main image display frame.
```

```
//This class extends Frame.  However, a compatible class is
// not required to extend the Frame class. This example
// extends Frame because it provides a GUI for user data
// input.

//A compatible class is required to implement the
// interface named ImgIntfc04.

class ProgramTest extends Frame implements ImgIntfc04{

  double slope;//Controls the slope of the line
  String inputData;//Obtained via the TextField
  TextField inputField;//Reference to TextField

  //Constructor must take no parameters
  ProgramTest(){
    //Create and display the user-input GUI.
    setLayout(new FlowLayout());

    Label instructions = new Label(
                         "Type a slope value and Replot.");
    add(instructions);

    inputField = new TextField("1.0",5);
    add(inputField);

    setTitle("Copyright 2006, Baldwin");
    setBounds(400,0,200,100);
    setVisible(true);
  }//end constructor
  //-----------------------------------------------------//

  //The following method must be defined to implement the
  // ImgIntfc04 interface.  Note that this method must
  // return references to two 3D array objects of type
  // double encapsulated in an object of the
  // ImgIntfc04Method01Output class.  That class is defined
  // in the same source file that defines the interface
  // named ImgIntfc04.
  public ImgIntfc04Method01Output
                      processImg01(double[][][] threeDPix){

    //Determine number of rows and cols
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    //Display some interesting information
    System.out.println("Program test");
    System.out.println("Width = " + imgCols);
    System.out.println("Height = " + imgRows);

    //Make a working copy of the 3D array to avoid making
    // permanent changes to the image data.
    double[][][] temp3D = copy3DArray(threeDPix);
```

```
    //Get slope value from the TextField
    slope = Double.parseDouble(inputField.getText());

    //Draw a white diagonal line on the image.
    for(int col = 0;col < imgCols;col++){
      int row = (int)(slope * col);
      if(row > imgRows -1) break;
      //Set values for alpha, red, green, and
      // blue colors.
      temp3D[row][col][0] = 255.0;
      temp3D[row][col][1] = 255.0;
      temp3D[row][col][2] = 255.0;
      temp3D[row][col][3] = 255.0;

    }//end for loop

    //Invert the colors.  Do not invert the alpha value.
    invertColors(temp3D);

    //Return two references to the same array of image
    // data.
    return new ImgIntfc04Method01Output(temp3D,temp3D);
  }//end processImg01
  //-----------------------------------------------------//

  //The following method must be defined to
  // implement the ImgIntfc04 interface.
  public double[][][] processImg02(double[][][] threeDPix){

    //Make a working copy of the 3D array to avoid making
    // permanent changes to the image data.
    double[][][] temp3D = copy3DArray(threeDPix);

    //Invert the colors.  Don't invert the alpha value.
    invertColors(temp3D);

    return temp3D;
  }//end processImg02
  //-----------------------------------------------------//

  //This method copies a double version of a 3D pixel array
  // to an new pixel array of type double.
  double[][][] copy3DArray(double[][][] threeDPix){
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    double[][][] new3D = new double[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        new3D[row][col][0] = threeDPix[row][col][0];
        new3D[row][col][1] = threeDPix[row][col][1];
        new3D[row][col][2] = threeDPix[row][col][2];
        new3D[row][col][3] = threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop
    return new3D;
```

```
  }//end copy3DArray
  //-------------------------------------------------//

  //This method inverts the colors in a double version of a
  // 3D array of pixel data.  It doesn't invert the alpha
  // value.
  void invertColors(double[][][] data3D){
    int imgRows = data3D.length;
    int imgCols = data3D[0].length;
    //Invert the colors.  Don't invert the alpha value.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        data3D[row][col][1] = 255 - data3D[row][col][1];
        data3D[row][col][2] = 255 - data3D[row][col][2];
        data3D[row][col][3] = 255 - data3D[row][col][3];
      }//end inner loop
    }//end outer loop
  }//end invertColors
  //-------------------------------------------------//
}//end class ProgramTest


Listing 28
```

Listing 29

```
/*File ImgIntfc04.java
Copyright 2006, R.G.Baldwin

The purpose of this interface is to declare the two
methods required by image processing classes that are
compatible with the program named ImgMod04.java.

Note that this interface was modified relative to the
interface named ImgIntfc02 to deal only with image pixel
data as type double instead of type int.

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

interface ImgIntfc04{
  ImgIntfc04Method01Output processImg01(
                                      double[][][] input);

  double[][][] processImg02(double[][][] input);
}//end ImgIntfc04
//=================================================//

//This class makes it possible for the method named
// processImg01 to return two references to 3D array object
// of type double.
class ImgIntfc04Method01Output{
  public double[][][] outputA;
  public double[][][] outputB;
```

```
  //Constructor
  public ImgIntfc04Method01Output(double[][][] outputA,
                                  double[][][] outputB){
    this.outputA = outputA;
    this.outputB = outputB;
  }//end constructor
}//end class ImgIntfc04Method01Output
//=====================================================//
```

Listing 29

Listing 30

```
/*File ImgMod04a.java
Copyright 2006, R.G.Baldwin

This is a modification to the class named ImgMod04 that
eliminates the use of a second image-processing method.
Otherwise, it is identical to the class named ImgMod04.

Note that some of the comments may not have been updated to
reflect the modifications.

The purpose of this program is to make it easy to
experiment with the modification of pixel data in an image
and to display a modified version of the image along
with the original image.

This program is an update of the earlier program named
ImgMod02a.  This update supports the following new
features:

The ability to write 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 elimination of several conversions back and forth
between type double and type int.  All computations are now
performed as type double, and the data is maintained as
type double from the initial conversion from int to double
to the point where it is ready to be displayed or written
into a JPEG file.

The Replot button was moved to the top of the display to
make it accessible when the display is too long 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 extracts the pixel data from an image file into
a 3D array of type:

double[row][column][depth].
```

The first two dimensions of the array correspond to the rows and columns of pixels in the image. The third dimension always has a value of 4 and contains the following values by index:

0 alpha
1 red
2 green
3 blue

Note that these values are stored as type double rather than type unsigned byte which is the format of pixel data in the original image file. This type conversion eliminates many problems involving the requirement to perform unsigned arithmetic on unsigned byte data.

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

Operation: This program provides a framework that is designed to invoke another program to process the pixels extracted from an image. In other words, this program extracts the pixels and puts them in a format that is relatively easy to work with. A second program is invoked to actually process the pixels. Typical usage is as follows:

java ImgMod04a ProcessingProgramName ImageFileName

For test and illustration purposes, the source code includes a class definition for an image processing program named ProgramTestA.

If the ImageFileName is not specified on the command line, the program will search for an image file in the current directory named ImgMod04aTest.jpg and will process it using the processing program specified by the second command-line argument.

If both command-line arguments are omitted, the program will search for an image file in the current directory named ImgMod04aTest.jpg and will process it using the built in processing program named ProgramTestA. A complete description of the behavior of the test program is provided by comments in the source code for the class named ProgramTestA.

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 one processed version of the image are displayed in a frame with the original image above the processed image. A Replot button appears at the top of the frame. If the

user clicks the Replot button, the image processing method is re-run on the original image. The original image is reprocessed and the new processed version of the image replaces the old version.

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 it is run. This capability is illustrated in the built-in processing program named ProgramTestA.

The image processing program must implement the interface named ImgIntfc04a. That interface declares an image processing method with the following signature:

```
double[][][]  processImg(double[][][] input);
```

The image processing method must return a reference to a 3D double array object.

The processing method receives a reference to a 3D double array object containing image pixel data in the format described earlier.

The image processing program cannot use a parameterized constructor. This is because an object of the class is instantiated by invoking the newInstance method of the class named Class on the name of the image processing program provided as a String on the command line. This approach to object instantiation does not support parameterized constructors.

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

The processImg method receive a 3D array containing pixel data. It should make a copy of the incoming array and modify the copy rather than modifying the original. Then the method should return a reference to the modified copy of the 3D pixel array.

The processImg method is free to modify the values of the pixels in the incoming array in any manner before returning reference to the modified array. Note however that native pixel data consists of four unsigned bytes. If the modification of the pixel data produces negative values or positive value greater than 255, this should be dealt with before returning the modified pixel data. Otherwise, the results of displaying the modified pixel data may not be as expected.

There are at least two ways to deal with this situation. One way is to simply clamp all negative values at zero and to clamp all values greater than 255 at 255. The other way is to perform a further modification so as to map the range from -x to +y into the range from 0 to 255. There is

```
no single correct way for all situations.

When the processImg method returns, this program causes the
original image and the modified image to be displayed in a
frame on the screen with the original image above the
modified image.

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

class ImgMod04a extends Frame{
  Image rawImg;//A reference to the raw image.
  int imgCols;//Number of horizontal pixels
  int imgRows;//Number of rows of pixels
  Image modImg;//Reference to modified image

  //Default image processing program.  This class will be
  // executed to process the image if the name of another
  // class is not entered on the command line.  Note that
  // the source code for this class file is included in
  // this source code file.
  static String theProcessingClass = "ProgramTestA";

  //Default image file name.  This image file will be
  // processed if another file name is not entered on the
  // command line.  You must provide this file in the
  // current directory if it is going to be processed.
  static String theImgFile = "ImgMod04aTest.jpg";

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

  //Reference to the image processing object.
  ImgIntfc04a imageProcessingObject;
  //-----------------------------------------------------//

  public static void main(String[] args){
    //Get names for the image processing class and the
    // image file to be processed.  Program supports 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 == 1){
      theProcessingClass = args[0];
```

```
      //Use default image file along with this class.
    }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.
    ImgMod04a obj = new ImgMod04a();
  }//end main
  //--------------------------------------//

  public ImgMod04a(){//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.  This local
    // variable mustbe declared final because it is
    // accessed from within an anonymous inner class.
    final double[][][] threeDPix = 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 so as 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.
    int inTop = this.getInsets().top;
    int inLeft = this.getInsets().left;
    int buttonHeight = replotButton.getSize().height;

    //Size the frame so that a small amount of yellow
    // background will show on the right, between the
    // images, and on the bottom when both images are
    // displayed, one above the other.
    this.setSize(2*inLeft+imgCols + 2,inTop
                          + buttonHeight + 2*imgRows + 8);

    //================================================//
    //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){
          //Pass a 3D array of pixel data to the
          // processing method.  This method returns one 3D
          // array of pixel data.
          //The contents of output array are displayed as
          // the second image in the display.
          //The contents of the output array are also
          // written into an output JPEG file named
          // junk.jpg.
          double[][][] threeDPixModA =
               imageProcessingObject.processImg(threeDPix);
          //Now prepare the pixel array for display.

          //Convert the 3D array of modified pixel data of
          // type double to a 1D array of pixel data of
          // type int.  This 1D array is in a standard
          // pixel format.  See a brief description of the
          // format in the comments in the method named
          // convertTo1D.
          int[] oneDPixA = convertTo1D(threeDPixModA);
          //Use the createImage() method of the Component
          // class to create a new Image object from the
          // 1D array of pixel data.  Note that
          // MemoryImageSource implements the
          // ImageProducer interface and therefore
          // satisfies one of the overloaded versions of
          // the createImage method.
          modImg = createImage(new MemoryImageSource(
                      imgCols,imgRows,oneDPixA,0,imgCols));

          //Repaint the image display frame with the
          // original image at the top and the modified
          // image at the bottom.
          display.repaint();

          //Write the modified image into a JPEG
          // file named junk.jpg.
          writeJpegFile(modImg,imgCols,imgRows);


        }//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 = (ImgIntfc04a)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, 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 constructor
  //=====================================================//

  //Inner class for canvas object on which to display the
  // two images.
  class Display extends Canvas{
    //Override the paint method to display the rawImg 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((rawImg != null) &&
           (modImg != null)){
          g.drawImage(rawImg,0,0,this);
          g.drawImage(modImg,0,imgRows + 2,this);
        }//end if
      }//end if
    }//end paint()
  }//end class myCanvas
//=========================================================//

  //Save pixel values as type double to make
  // arithmetic easier later.

  //The purpose of this method is to convert the data in
  // the int oneDPix array into a 3D array of type double.
  //The dimensions of the 3D array are row, col, and color
  // in that order.
  //Row and col correspond to the rows and columns of the
  // pixels in the image.  Color corresponds to
  // transparency and color information at the following
  // index levels in the third dimension:
  // 0 alpha (tranparency)
  // 1 red
  // 2 green
  // 3 blue
  // The structure of this code is determined by the way
  // that the pixel data is formatted into the 1D array of
  // pixel data of type int when the image file is read.
  double[][][] convertTo3D(
                    int[] oneDPix,int imgCols,int imgRows){
    //Create the new 3D array to be populated with pixwl
    // data.
    double[][][] data = new double[imgRows][imgCols][4];

    for(int row = 0;row < imgRows;row++){
      //Extract a row of pixel data into a temporary array
      // of ints
      int[] aRow = new int[imgCols];
      for(int col = 0; col < imgCols;col++){
        int element = row * imgCols + col;
        aRow[col] = oneDPix[element];
      }//end for loop on col

      //Move the data into the 3D array.  Note the use of
      // bitwise AND and bitwise right shift operations to
      // mask all but the correct set of eight bits.
      for(int col = 0;col < imgCols;col++){
        //Alpha data
        data[row][col][0] = (aRow[col] >> 24) & 0xFF;
        //Red data
```

```
        data[row][col][1] = (aRow[col] >> 16) & 0xFF;
        //Green data
        data[row][col][2] = (aRow[col] >> 8)  & 0xFF;
        //Blue data
        data[row][col][3] = (aRow[col]) & 0xFF;
      }//end for loop on col
    }//end for loop on row
    return data;
  }//end convertTo3D
  //-------------------------------------------------//

  //The purpose of this method is to convert the data in
  // the 3D array of type double back into the a 1d array
  // of type int.  This is the reverse of the method named
  // convertTo3D.  Note that the data values are clamped at
  // 0 and 255 before casting to type int.
  int[] convertTo1D(double[][][] data){
    int imgRows = data.length;
    int imgCols = data[0].length;

    //Create the 1D array of type int to be populated with
    // pixel data, one int value per pixel, with four
    // color and alpha bytes per int value.
    int[] oneDPix = new int[imgCols * imgRows * 4];

    //Move the data into the 1D array.  Note the use of the
    // bitwise OR operator and the bitwise left-shift
    // operators to put the four 8-bit bytes into each int.
    // Also note that the values are clamped at 0 and 255.
    for(int row = 0,cnt = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        if(data[row][col][0] < 0) data[row][col][0] = 0;
        if(data[row][col][0] > 255) data[row][col][0]=255;
        if(data[row][col][1] < 0) data[row][col][1] = 0;
        if(data[row][col][1] > 255) data[row][col][1]=255;
        if(data[row][col][2] < 0) data[row][col][2] = 0;
        if(data[row][col][2] > 255) data[row][col][2]=255;
        if(data[row][col][3] < 0) data[row][col][3] = 0;
        if(data[row][col][3] > 255) data[row][col][3]=255;

        oneDPix[cnt] = (((int)data[row][col][0] << 24)
                                          & 0xFF000000)
                     | (((int)data[row][col][1] << 16)
                                          & 0x00FF0000)
                      | (((int)data[row][col][2] << 8)
                                          & 0x0000FF00)
                          | (((int)data[row][col][3])
                                          & 0x000000FF);
        cnt++;
      }//end for loop on col
    }//end for loop on row

    return oneDPix;
  }//end convertTo1D
  //-------------------------------------------------//
```

```
  //Write the contents of an Image object to a JPEG file
  // named junk.jpg.
  void writeJpegFile(Image img,int width,int height){
    //Create an off-screen drawable image by calling the
    // createImage method of the Component class.  Cast
    // it to type BufferedImage.
    BufferedImage bufferedImg =
                 (BufferedImage)createImage(width,height);

    //Call the createGraphics method of the BufferedImage
    // class to create a Graphics2D object, which can be
    // used to draw into the BufferedImage object.
    Graphics2D bufferedGraphics =
                             bufferedImg.createGraphics();

    //Call the drawImage method of the Graphics class to
    // draw the image into the off-screen buffer.  Pass
    // null as the ImageObserver.
    bufferedGraphics.drawImage(img,0,0,null);

    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(bufferedImg,"jpeg",outStream);
      outStream.close();
    }catch (Exception e) {
      e.printStackTrace();
    }//end catch
  }//end writeJpegFile
  //-------------------------------------------------//

  //This method reads an image from a specified image file
  // and converts the pixel data into a 3D array of type
  // double.  The name of the image file is specified as
  // a String by theImgFile.
  double[][][] getTheImage(){
    rawImg = Toolkit.getDefaultToolkit().
                                      getImage(theImgFile);

    //Use a MediaTracker object to block until the image is
    // loaded or ten seconds has elapsed.
    tracker = new MediaTracker(this);
    tracker.addImage(rawImg,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

    //Raw image has been loaded.  Get width and height of
    // the raw image.
    imgCols = rawImg.getWidth(this);
    imgRows = rawImg.getHeight(this);

    //Create a 1D array object to receive the pixel
    // representation of the image
    int[] oneDPix = new int[imgCols * imgRows];

    //Create an empty BufferedImage object
    BufferedImage buffImage = new BufferedImage(
                              imgCols,
                              imgRows,
                              BufferedImage.TYPE_INT_ARGB);

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

    //Convert the BufferedImage to numeric pixel
    // representation.
    DataBufferInt dataBufferInt =
                      (DataBufferInt)buffImage.getRaster().
                                              getDataBuffer();
    oneDPix = dataBufferInt.getData();

    //Convert the pixel byte data in the 1D array to
    // double data in a 3D array to make it easier to work
    // with the pixel data later.  Recall that pixel data
    // is unsigned byte data and Java does not support
    // unsigned arithmetic. Performing unsigned arithmetic
    // on byte data is particularly cumbersome.
    return convertTo3D(oneDPix,imgCols,imgRows);
  }//end getTheImage
  //-----------------------------------------------------//
}//end ImgMod04a.java class
//=======================================================//

//The ProgramTestA 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 class program named ImgMod04a and the
// interface named ImgIntfc04a.

//The constructor for the class displays a small frame on
```

```
// the screen with a single textfield. The purpose of the
// text field is to allow the user to enter a value that
// represents the slope of a line.  In operation, the user
// types a value into the text field and then clicks the
// Replot button on the main image display frame.  The user
// is not required to press the Enter key after typing the
// new value, but it doesn't do any harm to do so.
// Negative slopes are not supported.  An attempt to use
// a negative slope will cause the program to abort with
// an error.

//The method named processImage receives a 3D array of
// type double containing alpha, red, green, and blue
// values for an image.

// The 3D array that is received by processImg is
// modified my the method to cause a white diagonal line to
// be drawn down and to the right from the upper left
// corner of the image.  The slope of the line is
// controlled by the value in the text field.  Initially,
// this value is 1.0, but the value can be modified by the
// user.  (If the characters in the text field cannot be
// converted to a numeric type double, the program will
// abort with an error.)
//After drawing the line, the method inverts the colors in
// the image.  This results in a black line on an image
// with inverted colors.

//The method named processImg is required to return one
// reference to a 3D array objects of type double

//To cause a new line to be drawn, type a new slope
// value into the text field and click the Replot button
// at the top of the main image display frame.

//This class extends Frame.  However, a compatible class is
// not required to extend the Frame class. This example
// extends Frame because it provides a GUI for user data
// input.

//A compatible class is required to implement the
// interface named ImgIntfc04a.

class ProgramTestA extends Frame implements ImgIntfc04a{

  double slope;//Controls the slope of the line
  String inputData;//Obtained via the TextField
  TextField inputField;//Reference to TextField

  //Constructor must take no parameters
  ProgramTestA(){
    //Create and display the user-input GUI.
    setLayout(new FlowLayout());

    Label instructions = new Label(
                         "Type a slope value and Replot.");
```

```
    add(instructions);

    inputField = new TextField("1.0",5);
    add(inputField);

    setTitle("Copyright 2006, Baldwin");
    setBounds(400,0,200,100);
    setVisible(true);
  }//end constructor
  //-----------------------------------------------------//

  //The following method must be defined to implement the
  // ImgIntfc04a interface.
  public double[][][] processImg(double[][][] threeDPix){

    //Determine number of rows and cols
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    //Display some interesting information
    System.out.println("Program test");
    System.out.println("Width = " + imgCols);
    System.out.println("Height = " + imgRows);
    //Make a working copy of the 3D array to avoid making
    // permanent changes to the image data.
    double[][][] temp3D = copy3DArray(threeDPix);
    //Get slope value from the TextField
    slope = Double.parseDouble(inputField.getText());

    //Draw a white diagonal line on the image.
    for(int col = 0;col < imgCols;col++){
      int row = (int)(slope * col);
      if(row > imgRows -1) break;
      //Set values for alpha, red, green, and
      // blue colors.
      temp3D[row][col][0] = 255.0;
      temp3D[row][col][1] = 255.0;
      temp3D[row][col][2] = 255.0;
      temp3D[row][col][3] = 255.0;

    }//end for loop
    //Invert the colors.  Do not invert the alpha value.
    invertColors(temp3D);
    //Return two references to the same array of image
    // data.
    return temp3D;
  }//end processImg
  //-----------------------------------------------------//

  //This method copies a double version of a 3D pixel array
  // to an new pixel array of type double.
  double[][][] copy3DArray(double[][][] threeDPix){
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    double[][][] new3D = new double[imgRows][imgCols][4];
```

```
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        new3D[row][col][0] = threeDPix[row][col][0];
        new3D[row][col][1] = threeDPix[row][col][1];
        new3D[row][col][2] = threeDPix[row][col][2];
        new3D[row][col][3] = threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop
    return new3D;
  }//end copy3DArray
  //-----------------------------------------------------//

  //This method inverts the colors in a double version of a
  // 3D array of pixel data.  It doesn't invert the alpha
  // value.
  void invertColors(double[][][] data3D){
    int imgRows = data3D.length;
    int imgCols = data3D[0].length;
    //Invert the colors.  Don't invert the alpha value.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        data3D[row][col][1] = 255 - data3D[row][col][1];
        data3D[row][col][2] = 255 - data3D[row][col][2];
        data3D[row][col][3] = 255 - data3D[row][col][3];
      }//end inner loop
    }//end outer loop
  }//end invertColors
  //-----------------------------------------------------//
}//end class ProgramTestA

Listing 30
```

Listing 31

```
/*File ImgIntfc04a.java
Copyright 2006, R.G.Baldwin

This is a modification to the interface named ImgIntfc04
that eliminates the use of a second image-processing
method. Otherwise, it is identical to the interface named
ImgIntfc04.

The purpose of this interface is to declare the method
required by image processing classes that are
compatible with the program named ImgMod04a.java.

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

interface ImgIntfc04a{
double[][][] processImg(double[][][] input);
}//end ImgIntfc04a
//=========================================================//

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 image pixel framework

-end-