# Redeye Correction in Digital Photographs

*Learn how to write a Java program that can be used to correct for redeye problems in digital photographs.*

**Published:** March 19, 2009
**By Richard G. Baldwin**

Java Programming Notes # 363

# Preface

## General

This lesson is the next in a series *(see Resources)* designed to teach you how to write Java programs to do things like:

- Edit the color of your digital photos.
- Create a lasso to isolate pixels by dragging the mouse in an image.
- Blur, soften, or sharpen your digital photos.
- Remove *redeye* from your digital photos.
- Distort the human voice.
- Display one image inside another image.
- Do edge detection, blurring, and other filtering operations on images.
- Insert animated cartoon characters into videos of live humans.

If you have ever wondered how to do these things, you've come to the right place.

## What you have learned from earlier lessons

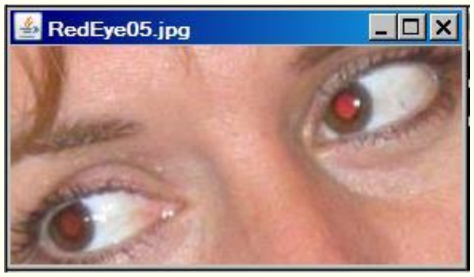
If you have studied the earlier lessons in this series, among other things, you have learned:

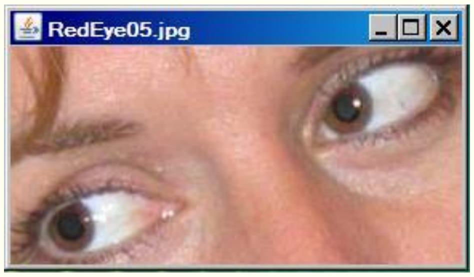
- How to download, install, and test a Java multimedia library from Georgia Institute of Technology.
- How to edit the pixels in an image on a pixel-by-pixel basis using a program written entirely in Java.
- About the HSB color model and how to use that model to adjust the hue, saturation, and brightness of your digital photos.
- Many aspects of image processing, including color distance, projecting 3D coordinates onto a 2D display plane, and edge detection.
- How to write an animated flocking program.
- How to sharpen or soften your digital photos.
- How to control temperature and tint in a digital photograph.

## What you will learn in this lesson

You will learn how to write the code to correct for redeye problems in digital photographs. For example, the top image in Figure 1 shows a portion of a digital photograph with a serious redeye problem. The bottom image in Figure 1 shows the results obtained by using this program to correct the redeye problem.

**Figure 1. Sample results from the redeye correction program**.





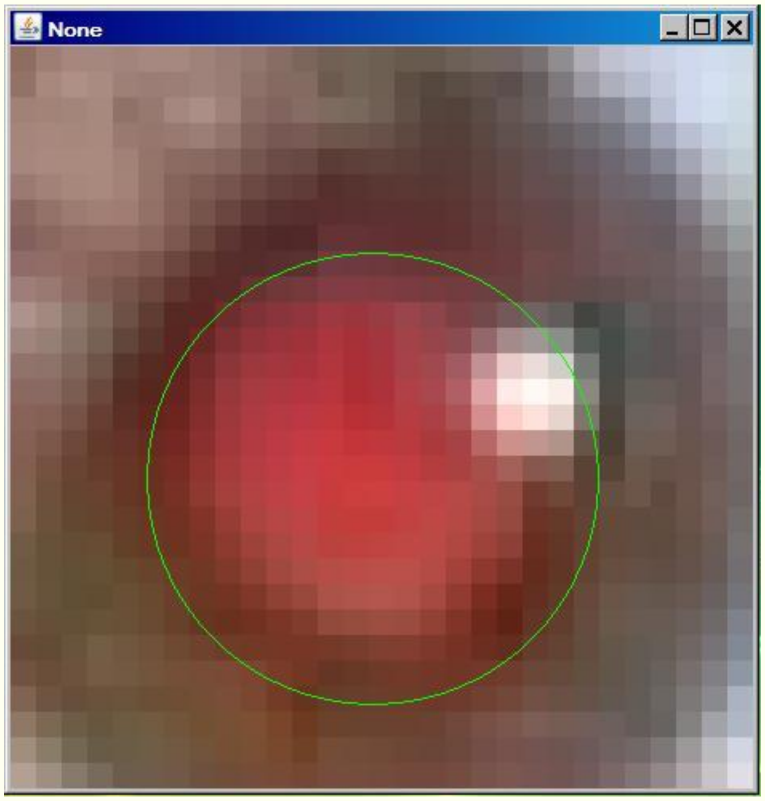
## Other things that you will learn

In addition to learning how to correct for redeye problems in digital photographs, you will also learn several important Java programming techniques that were employed to make this an effective program that is easy to use.

## Enlarging an image and identifying problem pixels

For example, one of the most important techniques used in this program allows the user to enlarge the image of the eye and to place a circular lasso around the offending redeye pixels to help identify the pixels that will be considered for a color change. Figure 2 shows a screen shot captured from this step in the process.

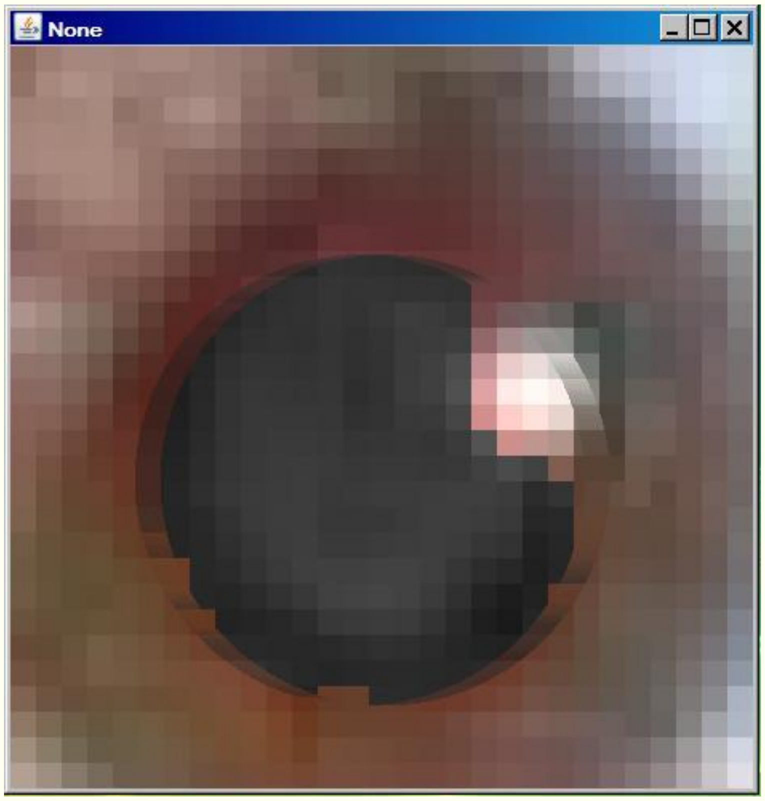
**Figure 2. A lasso surrounding offending pixels in an enlarged version of the eye.**



## Correcting the problem pixels

Figure 3 shows the results of applying redeye decision and correction algorithms to the pixels inside the lasso shown in Figure 2.

**Figure 3. Results of applying redeye decision and correction algorithms to pixels inside the lasso.**



## Preservation of highlights

As you can see in Figure 3, the redeye correction algorithm preserves some of the highlights in the original image and softens the edges of the color-corrected area to avoid a "blank stare" look in the finished product.

## Merging two images

After the redeye correction is performed on the enlarged version of the eye, the modified image is reduced back to its normal size and merged back into the main display producing results similar to those shown in the bottom image of Figure 1.

## Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

### Figures

- Figure 1. Sample results from the redeye correction program.
- Figure 2. A lasso surrounding offending pixels in an enlarged version of the eye.

## Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com.

# General background information

## A multimedia class library

In this series of lessons, I will present and explain many of the classes in a multimedia class library that was developed and released under a **Creative Commons Attribution 3.0 United States License** *(see Resources)* by Mark Guzdial and Barbara Ericson at Georgia Institute of Technology.  In doing this, I will also present some interesting sample programs that use the library.

## Software installation and testing

I explained how to download, install, and test the multimedia class library in an earlier lesson titled *Multimedia Programming with Java, Getting Started (see Resources)*.

The program that I will explain in this lesson requires access to that class library.

## Redeye correction

Although there are a large number of products available for performing redeye correction on digital photographs, this technology is far from being completely settled.

### A major problem related to the size of the camera

The redeye problem is a major problem for the manufacturers of small, relatively inexpensive digital cameras where the flash is positioned physically close to the optics.  The problem is not one of resolution, but rather is one of size.  When the flash is very close to the optics, *(as is almost always the case with a small camera)*, the reflection of the flash from the retina at the back of the eyeball comes straight back into the optics.

### A red filter

Apparently the retina has the characteristic of filtering out colors other than red in the reflected light.  This is what causes the redeye problem on the millions of digital photographs that are taken daily on small digital cameras.

### Jockeying for the best patent

A quick search of the web indicates that individuals and companies are jockeying for position trying to nail down a patent on the best way to solve the problem.  This includes not only software solutions, but also hardware solutions associated with the camera itself.

### An ideal solution

A hardware solution at the camera would be the best solution.  Lacking that, in the ideal case for a software solution, you would simply present your digital photographs containing images of people to a software program and it would do a perfect job of first identifying and then correcting all of the redeye problems in the photographs.  Few, if any current products provide that ideal capability.

## Identification of redeye problems

There are software products that attempt to identify redeye problems in a digital photo.  These products must first discriminate against false positives such as red buttons on a dress, or red berries on a holly wreath.  *(You wouldn't be happy with a product that changes the color of the berries on the holly wreaths in your Christmas pictures from red to some other color.)* Then they must correct the problems that are identified.

## Pattern recognition software

Such programs make use of facial recognition or other pattern-recognition algorithms to separate the actual redeye problems from the false positives.  Sometimes they are successful and sometimes they are not successful.  Most of these programs fall back on the human user to identify the redeye problems when the automated algorithm is unable to do so.  *(The human brain has a fantastic pattern recognition capability.)*

## Program will depend on the user

The program that I will explain in this lesson makes no attempt to automatically identify the redeye problems in a digital photo.  Instead, this program depends entirely on the user to identify the areas containing offending redeye pixels.

## Decision and correction algorithms

Once the user has identified the general area containing offending pixels using a circular lasso as shown in Figure 2, this program applies decision and correction algorithms that attempt to exclude any non-offending pixels and to correct the color of the remaining pixels inside the lasso.  I will usually refer to these algorithms hereafter simply as the *decision algorithm* and the *correction algorithm*.

## A difficult task

Even after the user has identified a group of potentially offending pixels, deciding which pixels are actually redeye pixels with a high degree of certainty is a difficult task.

If you carefully examine a large number of digital photos with redeye problems, you will find that the colors of offending pixels often overlap the colors of normal body parts in the vicinity of the eye.

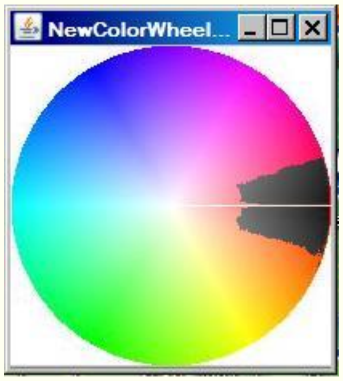
## Use of HSB for discrimination

The identification of a pixel as an offending redeye pixel in this program is made on the basis of the hue, saturation, and brightness value of the pixel.  Not only does this approach do a pretty good job, it also gives you another opportunity to work with the HSB color model.  *(See Resources for earlier lessons involving the HSB color model.)*

## Threshold constants

Constants in the program establish the allowable range for each of the HSB parameters that are interpreted to represent a redeye pixel color.  I have experimented with different values for different images and have come up with a set of values that seem to work well for most digital photos.  The set of HSB parameter values that I have settled on can be described by the image of the color wheel in Figure 4.

**Figure 4. Decision and correction algorithms applied to a color wheel.**



## Interpreting the color wheel results

What you are seeing in Figure 4 is the top surface of an HSB color cone.  *(See the earlier lesson titled The HSB Color Model in Resources for an explanation of the HSB color cone)*.  The gray area in Figure 4 shows the top surface of a 3D wedge in the cone that extends downward to a brightness value of 0.2.  Any pixel with a color inside the wedge is considered to be an offending redeye pixel.

The parameters that define the shape and position of the wedge are shown in Figure 5.

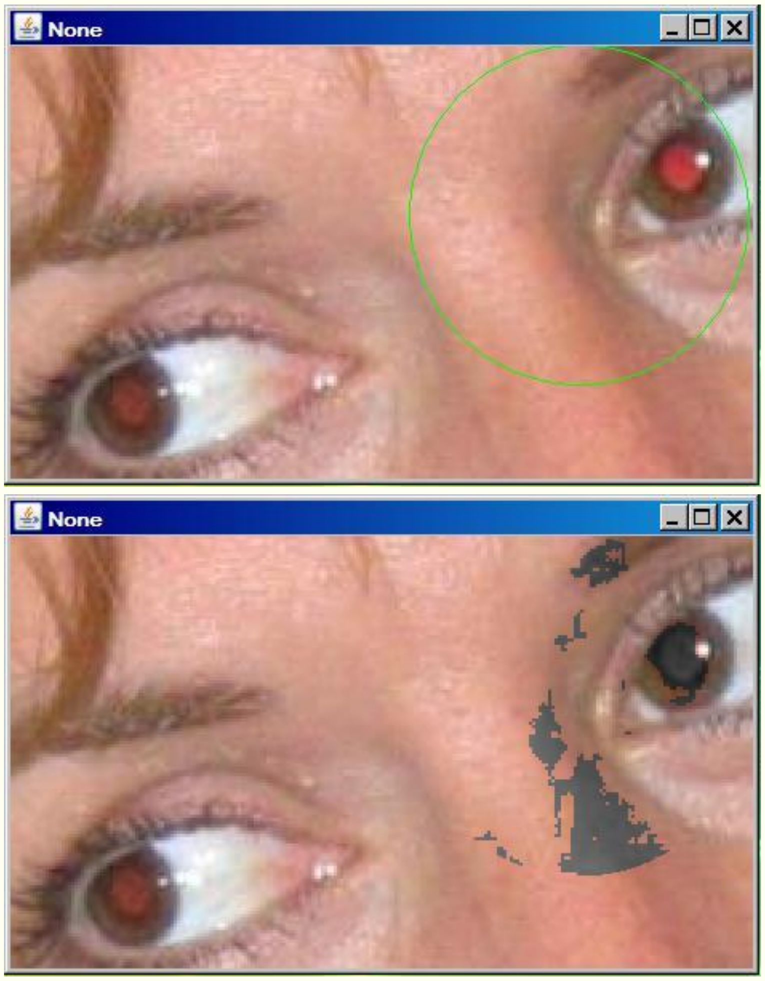
**Figure 5. HSB values that define the redeye color wedge.**

```
    0.95 <= Hue <= 1.05
    Saturation > 0.43
    Brightness > 0.2
```

## Some care is required

Although these parameters work reasonably well for many digital photos, it is still important to avoid including pixels in the lasso that fall outside of the wedge if possible.  For example, the top and bottom images in Figure 6 show what happens when the lasso is allowed to include the iris and some of the skin in the area of the eye.

**Figure 6. Application of decision and correction algorithms to skin near the eye.**



The top image in Figure 6 shows the lasso and the bottom image shows the results of applying the redeye decision and correction algorithms to all of the pixels contained inside of the lasso.

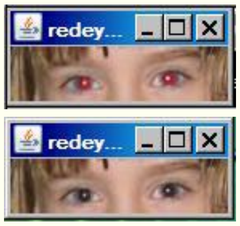
## Color of skin and iris modified

As you can see from the bottom image in Figure 6, some of the pixels in the iris and some of the pixels in the skin were inside the wedge of colors that were deemed to be offending redeye pixels.  Therefore, these pixels were also colored gray in addition to the actual redeye pixels.

## A success story

Figure 7 shows one of many success stories that I could show you where I was able to do a very good job of correcting the redeye problem on a digital photograph.

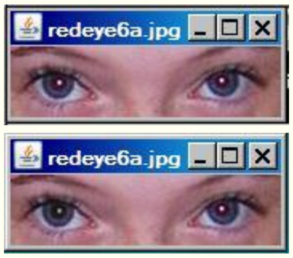
**Figure 7. A success story.**



## Some cases are more difficult

There are some cases, however, that this program can't handle by applying the existing decision algorithm for identifying offending redeye pixels.  One such case is shown in the top image in Figure 8.  Selecting either eye in the top image of Figure 8 and applying the decision and corrections algorithms results in essentially no change in the appearance of the eye.

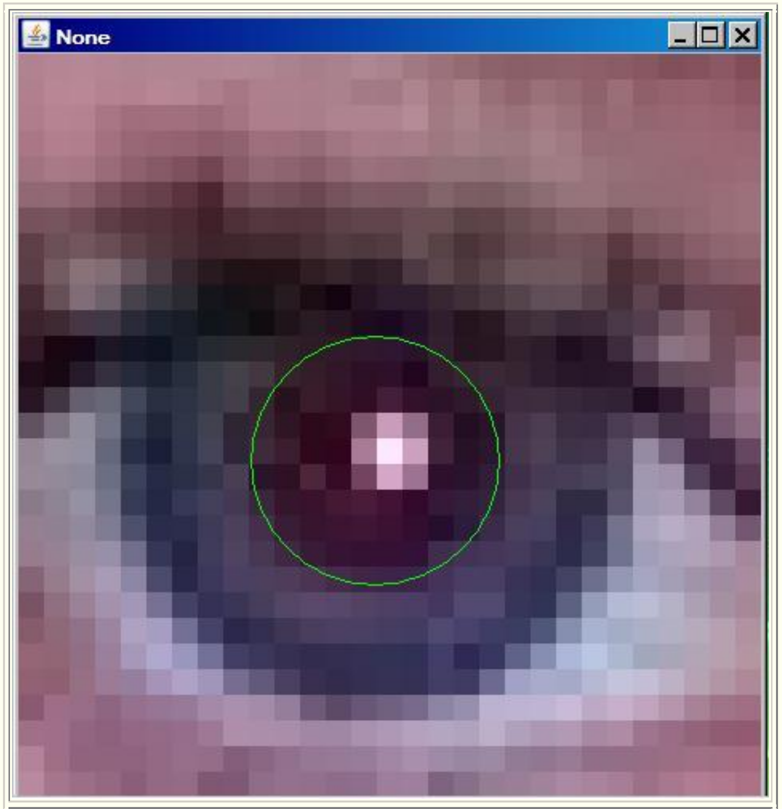
**Figure 8. A difficult case to correct.**

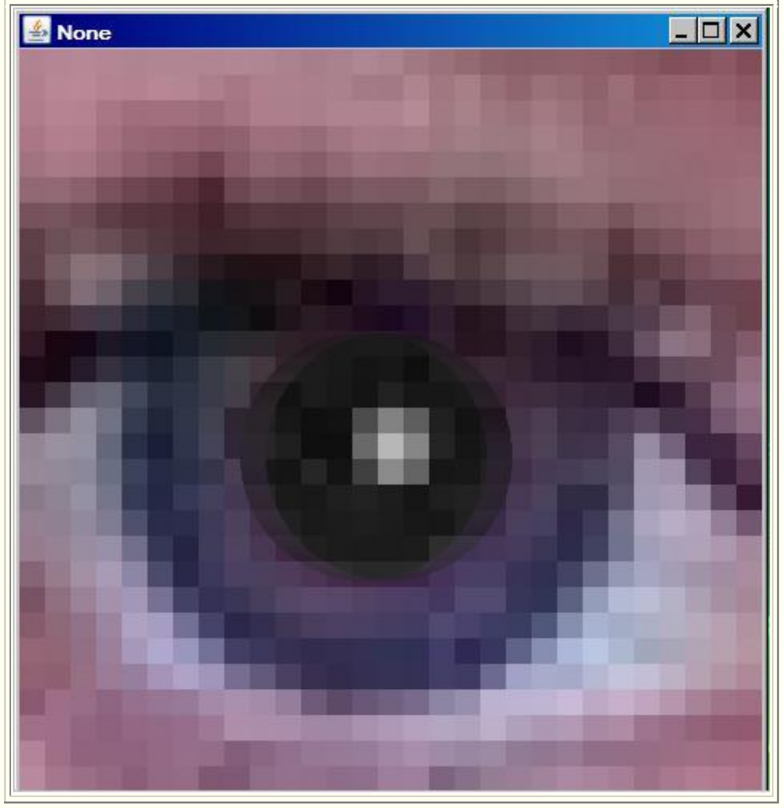


## Enlarged view of the difficult case

The reason for the problem is shown in the top image of Figure 9 where the right eye from Figure 8 has been selected and enlarged by a factor of sixteen.

**Figure 9. Enlarged images from the difficult case.**





Offending redeye pixel colors are outside the wedge

The color of the offending pixels in the top image of Figure 9 is a  deep purple instead of being closer to red.  In fact, the hue value for the offending pixels in this case is in the neighborhood of 0.9, which causes them to be well outside of the wedge shown in the HSB color wheel in Figure 4.  Therefore, when the decision algorithm described by the wedge in Figure 4 is applied to the pixels inside the lasso, they are not deemed to be offending redeye pixels and their colors are not changed.

## Would sliders be a good idea?

At one point, I considered adding three sliders to the GUI and allowing the user to change the threshold values for hue, saturation, and brightness.  This would make it possible for the user to change the shape and position of the wedge shown in Figure 4 on a case by case basis.

## Too complex

On further reflection, however, I decided that would be far too complex.  The average person trying to correct redeye problems in digital photos wouldn't even know what the HSB color model is, much less how to adjust the parameters to control the sensitivity of the decision algorithm.

## An Override button

The solution that I came up with was to place an **Override** radio button on the GUI.  When the user encounters a case that is not amenable to the existing decision algorithm, the user can select the **Override** button to eliminate the application of the decision algorithm.  When that is done, the program will simply assume that every pixel within the lasso is an offending pixel and change the color of the pixels accordingly.

## Experimental results

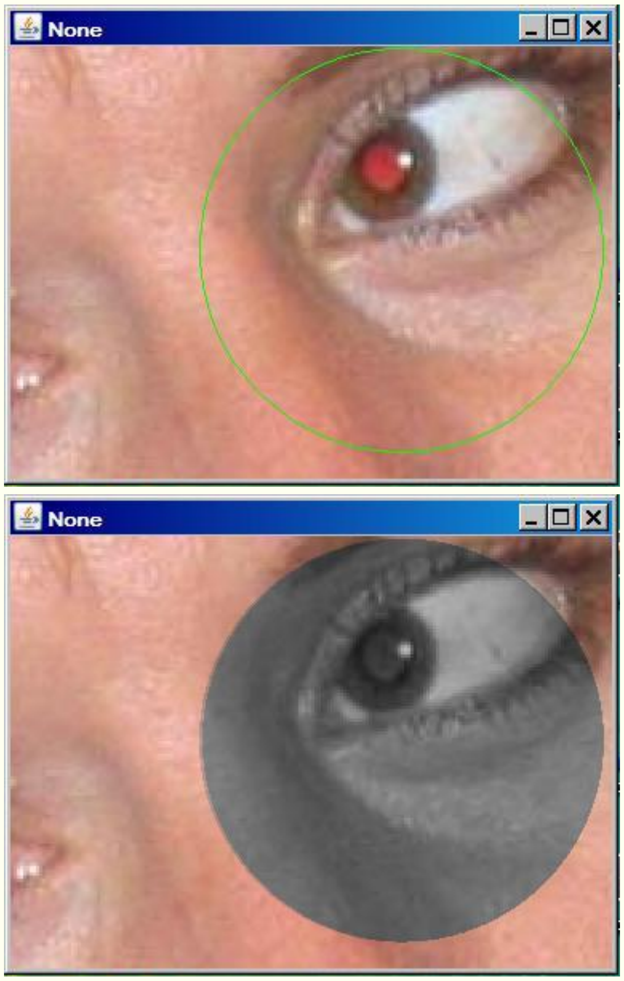
The bottom image in Figure 9 shows the results of selecting the override button before clicking the **Fix Redeye** button after having placed the lasso as shown in the top image in Figure 9.

Clicking the **Recombine Images** button to merge this modified image back into the image shown in the top of Figure 8 produced the results shown in the bottom of Figure 8.  *(Note that only the right eye has had the redeye problem corrected in the bottom image in Figure 8.  The left eye remains the same as before.)*

## A word of caution

Just as a word of caution, the two images in Figure 10 show what happens if the *override* option is used without being careful about the placement of the lasso around the offending redeye pixels.

**Figure 10. Incorrect use of the override option.**



As described earlier, when the override button is selected, the program assumes that every pixel inside the lasso is an offending redeye pixel and changes its color to gray.

## Behavior of the decision algorithm

On the bright side, a comparison of the bottom image in Figure 10 with the bottom image in Figure 6 shows an example of the behavior of the decision algorithm. A large portion of the girl's face was included in the lasso in both cases. The bottom image in Figure 6 shows the results of applying the decision and correction algorithms to each of the pixels within the lasso. In the case of Figure 6, a small percentage of the pixels outside the pupil of the eye were found to fall within the offending pixel color wedge. Those were the only pixels for which the color was changed to gray.

On the other hand, the decision algorithm was overridden in the bottom image in Figure 10, and all of the pixels contained within the lasso were changed to gray.

## A grayscale image

Another interesting aspect of the bottom image in Figure 10 is the appearance of the gray circle. As you can see, when the pixels within the lasso are converted to gray, they are converted to a grayscale image instead of being converted to a solid gray color. This maintains the structural quality of the pupil, including highlights, and avoids the "blank stare" look that would be produced by replacing the color of the offending pixels with a solid gray color.

## The program named RedEye05

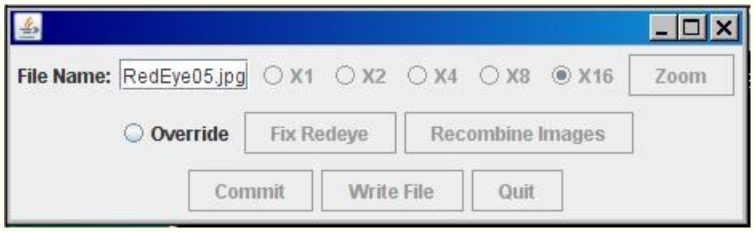
### Introductory remarks

The purpose of this program is to demonstrate a programming technique using Java for correcting redeye problems in digital photographs.

### The graphical user interface *(GUI)*

The program begins by displaying a GUI in the upper left corner of the screen as shown in Figure 11.

**Figure 11. Program GUI at startup.**



### The input image file

At that point, the GUI contains a text field for entry of the name of the image file to be processed and some other user-input components, which are all disabled. *(A default file name appears in the text field mainly for ease of test and debugging.)*

**Browsing for the input file**
If you prefer to browse for the input file, you can easily upgrade the program to provide that capability using the **setMediaPath** and **pickAFile** methods of Ericson's **FileChooser** class.

If the file is in the current directory, only the file name and extension must be entered. Otherwise, the full path, name, and extension for the file must be entered. Files of types jpg, bmp, and png are supported.

### Loading the image from the file

**Multimedia library required**
Ericsson's multimedia library is required to compile and execute this program.

When the user enters the name of the image file into the text field, the file is loaded into a **Picture** object. The **Picture** object is displayed in the upper left corner of the screen as shown by the top image in Figure 1. *(I will often refer to this as the main display in the discussions that follow.)* When the main display appears, the GUI is moved to a location immediately below the main display.

At this point, the text field becomes disabled and several buttons and radio buttons are enabled. *(Different components are enabled and disabled during the running of the program as appropriate.)*

## The overall procedure

The overall procedure for correcting redeye problems in a digital photograph is as follows. Once the image is loaded and displayed as shown by the top image in Figure 1, the user selects a pixel in the eye containing the redeye problem by clicking the mouse in the red area. *(There is no requirement for extreme accuracy in selecting the pixel unless the eye is very small.)*

Then the user selects one of five radio buttons specifying a zoom factor of 1, 2, 4, 8, or 16 and clicks the button labeled **Zoom** in Figure 11. *(A zoom factor of 16 is selected by default at program startup.)*

## An enlarged view of the eye

Clicking the **Zoom** button causes the area surrounding the selected pixel to be enlarged by the specified zoom factor and displayed in another window to the right of the main display object as shown by Figure 2. *(I will often refer to this as the zoomed image or the zoomed picture. The green circular lasso shown in Figure 2 doesn't exist at this point.)*

Clicking the **Zoom** button also causes the **Zoom** button and several other buttons in the GUI to be disabled.

## Surround offending pixels with a lasso

To correct the redeye problem in the selected eye, the user surrounds the offending pixels in the zoomed image with a circular lasso by dragging the mouse in the zoomed image as shown in Figure 2. Then the user clicks the button labeled **Fix Redeye** shown in Figure 11.

## Apply the decision and correction algorithms

Clicking the **Fix Redeye** button causes decision and correction algorithms to be executed that attempt to correct the redeye problem in the area enclosed by the circular lasso.

The decision algorithm scans all of the pixels isolated by the lasso and attempts to exclude any pixels that are not part of the problem. The correction algorithm changes the color of the remaining pixels to a dark gray with highlights preserved as shown in Figure 3.

## Undo the previous operation

Clicking the mouse in the zoomed image at any time will erase an existing lasso and undo the effects of having clicked the **Fix Redeye** button.  Thus, the process of surrounding the offending pixels with the lasso and changing their color can be repeated as many times as necessary until the user is satisfied that the best fit of the circular lasso and the offending pixels has been achieved.

## Merge the two images

Then the user clicks the button labeled **Recombine Images** in Figure 11 to cause the modified zoomed image to be reduced back to its original size and merged back into the main display as shown by the bottom image in Figure 1.

## If the user is not happy...

If the user is not happy with the results at this point, the entire process can be repeated by selecting the same eye again and clicking the **Zoom** button.  This will undo everything back to the most recent **Commit** operation *(see below)*.

## Using the Override button

As I explained earlier, for those cases where the decision algorithm doesn't seem to be capable of selecting the offending pixels inside the lasso, the user can override the decision algorithm by selecting the **Override** button before clicking the **Fix Redeye** button.  In this case, the user should be very careful to avoid including pixels from the eyelid, the iris, etc., in the lasso because the color of all of the pixels inside the lasso will be changed to gray as shown in Figure 10.

## Committing the main display

Once the user is happy with the results for the eye currently being processed, the user can accept the results by clicking the **Commit** button shown in Figure 11.  Then the user can perform the same procedure on any other eyes in the photograph that exhibit redeye problems.

Once committed, the pixels are permanently changed in a backup image that is held in memory and the changes cannot be undone.  However, at no time does this program modify the original image file.

## Creating a circular lasso

As explained earlier, the user can drag the mouse in the zoomed image to create a circular lasso. An anchor point is established when the user first presses the mouse button to begin the drag operation. The lasso can be created in any direction from the anchor point. The diameter of the lasso is equal to the distance from the mouse pointer to the anchor point. The final size and position of the lasso are established when the user releases the mouse button to end the drag operation.

Dragging the mouse outside the bounds of the image causes the size of the lasso to continue to grow. However, if the lasso extends outside the bounds of the zoomed image, clicking the **Fix Redeye** button will have no effect.

## Erasing an existing lasso

Once the drag operation is completed, the lasso remains on the screen until the user:

- Clicks the zoomed image again with the mouse,
- Clicks the **Fix Redeye** button,
- Clicks the **Recombine Images** button, or
- Does something else to cause the image to be repainted such as minimizing and then restoring the zoomed image.

## Writing backup files

Clicking the **Write** button shown in Figure 11 *(when it is enabled)* causes a backup bmp file of the main display to be written into the same directory from which the image file was read.

The five most recent backup files are saved. The names of the backup files are the same as the name of the original image file except that the characters BAKn are inserted immediately before the extension. The character n is replaced by a digit from 0 through 4.

## Terminating and writing a final output file

The large X buttons in the upper right corners of both image displays are disabled. Clicking them does nothing. The program can be terminated by clicking the **Quit** button shown in Figure 11 *(when it is enabled)* or by clicking the large X in the upper right corner of the GUI at any time.

Before terminating, the program writes an output file containing the final state of the main display in the same format as the input file. The file is written into the folder from which the original image file was read. The name of the output file is the same as the name of the input file except that the word FINAL is inserted immediately before the extension.

## Will explain in fragments

As is my custom, I will explain this program by breaking it down into code fragments and explaining the fragments.  A complete listing of the program is provided in Listing 28 near the end of the lesson.

## The constructor

The class named **RedEye05** begins by declaring a large number of instance variables and defining a very simple **main** method that simply instantiates an object of the class.  There is nothing remarkable about that code, so I won't bother showing it in code fragments.  You can view that code in Listing 28 near the end of the lesson.

### Beginning of the constructor

The constructor begins in Listing 1.

**Listing 1. Beginning of the constructor.**

```
  public RedEye05(){//constructor

    //All close operations are handled in a
WindowListener
    // object.
    setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);

    //Construct the GUI. Components are
arranged in the
    // GUI from left to right, top to bottom
in
    // approximately the order that they are
used.
    mainPanel.setLayout(new BorderLayout());
//Code deleted for brevity.

    //Add components to the north panel.
    northPanel.add(fileNameLabel);
//Code deleted for brevity.

    //Add components to the center panel
    centerPanel.add(overrideButton);
//Code deleted for brevity.

    //Add components to the south panel.
    southPanel.add(commitButton);
//Code deleted for brevity.

    //Disable the buttons until the user
enters the file
    // name.
    zoom1.setEnabled(false);
//Code deleted for brevity.
```

```
    //Set the size of the GUI and display it
in the upper
    // left corner of the screen. It will be
moved later
    // to a position immediately below the
display of the
    // picture.
    getContentPane().add(mainPanel);
    pack();
    setVisible(true);

    //Request that the focus move to the text
field where
    // the file name is to be entered.
    fileNameField.requestFocus();
```

Much of the code in the early part of the constructor involves the physical construction of the GUI shown in Figure 11.  This code is somewhat boring so I deleted much of it from Listing 1 for brevity.  You can view the deleted code in Listing 28.

The remaining code in Listing 1 is straightforward, so there should be no need to provide an explanation for that code beyond the embedded comments.

## Create listener objects

The constructor starts to be more interesting in Listing 2 where the constructor starts defining anonymous listener classes and registering objects of those classes on the various components in the GUI and the displays.

### Register a listener object on the text field

Listing 2 shows the beginning of the code that registers a listener object of type **ActionListener** on the text field shown in Figure 11.

When the user enters the file name in the text field, the **actionPerformed** method of the listener object is called.  This method sets everything up properly so that the program will function as an event-driven picture-manipulation program until the user clicks the large X in the upper-right corner of the GUI.

**Listing 2. Register a listener object on the text field.**

```
    fileNameField.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          //Disable the text field and its
label to
```

```
          // prevent the user from entering
anything else
          // into it and causing it to fire
another event.
          fileNameField.setEnabled(false);
          fileNameLabel.setEnabled(false);

          //Get the file name from the text
field and use
          // it to create a new Picture
object.
          fileName = fileNameField.getText();
          backupPicture = new
Picture(fileName);

          //Get information that will be used
to write the
          // output files.
          String inputPath = new
File(fileName).

getAbsolutePath();
          int posDot =
inputPath.lastIndexOf('.');
          outputPath =
inputPath.substring(0,posDot);

          //Write the first copy of the output
backup
          // file before any processing is
done.
          backupPicture.write(outputPath
                       + "BAK" +
writeCounter++ + ".bmp");

          //Get filename extension. It will be
used later
          // to write the final output file.
          extension =
inputPath.substring(posDot);

          //Decorate the GUI.
          setTitle("Copyright 2009,
R.G.Baldwin");
```

The code in Listing 2 is straightforward and should not require an explanation beyond the embedded comments.

## Create and display the picture

The **actionPerformed** method continues in Listing 3 where it creates and displays the picture as shown by the top image in Figure 1.

**Listing 3. Create and display the picture.**

```
          //Create the picture that will be
used for
          // processing.
          //Note that the original image file
is not
          // modified by this program.
          displayPicture = new
Picture(backupPicture);

          //Display the picture.
          displayPicture.show();

          //Save a reference to the image.
Also save the
          // width and height of the picture.
          theImage =

(BufferedImage)(backupPicture.getImage());
          pictureWidth =
backupPicture.getWidth();
          pictureHeight =
backupPicture.getHeight();

          //Get and save a reference to the
JFrame object
          // that contains the image.
          displayPictureFrame =

displayPicture.getPictureFrame().frame;

          //Get and save the insets for the
JFrame object.
          leftInset =

displayPictureFrame.getInsets().left;
          topInset =
displayPictureFrame.getInsets().top;
          rightInset =

displayPictureFrame.getInsets().right;
          bottomInset =

displayPictureFrame.getInsets().bottom;
```

Listing 3 also gets and saves some miscellaneous values that will be used later in the program.

## Resize and move the GUI

Strictly for cosmetic purposes, Listing 4 begins by adjusting the width of the GUI to match the width of the main display if possible.  Then Listing 4 relocates the GUI to a position immediately below the main display so that it won't be covered up by the display.

**Listing 4. Resize and move the GUI.**

```
          pack();
          int packedHeight = getHeight();
          int packedWidth = getWidth();
          if((pictureWidth + 7) >= 
packedWidth){
            //Make the width of the GUI the 
same as the
            // width of the display.
            setSize(pictureWidth + 
7,packedHeight);
          }//Else, just leave the GUI at its 
current size.
          //Put the GUI in its new location 
immediately
          // below the display.
          setLocation(0,pictureHeight + 30);

          //Enable user input controls.
          zoom1.setEnabled(true);
          zoom2.setEnabled(true);
          zoom4.setEnabled(true);
          zoom8.setEnabled(true);
          zoom16.setEnabled(true);
          zoomButton.setEnabled(true);
          commitButton.setEnabled(true);
          writeButton.setEnabled(true);
          quitButton.setEnabled(true);

          //Disable the X-button on the 
display.

displayPictureFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);
```

Listing 4 also performs some other routine tasks that are explained by the embedded comments.

**Register a listener object on the picture frame**

In order to enlarge the image in the area of a pixel selected by the user, the program needs a way for the user to select a pixel.  The normal way of doing this is to have the user press a mouse button in the picture and to record the coordinates of the mouse pointer when the picture fires a mouse event.  Many of the other listener objects in this program are created by code blocks in the constructor.  However, the picture hasn't been loaded when the constructor executes, so it isn't possible to register a mouse listener on the picture at that point in time.

**Defined inside of the actionPerformed method**

The code in Listing 5 is defined inside the **actionPerformed** method of the **ActionListener** object that is registered on the text field shown in Figure 11. This code defines an anonymous class that implements the **MouseListener** interface and registers an object of that class on the frame containing the picture after:

1. The user enters the file name in the text field.
2. The image from the file has been loaded.
3. The **Picture** object has been created.

**Listing 5. Register a listener object on the picture frame.**

```
displayPictureFrame.addMouseListener(
            new MouseAdapter(){
              public void 
mousePressed(MouseEvent e){
                //Draw a new copy of the image
                graphics = 
displayPicture.getGraphics();
                graphics.drawImage(

backupPicture.getImage(),0,0,null);
                displayPicture.repaint();

                //Prepare some working 
variables.
                anchorX = e.getX();
                anchorY = e.getY();
                //Draw a small white cursor to 
mark the
                // location of the mouse press 
on the
                // image.
                drawCursor();
              }//end mousePressed
            }//end new MouseAdapter
          );//end addMouseListener
          //----------------------------------
----------//

        //Now finish defining the action 
listener that is
        // registered on the text field.
        }//end actionPerformed
      }//end new ActionListener
    );//end addActionListener
```

## Behavior of the mousePressed method

The **mousePressed** method in Listing 5 begins by drawing the backup copy of the image on the main display each time the user clicks the main display with the mouse.  This creates a fresh copy of the image in the main display.  *(Note however that the image being copied may contain changes made earlier when the user clicked the **Commit** button.)*

## Prepare some working variables

Then Listing 5 calls the **getX()** method and the **getY()** method on the **MouseEvent** object to get and save the location of the mouse pointer when the event was fired.

## A word of caution

It is important to note, however, that the coordinate values returned by these two methods are relative to an origin at the upper left corner of the frame containing the picture.  Therefore, they don't represent the coordinates of the pixel selected by the user relative to the upper-left corner of the image.  Instead, the values must later be adjusted by the *top* and *left* inset values in order to represent the actual coordinates of the pixel.

## Draw a small white cursor

Then Listing 5 calls the **drawCursor** method to draw a small white cursor to mark the/ location of the mouse press on the image.  The **drawCursor** method is completely straightforward so I won't bore you with an explanation.  You can view the code for the method in Listing 28.

## Finish defining the ActionListener class

That signals the end of the definition and instantiation of an anonymous **MouseListener** object on the frame containing the picture.

Listing 5 continues by completing the definition and instantiation of the anonymous **MouseListener** object being registered on the text field in Figure 11.

## Register an ActionListener object on the Zoom button

That brings us back to the constructor where Listing 6 defines an anonymous class that implements the **ActionListener** interface and registers an object of that class on the **Zoom** button shown in Figure 11.

**Listing 6. Register an ActionListener object on the Zoom button.**

```
    zoomButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          //Get a fresh image in order to
prevent the
```

```
          // current cursor from showing in
the zoomed
          // version.
          //Note however that if the user
previously
          // clicked the Commit button when a
cursor was
          // showing in the picture, that
cursor will have
          // been permanently written into the
backup
          // picture.
          graphics =
displayPicture.getGraphics();
          graphics.drawImage(

backupPicture.getImage(),0,0,null);
          displayPicture.repaint();

          //Call the zoom method to do all the
hard work.
          zoom();
        }//end action performed
      }//end newActionListener
    );//end addActionListener
```

### The actionPerformed method

The **actionPerformed** method shown in Listing 6 is called each time the user clicks the **Zoom** button.

This event-handler method copies the backup picture into the main display and then calls the **zoom** method to produce a zoomed version of the area surrounding the selected pixel in the main display. *(As explained earlier, the zoomed version makes it easier for the user to place a lasso around the offending redeye pixels.)*

## The zoom method

I am going to put the explanation of the constructor on hold for now and explain the **zoom** method. I will return to the explanation of the remainder of the constructor later.

### Behavior of the zoom method

The **zoom** method creates a new **Picture** object that contains a scaled rectangular sub image of the pixels surrounding the pixel selected by the user before clicking the **Zoom** button. The amount of scaling is determined by the radio button that is selected when the method is called. Scaling factors of 1, 2, 4, 8, and 16 are available.

### The size of the zoomed image

If the selected pixel is not close to the edge of the image, the size of the zoomed image will be 464x464 pixels.  If the selected pixel is close to one of the edges of the image, the zoomed image will be smaller.

There is nothing magic about the number 464 except that the size of the zoomed image should be evenly divisible by 16 if possible to minimize any corruption of the image that might result from first enlarging and later reducing the size of the image.

## A convenient size

I chose 464 for the size of the zoomed image because that is the largest value that is divisible by 16 and is smaller than the widest image that I can publish at Developer.com.  You can increase the dimensions if you like but be sure to make each dimension evenly divisible by 16.

## Beginning of the zoom method

The **zoom** method begins in Listing 7.  The method begins by enabling and disabling various buttons appropriate to the current state of the program.  This will be a recurring theme throughout the program.

**Listing 7. Beginning of the zoom method.**

```
  private void zoom(){
    //Enable and disable various buttons.
    zoomButton.setEnabled(false);
    commitButton.setEnabled(false);
    writeButton.setEnabled(false);
    quitButton.setEnabled(false);
    recombineButton.setEnabled(true);

    //Establish the amount of zoom that will
be applied.
    if(zoom1.isSelected()){
      zoomFactor = 1;
    }else if(zoom2.isSelected()){
      zoomFactor = 2;
    }else if(zoom4.isSelected()){
      zoomFactor = 4;
    }else if(zoom8.isSelected()){
      zoomFactor = 8;
    }else{
      zoomFactor = 16;
    }//end else
```

## Set the zoom factor

Then Listing 7 checks to see which radio button is selected *(see Figure 11)* and sets the zoom factor to either 1, 2, 4, 8, or 16 depending on which button is selected. The value of **zoomFactor** will be used to control the factor by which the image is enlarged later.

## Identify the rectangular sub image that will be scaled

Listing 8 uses the value of **zoomFactor** along with the location of the selected pixel to determine the location of the rectangular section of the image that will be copied, scaled, and presented as the zoomed image.

**Listing 8. Identify the rectangular sub image that will be scaled.**

```
    //The numerator in the following fraction
needs to be
    // divisible by 16 to maintain the best
picture
    // quality when the image is later
restored to its
    // original size.
    zoomPictureWidth = 464/zoomFactor;
    zoomPictureHeight = 464/zoomFactor;

    //Compute the coordinates for the upper-
left corner
    // along with the width and height of the
rectangular
    // sub image.
    xMin = anchorX - 232/zoomFactor -
leftInset;

    if(xMin < 0){//Avoid negative coordinate
values.
      xMin = 0;
    }//end if

    if((xMin + zoomPictureWidth) >
(pictureWidth - 1)){
      zoomPictureWidth = pictureWidth - xMin -
1;
    }//end if

    yMin = anchorY - 232/zoomFactor -
topInset;

    if(yMin < 0){
      yMin = 0;//Avoid negative coordinate
values.
    }//end if

    if((yMin + zoomPictureHeight) >
(pictureHeight - 1)){
      zoomPictureHeight = pictureHeight - yMin
- 1;
```

```
    }//end if
```

Four values are needed

Four values are needed to specify the rectangular section of pixels that will be copied and scaled.

- The x-coordinate of the upper left corner.
- The y-coordinate of the upper left corner.
- The width.
- The height.

The code in Listing 8 computes those four values, being careful to avoid specifying a rectangle that isn't fully contained within the image.

As I mentioned earlier, the code in Listing 8 converts the values from coordinates relative to the upper left corner of the frame to coordinates relative to the upper left corner of the image.

**Get, scale, and display the rectangular sub image**

Listing 9 calls the **getSubimage** method of the **BufferedImage** class on the main image to get an image that is a copy of the pixels contained in the rectangular sub image defined above.

**Listing 9. Get, scale, and display the rectangular sub image.**

```
    //Get the subimage that will be scaled up
to create
    // the zoomed image.
    BufferedImage subImage = ((BufferedImage)(

displayPicture.getImage())).getSubimage(
                                       xMin,
                                       yMin,

zoomPictureWidth,

zoomPictureHeight);

    //Use the subimage to create two new
Picture objects,
    // one for backup, and the other for a
working
    // picture. Both are scaled up by
zoomFactor. Display
    // the working picture.
    zoomBackupPicture = new Picture(subImage);
    zoomBackupPicture =
zoomBackupPicture.scale(

zoomFactor,zoomFactor);
    zoomDisplayPicture = new
Picture(zoomBackupPicture);
    zoomDisplayPicture.show();
```

## Create a new scaled Picture object

Listing 9 then uses the sub image to create a new **Picture** object and scales it by the value of **zoomFactor** in both dimensions.  This picture is saved as a zoomed backup picture and is also used to create another picture object, which is displayed to the right of the main display *(as shown in Figure 2)*.

## Perform miscellaneous tasks on the zoomed picture

There are several miscellaneous tasks that need to be performed with regard to the zoomed picture after it is created. Those  tasks are handled by the code in Listing 10.

**Listing 10. Perform miscellaneous tasks on the zoomed picture.**

```
    //Get and save a reference to the JFrame
object that
    // contains the working picture.
    zoomPictureFrame =

zoomDisplayPicture.getPictureFrame().frame;

    //Disable the X-button on the zoom
display.
    zoomPictureFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);

    //Position the zoomed working picture at
the top of
    // the screen immediately to the right of
either
    // displayPicture or the GUI, whichever is
wider.
    int zoomLocationX = 0;
    if((pictureWidth + leftInset + rightInset)
>

this.getWidth()){
      zoomLocationX =
                    pictureWidth + leftInset +
rightInset;
    }else{
      zoomLocationX = this.getWidth();
    }//end else
```

```
zoomPictureFrame.setLocation(zoomLocationX,0);

    //Get and save a reference to the object
on which the
    // circular lasso will be drawn.
    g2d =
(Graphics2D)(zoomPictureFrame.getGraphics());
    zoomImage =

(BufferedImage)(zoomBackupPicture.getImage());
```

None of the tasks are complicated so no further explanation of Listing 10 should be required.

## Register a MouseListener object on the zoomed picture

Now that the zoomed picture exists, the program needs to make it possible to draw a lasso on it by dragging the mouse in the image.  This requires that a **MouseListener** object and a **MouseMotionListener** object be registered on the frame that contains the picture.

Listing 11 defines an anonymous class that implements the **MouseListener** interface and registers an anonymous object of that class on the frame that contains the picture.

**Listing 11. Register a MouseListener object on the zoomed picture.**

```
    //Register a mousePressed listener and a
mouseDragged
    // listener on the zoomPictureFrame. This
is done here
    // because it couldn't be done before the
zoomed
    // display picture was created.
    //----------------------------------------------
----------//

    // This mousePressed event handler creates
a fresh
    // zoomed image using the zoomed backup
picture and
    // establishes the anchor point for the
ellipse that
    // will be drawn.
    zoomPictureFrame.addMouseListener(
      new MouseAdapter(){
        public void mousePressed(MouseEvent
e){
          graphics =
zoomDisplayPicture.getGraphics();
          graphics.drawImage(
zoomBackupPicture.getImage(),0,0,null);
          zoomDisplayPicture.repaint();

          zoomAnchorX = e.getX();
          zoomAnchorY = e.getY();
          zoomDeltaX = 0;
          zoomDeltaY = 0;

          //Enable the button that will be
used to call
          // the method that modifies the
colors of the
          // pixels enclosed by the lasso.
          fixRedeyeButton.setEnabled(true);

        }//end mousePressed
      }//end new MouseAdapter
    );//end addMouseListener
```

## Yesterday's news

By now, code like that shown in Listing 11 should be yesterday's news to you and any explanation beyond the embedded comments should not be required.

## Register a MouseMotionListener object on the zoomed picture

Listing 12 defines an anonymous class that implements the **MouseMotionListener** interface and registers an object of that class on the frame that contains the zoomed picture.

**Listing 12. Register a MouseMotionListener object on the zoomed picture.**

```
    //This mouseDragged event handler will
call a method
    // to draw a lasso when the mouse is
dragged in the
    // zoomed image.
    zoomPictureFrame.addMouseMotionListener(
      new MouseMotionAdapter(){
        public void mouseDragged(MouseEvent
e){
          drawLasso(e.getX(),e.getY());
        }//end mouseDragged
      }//end new MouseMotionAdapter
    );//end addMouseMotionListener
    //----------------------------------------
----------//

  }//end zoom method
```

## A series of events

A series of events of type **MouseEvent** will be fired as the user drags the mouse in the zoomed picture.  Each time such an event is fired, the **mouseDragged** method defined in Listing 12 will be called.  This method calls a method named **drawLasso**, passing the coordinates of the mouse pointer as parameters to the **drawLasso** method.

Listing 12 also signals the end of the **zoom** method.

## The drawLasso method

The **drawLasso** method, which is called from an event handler defined inside the **zoom** method in Listing 12, is not completely new to this lesson.  I explained a method similar to this one in an earlier lesson titled *Creating a lasso for editing digital photos in Java (see Resources)*.  However, due to the critical nature of this method to the operation of this program, I will explain the method again in this lesson.

### Draw a circular lasso

The **drawLasso** method draws a circular lasso that always touches the anchor point established by the **mousePressed** event handler defined in Listing 11.  The **drawLasso** method is called each time the **mouseDragged** event handler method is called as a result of the user dragging the mouse in the zoomed image.

### Beginning of the drawLasso method

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

**Listing 13. Beginning of the drawLasso method.**

```
  private void drawLasso(int x,int y){

    //The parameters x and y contain the 
coordinates of
    // the mouse pointer when the event was 
fired. Update
    // the diameter of the circular lasso.
    zoomDeltaX = x - zoomAnchorX;
    zoomDeltaY = y - zoomAnchorY;
    diameter = 
(int)Math.hypot(zoomDeltaX,zoomDeltaY);

    g2d.setColor(Color.GREEN);

    //Copy the entire image from the backup 
picture
    // stored in memory to erase any lassos 
drawn
    // earlier. This causes a single circular 
lasso to
    // appear to track the mouse position.
    g2d.drawImage(zoomImage,
zoomPictureFrame.getInsets().left,

zoomPictureFrame.getInsets().top,null);
```

## Set the diameter of the lasso

The method begins by using the current location of the mouse pointer along with the anchor point established in Listing 11 to compute the diameter of the circular lasso.

## Make the lasso pure green

Then Listing 13 sets the color of the lasso to pure green.  I chose this color because it is least likely to appear in the vicinity of a normal human eye.  A pure green lasso should stand out from the background reasonably well as shown in Figure 2.

## Erase the old lasso

Finally, Listing 13 copies the zoomed background image into the zoomed picture to erase the lasso that was drawn during the previous call to this method.  Without this, the lasso would look like a big disorganized pile of hula hoops of different sizes instead of a single circle.

## Create and draw the lasso

The code in Listing 14 uses trigonometry to compute the angle and to draw the circle as a circular ellipse of the class **Ellipse2D.Double**.

## An ellipse inscribed in a rectangle

In order to construct and then draw such an ellipse, you must specify the coordinates of the upper left corner along with the width and height of a rectangle in which the ellipse will be inscribed.

Because the ellipse is inscribed in the rectangle and the ellipse must touch the anchor point at all times, the upper left corner of the rectangle *(in this case a square)*, does not coincide with the anchor point except when the circle is to the right of, to the left of, above, or below the anchor point.

## The major challenge

Therefore, the major challenge is to determine the location of the upper left corner of the square for all angles other than 0, 90, 180, and 270 degrees.  This is accomplished by the code in Listing 14.

**Listing 14. Create and draw the lasso.**

```
    //Get the angle in radians that a line joining
the
    // anchor point and the current mouse location
makes
    // with a horizontal line going through the
anchor
    // point. This is the angle that will be used
in the
    // computations required to rotate the
circular
    // ellipse around the anchor point while
continually
    // touching the anchor point.
    angle =

Math.atan2((double)zoomDeltaY,(double)zoomDeltaX);

    //Create and draw a circular ellipse that
touches the
    // anchor point at all times.
    ellipse = new Ellipse2D.Double(
          //Compute and specify the coordinates of
the
          // upper left corner of a box that will
contain
          // the circular ellipse.
          zoomAnchorX-(
                  diameter/2-
Math.cos(angle)*diameter/2),
          zoomAnchorY-(
                  diameter/2-
Math.sin(angle)*diameter/2),
          //Specify the width and the height of
the box.
          diameter,
          diameter);

    //Draw the ellipse.
    g2d.draw(ellipse);

  }//end drawLasso
```

## If you understand trigonometry

If you know how to program using Java and you also have a pretty good understanding of trigonometry, you should be able to understand, or at least to figure out the code in Listing 14.  If you don't understand trigonometry, it is unlikely that you can understand Listing 14.  In that case, just take my word for it that the code does what it is intended to do.

After constructing the **Ellipse2D.Double** object, Listing 14 draws it on the zoomed image and the **drawLasso** method terminates.

## Fixing the redeye problem

After the user surrounds the offending pixels with the lasso, the user clicks the **Fix Redeye** button to cause the color of the offending pixels to be changed to gray.

## Register an anonymous listener object on the Fix Redeye button

Listing 15 defines an anonymous class that implements the **ActionListener** interface  and registers an object of that class on the button labeled **Fix Redeye**.

**Listing 15. Register an anonymous listener object on the Fix Redeye button.**

```
    //The event handler on this button calls a 
method that
    // changes the color of the offending 
redeye pixels
    // that are enclosed in a circular lasso.
    fixRedeyeButton.addActionListener(
      new ActionListener(){
        public void 
actionPerformed(ActionEvent e){
          fixRedeye();
        }//end action performed
      }//end newActionListener
    );//end addActionListener
```

## The actionPerformed method

The **actionPerformed** method that is defined in Listing 15 is called each time the user clicks the button.  This method in turn calls the method named **fixRedeye**.

## Not a perfect circle

It is unlikely that the offending redeye pixels in an image will form a perfect circle.  Therefore, when the user attempts to surround the offending pixels with a circular lasso, it is likely that the lasso will enclose some pixels that are not offending redeye pixels such as the bright highlight in Figure 2.

## A key feature

The color of those non-offending pixels should not be changed.  Therefore, one of the key features of this program is the application of a decision algorithm that attempts to separate offending redeye pixels from non-offending pixels in order to prevent the color of non-offending pixels from being changed.

## The isRedeye method

The decision algorithm is written into a method named **isRedeye**.  As you will see later, the method named **fixRedeye** calls the **isRedeye** method to qualify each pixel before changing the color of the pixel.  Therefore, I will explain the **isRedeye** method before I explain the **fixRedeye** method.

## A 3D wedge in the HSB color cone

**The 3D wedge**
Figure 4 shows the shape of the wedge at the top surface of the HSB color cone.  The wedge extends downward through the cone to a brightness value of 0.2.

The **isRedeye** method, which begins in Listing 16, is called to test a pixel to determine if it meets the redeye criterion.  The method tests to determine if the color of the pixel is inside a 3D wedge in the HSB color cone as shown by the gray area in Figure 4.

Only those pixels whose color is inside the wedge are deemed to be offending redeye pixels.

## Beginning of the isRedeye method

The method begins by defining three *selectivity constants* for hue, saturation, and brightness that establish the position and shape of the wedge shown in Figure 4.

**Listing 16. Beginning of the isRedeye method.**

```
  private boolean isRedeye(Pixel pixel){

    final float hueThreshold = (float)0.95;
    final float saturationThreshold =
(float)0.43;
    final float brightnessThreshold =
(float)0.2;
```

## The selectivity constants

The constants that are defined in Listing 16 control the selectivity of the decision algorithm.  Increasing the values towards an upper limit of 1.0 makes the wedge shown in Figure 4 smaller and makes the algorithm more selective.  If you make the algorithm more selective, some offending redeye pixels may not be identified.  *(This is the case with the offending redeye pixels in Figure 9.)*

**The HSB color model**
If this discussion doesn't seem to be making any sense to you, you probably need to go back and study the earlier lesson titled *The HSB Color Model* in Resources for an

explanation of the HSB color cone.

## Decreasing the selectivity

Decreasing the values of the selectivity constants causes the wedge shown in Figure 4 to become larger, and will make the decision algorithm less selective. This can cause the program to modify the color of pixels that aren't really offending redeye pixels. *(For example, some of the brown pixels in the iris selected and inappropriately modified in the bottom image of Figure 6.)*

## Values of constants and placement of lasso are critical

Therefore, the values of these constants, along with the placement of the lasso are critical. *(You may want to experiment with different values to see if you can come up with a better set of values.)*

## Get the pixel color

Listing 17 gets the red, green, and blue color values for the pixel as defined by the RGB color model.

**Listing 17. Get the pixel color.**

```
    Color color = pixel.getColor();
    double red = (double)color.getRed();
    double green = (double)color.getGreen();
    double blue = (double)color.getBlue();

    //Get the color as defined by the HSB 
color model.
    float[] hsbvals = new float[3];
    Color.RGBtoHSB(
                                        
(int)red,(int)green,(int)blue,hsbvals);
```

Then Listing 17 calls the **RGBtoHSB** method to translate the RGB color values into the hue, saturation, and brightness values as defined in the HSB color model. The hue, saturation, and brightness values are stored *(in that order)* in the three elements of the array object referred to by **hsbvals**.

## Test the color against the redeye criterion

Listing 18 tests to determine if the pixel color is inside the wedge shown in Figure 4. If so, the method returns true. Otherwise, the method returns false.

**Listing 18. Test the color against the redeye criterion.**

```
    if(((hsbvals[0] > hueThreshold) ||
        (hsbvals[0] < (1.0 - hueThreshold)))
&&
        (hsbvals[1] > saturationThreshold) &&
        (hsbvals[2] > brightnessThreshold)){
      return true;
    }//end if
    return false;
  }//end isRedeye
```

### Hue values are cyclical

Recall that the hue values are cyclical with the three-o'clock position in Figure 4 representing both 0.0 and 1.0.  Therefore, it is necessary to test the hue as being close to 1.0 and also as being close to 0.0 in Listing 18.

Listing 18 signals the end of the **isRedeye** method.

## The fixRedeye method

If the **Override** button shown in Figure 11 is not selected, the **fixRedeye** method calls the **isRedeye** method in an attempt to exclude any pixels contained in the lasso that aren't offending redeye pixels.  Then it changes the color of the remaining pixels in the lasso to gray while maintaining some of the highlights.  The method also smooths the edges of the lasso.

If the **Override** button is selected, the **fixRedeye** method skips the call to **isRedeye** and changes the color of all the pixels inside the lasso as shown in Figure 10,

### Beginning of the fixRedeye method

The **fixRedeye** method begins in Listing 19.

**Listing 19. Beginning of the fixRedeye method.**

```
  private void fixRedeye(){
    //Protect against clicking the button
before drawing
    // a lasso.
    if(ellipse == null) return;

    //Don't allow another event to be fired by
the button
    // while this method is being executed.
    fixRedeyeButton.setEnabled(false);

    //Working variables.
    Color color = null;
    int red = 0;
    int green = 0;
    int blue = 0;
```

The code in Listing 19 is straightforward and shouldn't require an explanation beyond the embedded comments.

## Change color of offending redeye pixels to gray

Listing 20 begins by using a pair of nested **for** loops along with an **if** statement to identify the pixels that are contained in the circular lasso *(ellipse)*. *(Note that it is necessary to compensate for the insets when making the test.)*

If a pixel is determined to be outside of the lasso, it is simply skipped and the next pixel is tested.

**Listing 20. Change color of offending redeye pixels to gray.**

```
    for(int col = 0;
              col < 
zoomDisplayPicture.getWidth();col++){
      for(int row = 0;
             row < 
zoomDisplayPicture.getHeight();row++){

        if(ellipse.contains( col + leftInset,
                             row + topInset)){
          //The pixel is inside the lasso. 
Consider
          // changing its color.
          //Get a reference to the pixel.
          Pixel pixel = 

zoomDisplayPicture.getPixel(col,row);

          //Test to see if the override button 
is
          // selected or the color of this 
pixel meets
          // the red eye criterion.
          if((overrideButton.isSelected()) || 

(isRedeye(pixel))){
            //This pixel is believed to be an 
offending
            // redeye pixel.
            color = pixel.getColor();
            red = color.getRed()/4;//Decrease 
red by 4.
            green = color.getGreen();
            blue = color.getBlue();

            //Create a gray from the average 
of the
```

```
            // modified red, the green, and
the blue
            // color values.
            int avg = (int)((red + green +
blue)/3.0);
            color = new
Color(avg,avg,avg);//gray
            pixel.setColor(color);
          }//end if
        }//end if

      }//end inner loop
    }//end outer loop
```

**When the pixel is inside the lasso...**

Once it is determined that the pixel is inside the lasso, a test is made to determine if the **Override** button is selected. If the **Override** button is not selected, the **isRedeye** method is called to determine if the color of the pixel meets the red eye criterion.

If the pixel meets the redeye criterion, or if the **Override** button is selected *(in which case the call to the **isRedeye** method is simply skipped)*, code is executed to change the color of the pixel.

**The new color is gray**

The new color is gray. The gray **Color** object is constructed by averaging weighted versions of the existing colors of red, green, and blue in order to reduce the red content and also maintain the highlights.

Listing 20 also signals the end of both the inner and the outer loops. When the code in Listing 20 has been executed, all of the pixels in the zoomed image have been examined. Those inside the lasso have been qualified to select only offending redeye pixels. The colors of the offending redeye pixels have been changed to various shades of gray.

**Smooth the edges and display the modified image**

Listing 21 calls a method named **smoothEdges**. This method averages pixels at the boundary of the lasso to avoid having a harsh color discontinuity at the boundary.

**Listing 21. Smooth the edges and display the modified image.**

```

    //Call a method to smooth the edges of the
area
    // contained in the lasso.
    smoothEdges();

    zoomDisplayPicture.repaint();
  }//end fixRedeye method
```

Listing 21 also signals the end of the method named **fixRedeye**.

## The smoothEdges method

I will stipulate up front that the inclusion of this method in the program may be overkill. I'm not certain that the cosmetic improvement provided by the method is worthwhile except possibly in the case of images where the eye is very large.

The purpose of this method is to break up the harsh outline of the circular lasso used to change the color of the pixels.

### Average a small group of pixels...

This method averages the colors of a small group of pixels at the left and right boundaries of the lasso. Then it sets the color of each pixel in the group to the average color. The number of pixels in the group is equal to the **zoomFactor**.

The probability is high that when the zoomed image is restored to its original size, only one of the pixels in each group will be preserved.

### The method named smoothEdges

The method is shown in its entirety in Listing 22.

**Listing 22. The method named smoothEdges.**

```
  private void smoothEdges(){

    if(ellipse == null) return;

    //Smooth the left side of the circle.
    for(int row = 0;
             row <
zoomDisplayPicture.getHeight();row++){
      for(int col = 0;
              col <
zoomDisplayPicture.getWidth();col++){
        if(ellipse.contains(col + leftInset,
                            row + topInset)){
          averagePixels(col,row);
          //Break out of the inner loop and
process the
          // nest row.
          col = zoomDisplayPicture.getWidth();
        }//end if
      }//end inner loop
    }//end outer loop

    //Smooth the right side of the circle.
This code is
    // essentially the mirror image of the
code used to
    // smooth the left side.
    for(int row = 0;
              row <
zoomDisplayPicture.getHeight();row++){
      for(int col =
zoomDisplayPicture.getWidth() - 1;
                                          col
>= 0;col--){
        if(ellipse.contains(col + leftInset,
                            row + topInset)){
          averagePixels(col,row);
          col = -1;//Break out of inner loop.
        }//end if
      }//end inner loop
    }//end outer loop

  }//end smoothEdges
```

## An exercise in indexing

The code in the method is simply an exercise in indexing. Although the method is tedious, it is not complex. Therefore, no explanation beyond the embedded comments should be needed.

The code in Listing 22 calls a method named **averagePixels** where the averaging of the pixel colors is actually performed.

## The averagePixels method

This method computes the average color of a group of contiguous pixels on the same row and sets the color of all of the pixels in the group to the average color. As mentioned above, the number of pixels in the group is equal to **zoomFactor**.

The **averagePixels** method is shown in its entirety in Listing 23.

**Listing 23. The averagePixels method.**

```
  private void averagePixels(int col,int row){
    //Avoid coordinate out of bounds problem.
    if((col - zoomFactor/2) < 0){return;}

    //Initialize accumulators.
    int redTotal = 0;
    int greenTotal = 0;
    int blueTotal = 0;

    //Compute the sums of the color values.
    for(int cnt = col - zoomFactor/2;
                         cnt < col +
zoomFactor/2;cnt++ ){
      pixel =
zoomDisplayPicture.getPixel(cnt,row);
      redTotal += pixel.getRed();
      greenTotal += pixel.getGreen();
      blueTotal += pixel.getBlue();
    }//end for loop

    //Now set the color of the pixels to the
average
    // color.
    for(int cnt = col - zoomFactor/2;
                         cnt < col +
zoomFactor/2;cnt++ ){
      pixel =
zoomDisplayPicture.getPixel(cnt,row);
      pixel.setRed(redTotal/zoomFactor);
      pixel.setGreen(greenTotal/zoomFactor);
      pixel.setBlue(blueTotal/zoomFactor);
    }//end for loop
  }//end averagePixels
```

You should have no difficulty understanding the code in Listing 23 on the basis of the embedded comments.

## Reduce and recombine the zoomed image

When the user clicks the **Recombine Images** button, the zoomed image is reduced back to its original size and merged back into the image in the main display. This is accomplished by an event handler that begins in Listing 24.

**Listing 24. Beginning of the event handler on the Recombine Images button.**

```

    recombineButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          //Return if there is no
zoomDisplayPicture to
          // recombine with displayPicture.
          if(zoomDisplayPicture == null)
return;

          //Enable and disable various
buttons.
          zoomButton.setEnabled(true);
          commitButton.setEnabled(true);
          writeButton.setEnabled(true);
          quitButton.setEnabled(true);
          recombineButton.setEnabled(false);

          //Scale the zoomed image back down 
to the
          // correct size.
          zoomDisplayPicture = 
zoomDisplayPicture.scale(

1.0/zoomFactor,1.0/zoomFactor);
```

You should have no difficulty understanding the code in Listing 24. When the code in Listing 24 has executed, the **Picture** object referred to by **zoomDisplayPicture** has been scaled back down to the original size.

## Merge the two images

Listing 25 uses a pair of nested **for** loops to merge the image from the **Picture** object referred to by **zoomDisplayPicture** back into the correct position in the **Picture** object referred to by **pictureObject**.

**Listing 25. Merge the two images.**

```
          for(int col = 0;col < pictureWidth - 
1;col++){
            for(int row = 0;row < 
pictureHeight -1;row++){
              if((col >= xMin) && 
                 (col < (xMin + 
zoomPictureWidth)) && 
                 (row >= yMin) && 
                 (row < (yMin + 
zoomPictureHeight))){
                //Get access to the pixel that 
will be
                // modified.
                pixel = 
displayPicture.getPixel(col,row);
                //Modify the color of the 
pixel.
                pixel.setColor(

zoomDisplayPicture.getPixel(
                       col - xMin,row - 
yMin).getColor());
              }//end if
            }//end inner loop
          }//end outer loop
```

## Deciding where to insert the modified sub-image

Note that the code in Listing 25 uses **xMin**, **xMax**, **zoomPictureWidth**, and **zoomPictureHeight**, which were established in Listing 8 to decide where to insert the pixels from the zoomed image.  Recall that these are the four values that were used to define a sub image that would be copied and scaled in Listing 9.

When the code in Listing 25 has been executed, the pixels from the zoomed image have replaced the corresponding pixels in the image from which the sub image was originally copied.

### Dispose of the zoomed picture

Listing 26 disposes of the frame that contains the zoomed image.  This causes the zoomed picture shown in Figure 2 to disappear from the screen and releases all resources currently held by that picture.

**Listing 26. Dispose of the zoomed picture.**

```
          zoomPictureFrame.dispose();
          displayPicture.repaint();
        }//end action performed
      }//end newActionListener
    );//end addActionListener
```

### Repaint and end the anonymous class

Listing 26 also repaints the main display causing the modified image to be displayed as shown in the bottom image in Figure 1.

Finally, Listing 26 signals the end of the definition of the anonymous class that is defined to register an **ActionListener** on the button labeled **Recombine Images.**

## Commit the modified image

Any time that the **Commit** button is enabled, the user can commit the image in the main display into the backup image by clicking the button labeled **Commit**.  This is accomplished by an **ActionListener** object that is instantiated from the anonymous class defined in Listing 27.

**Listing 27. Register an ActionListener object on the Commit button.**

```
    commitButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          backupPicture = new
Picture(displayPicture);
        }//end action performed
      }//end newActionListener
    );//end addActionListener
```

**Replace the backup picture**

When the **Commit** button is clicked, the **Picture** object referred by **backupPicture** is replaced by a new **Picture** object that is a copy of the **Picture** object referred to by **the displayPicture**. This means that all changes made up to that point in time are committed and cannot be undone.

**Caution regarding the white cursor**
If the **Commit** button is clicked while a white cursor is showing, that cursor will be committed to the backup picture and cannot be removed without starting all over.

### Writing output files

As I explained earlier, any time the **Write** button shown in Figure 11 is enabled, the user can write an output file containing the image in the main display shown in Figure 1.

In addition, if the user clicks the **Quit** button when it is enabled, or clicks the large X in the upper right corner of the GUI, the program will terminate, and in the process will write a final output file containing the image in the main display.

Three additional anonymous classes are defined to handle these actions. However, the code in the three classes is routine so I won't bore you with an explanation. You can view that code in Listing 28.

## Run the program

I encourage you to copy the code from Listing 28. Compile and execute that code using the digital photographs of your choice. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Don't forget that Ericsson's multimedia library is required to compile and execute this program.

## Summary

In this lesson I showed you how to write a Java program that can be used to correct for redeye problems in digital photographs.

## What's next?

In the next lesson, you will learn how to construct a complex panel and add it to a JFrame, including event registration using anonymous listener classes, BoxLayout, and resource loading via the Class class.  This is an important aspect of understanding Ericson's multimedia library.

## Resources

- Creative Commons Attribution 3.0 United States License
- Media Computation book in Java - numerous downloads available
- Introduction to Computing and Programming with Java: A Multimedia Approach
- DrJava download site
- DrJava, the JavaPLT group at Rice University
- DrJava Open Source License
- The Essence of OOP using Java, The this and super Keywords
- Threads of Control
- Painting in AWT and Swing
- Wikipedia Turtle Graphics
- IsA or HasA
- Vector Cad-Cam XI Lathe Tutorial
- Classification of 3D to 2D projections
- Color model from Wikipedia
- Light and color:  an introduction by Norman Koren
- Color Principles - Hue, Saturation, and Value
- 200 Implementing the Model-View-Controller Paradigm using Observer and Observable
- 300 Java 2D Graphics, Nested Top-Level Classes and Interfaces
- 302 Java 2D Graphics, The Point2D Class
- 304 Java 2D Graphics, The Graphics2D Class
- 306 Java 2D Graphics, Simple Affine Transforms
- 308 Java 2D Graphics, The Shape Interface, Part 1
- 310 Java 2D Graphics, The Shape Interface, Part 2
- 312 Java 2D Graphics, Solid Color Fill
- 314 Java 2D Graphics, Gradient Color Fill
- 316 Java 2D Graphics, Texture Fill
- 318 Java 2D Graphics, The Stroke Interface
- 320 Java 2D Graphics, The Composite Interface and Transparency
- 322 Java 2D Graphics, The Composite Interface, GradientPaint, and Transparency
- 324 Java 2D Graphics, The Color Constructors and Transparency
- 400 Processing Image Pixels using Java, Getting Started
  402 Processing Image Pixels using Java, Creating a Spotlight
  404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
  406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
  408 Processing Image Pixels, Performing Convolution on Images
  410 Processing Image Pixels, Understanding Image Convolution in Java
  412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
  414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
  416 Processing Image Pixels, An Improved Image-Processing Framework in

Java
[418](#) Processing Image Pixels, Creating Visible Watermarks in Java
[450](#) A Framework for Experimenting with Java 2D Image-Processing Filters
[452](#) Using the Java 2D LookupOp Filter Class to Process Images
[454](#) Using the Java 2D AffineTransformOp Filter Class to Process Images
[456](#) Using the Java 2D LookupOp Filter Class to Scramble and Unscramble Images
[458](#) Using the Java 2D BandCombineOp Filter Class to Process Images
[460](#) Using the Java 2D ConvolveOp Filter Class to Process Images
[462](#) Using the Java 2D ColorConvertOp and RescaleOp Filter Classes to Process Images

- 506 JavaBeans, Introspection
- 2100 Understanding Properties in Java and C#
- 2300 Generics in J2SE, Getting Started
- 340 Multimedia Programming with Java, Getting Started
- 342 Getting Started with the Turtle Class: Multimedia Programming with Java
- 344 Continuing with the SimpleTurtle Class: Multimedia Programming with Java
- 346 Wrapping Up the SimpleTurtle Class: Multimedia Programming with Java
- 348 The Pen and PathSegment Classes: Multimedia Programming with Java
- 349 A Pixel Editor Program in Java: Multimedia Programming with Java
- 350 3D Displays, Color Distance, and Edge Detection
- 351 A Slider-Controlled Softening Program for Digital Photos
- 352 Adding Animated Movement to Your Java Application
- 353 A Slider-Controlled Sharpening Program for Digital Photos
- 354 The DigitalPicture Interface
- 355 The HSB Color Model
- 356 The show Method and the PictureFrame Class
- 357 An HSB Color-Editing Program for Digital Photos
- 358 Applying Affine Transforms to Picture Objects
- 359 Creating a lasso for editing digital photos in Java
- 360 Wrapping Up the SimplePicture Class
- 361 A Temperature and Tint Editing Program for Digital Photos
- 362 Getting Started with the PictureExplorer Class

## Complete program listing

A complete listing of the program that I explained in this lesson is shown in Listing 28 below.

**Listing 28. Source code listing for the program named RedEye05.**

```
/*File RedEye05 Copyright 2009 R.G.Baldwin

The purpose of this program is to demonstrate a
programming technique for correcting redeye problems in
digital photographs.
```

The program begins by displaying a GUI in the upper left corner of the screen. At that point, the GUI contains a text field for entry of the name of the image file to be processed and some other user-input components, which are disabled. If the file is in the current directory, only the file name and extension must be entered. Otherwise, the full path and name and extension for the file must be entered. Files of types jpg, bmp, and png are supported.

When the user enters the name of the image file into the text field, the file is loaded into a Picture object. The Picture object is displayed in the upper left corner of the screen and the GUI is moved to a location immediately below the Picture object. At this point, the text field is disabled and several buttons and radio buttons are enabled.

The overall process is as follows. The user selects a pixel in the eye containing the redeye problem by clicking the mouse in the red area.

Then the user selects one of five radio buttons specifying a zoom factor of 1, 2, 4, 8, or 16 and clicks a button labeled Zoom. (A zoom factor of 16 is selected by default at program startup.) This causes the area surrounding the selected pixel to be enlarged by the selected zoom factor and displayed in another window to the right of the original picture object and the GUI. It also causes the Zoom button and several other buttons in the GUI to be disabled.

To correct the redeye problem in the selected eye, the user surrounds the offending pixels in the zoomed image with a circular lasso by dragging the mouse in the zoomed image. Then the user clicks a button labeled Fix Redeye.

Clicking the Fix Redeye button causes an algorithm to be executed that attempts to correct the redeye problem in the area enclosed by the circular lasso. The algorithm scans all of the pixels isolated by the circular lasso, attempts to exclude any pixels that are not part of the problem, and changes the color of the remaining pixels to a dark gray with highlights preserved.

Clicking the mouse in the zoomed image at any time will erase an existing lasso and undo the effects of having clicked the Fix Redeye button.

The process can be repeated as many times as necessary until the user is satisfied that the best fit of the circular lasso and the offending pixels has been achieved. Then the user clicks a button labeled Recombine Images to cause the modified zoomed image to be reduced back to its original size and merged back into the original image.

```
If the user is not happy with the results, the entire
process can be repeated by selecting the same eye again
and clicking the Zoom button. This will undo everything
back to the most recent commit operation (see below). If
the user is happy with the results for the eye currently
being processed, the user can accept the results by
clicking the Commit button and then perform the same
procedure on the other eye. Once committed, the pixels are
permanently changed in a backup image being held in memory
and the changes cannot be undone. However, at no time does
this program modify the original image file.

As explained above, the user can drag the mouse in the
zoomed image to create a circular lasso. An anchor point
is established when the user first presses the mouse
button to begin the drag operation.

The lasso can be created in any direction from the anchor
point.  The diameter of the lasso is equal to the distance
from the mouse pointer to the anchor point. The final size
and position of the lasso is established when the user
releases the mouse button and ends the drag operation.

Dragging the mouse outside the bounds of the image
causes the size of the lasso to continue to grow. However,
if the lasso extends outside the bounds of the zoomed
image, clicking the Fix Redeye button will have no effect.

The lasso remains on the screen until the user clicks the
zoomed image again with the mouse, clicks the Fix Redeye
button, clicks the Recombine Images button, or does
something else to cause the image to be repainted such as
minimizing and then restoring the zoomed image.

Clicking the Write button (when it is enabled) causes a
backup bmp file of the displayed image to be written into
the same directory from which the image file was read. The
five most recent backup files are saved. The names of the
backup files are the same as the name of the original
image file except that the characters BAKn are inserted
immediately before the extension. The character n is
replaced by a digit from 0 through 4.

The large X buttons in the upper right corner of both
image displays are disabled.  Clicking them does nothing.

The program is terminated by clicking the Quit button
(when it is enabled) or clicking the large X in the
upper right corner of the GUI at any time.

Before terminating, the program writes an output file
containing the final state of the displayed image in the
same format as the input file. The file is written into
the folder from which the original image file was read.
The name of the output file is the same as the name of the
input file except that the word FINAL is inserted
```

```
immediately before the extension.

The class files in Ericson's multimedia library must be
accessible to the program.

Tested using Windows Vista Home Premium Edition,
Java 1.6x, and the version of Ericson's multimedia library
contained in bookClasses10-1-07.zip.
**********************************************************/

import java.awt.Graphics;
import java.awt.Image;
import java.awt.BorderLayout;
import java.awt.Graphics2D;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseMotionAdapter;

import java.awt.geom.Ellipse2D;
import java.awt.geom.Ellipse2D.Double;

import java.awt.image.BufferedImage;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JTextField;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;
import javax.swing.WindowConstants;

import java.io.File;

public class RedEye05 extends JFrame{
  //Create the components that are used to construct the
  // GUI.
  private JPanel mainPanel = new JPanel();
  private JPanel northPanel = new JPanel();
  private JPanel centerPanel = new JPanel();
  private JPanel southPanel = new JPanel();

  private JButton fixRedeyeButton =
                                new JButton("Fix Redeye");
  private JButton writeButton = new JButton("Write File");
  private JButton zoomButton = new JButton("Zoom");
  private JButton recombineButton =
                          new JButton("Recombine Images");
  private JButton quitButton = new JButton("Quit");
  private JButton commitButton = new JButton("Commit");
  private JRadioButton zoom1 = new JRadioButton("X1");
  private JRadioButton zoom2 = new JRadioButton("X2");
  private JRadioButton zoom4 = new JRadioButton("X4");
  private JRadioButton zoom8 = new JRadioButton("X8");
  private JRadioButton zoom16 =
                             new JRadioButton("X16",true);
  private JRadioButton overrideButton =
                             new JRadioButton("Override");
  private ButtonGroup buttonGroup = new ButtonGroup();

  private JTextField fileNameField =
                           new JTextField("RedEye05.jpg");
  private JLabel fileNameLabel = new JLabel("File Name:");

  //A reference to the original Picture object will be
  // stored here.
  private Picture backupPicture = null;
  //A reference to the picture that is zoomed will be
  // stored here.
  private Picture zoomBackupPicture = null;

  //A reference to a working copy of the original
  // Picture object will be stored here.
  private Picture displayPicture = null;
  //A reference to a working copy of the zoomed picture
  // will be stored here.
  private Picture zoomDisplayPicture = null;

  //Miscellaneous working variables.
  private Graphics graphics = null;

  private Pixel pixel = null;
  private int writeCounter = 0;

  private String fileName = null;
  private String outputPath = null;
  private String extension = null;

  private int pictureWidth = 0;
  private int pictureHeight = 0;
  private int zoomPictureWidth = 0;
  private int zoomPictureHeight = 0;
  private int xMin = 0;
  private int yMin = 0;

  private int anchorX = 0;
  private int anchorY = 0;
  private int zoomAnchorX = 0;
  private int zoomAnchorY = 0;

  private int zoomDeltaX = 0;
  private int zoomDeltaY = 0;

  private int diameter = 0;
  private double angle = 0;
  private int leftInset = 0;
  private int topInset = 0;
  private int rightInset = 0;
  private int bottomInset = 0;

  private int zoomFactor = 16;

  private BufferedImage theImage = null;
  private BufferedImage zoomImage = null;

  private JFrame displayPictureFrame = null;
  private JFrame zoomPictureFrame = null;

  private Ellipse2D.Double ellipse = null;

  private Graphics2D g2d = null;

  private final double pi = Math.PI;//convenience constant
  //---------------------------------------------------//

  public static void main(String[] args){//main method
    new RedEye05();
  }//end main method
  //---------------------------------------------------//

  public RedEye05(){//constructor

    //All close operations are handled in a WindowListener
    // object.
    setDefaultCloseOperation(
                     WindowConstants.DO_NOTHING_ON_CLOSE);

    //Construct the GUI. Components are arranged in the
    // GUI from left to right, top to bottom in
    // approximately the order that they are used.
    mainPanel.setLayout(new BorderLayout());
    mainPanel.add(northPanel,BorderLayout.NORTH);
    mainPanel.add(centerPanel,BorderLayout.CENTER);
    mainPanel.add(southPanel,BorderLayout.SOUTH);

    //Add components to the north panel.
    northPanel.add(fileNameLabel);
    northPanel.add(fileNameField);
    northPanel.add(zoom1);
    northPanel.add(zoom2);
    northPanel.add(zoom4);
    northPanel.add(zoom8);
    northPanel.add(zoom16);
    buttonGroup.add(zoom1);
    buttonGroup.add(zoom2);
    buttonGroup.add(zoom4);
    buttonGroup.add(zoom8);
    buttonGroup.add(zoom16);
    northPanel.add(zoomButton);
```

```
    //Add components to the center panel
    centerPanel.add(overrideButton);
    centerPanel.add(fixRedeyeButton);
    centerPanel.add(recombineButton);

    //Add components to the south panel.
    southPanel.add(commitButton);
    southPanel.add(writeButton);
    southPanel.add(quitButton);

    //Disable the buttons until the user enters the file
    // name.
    zoom1.setEnabled(false);
    zoom2.setEnabled(false);
    zoom4.setEnabled(false);
    zoom8.setEnabled(false);
    zoom16.setEnabled(false);
    zoomButton.setEnabled(false);
    fixRedeyeButton.setEnabled(false);
    recombineButton.setEnabled(false);
    commitButton.setEnabled(false);
    writeButton.setEnabled(false);
    quitButton.setEnabled(false);

    //Set the size of the GUI and display it in the upper
    // left corner of the screen. It will be moved later
    // to a position immediately below the display of the
    // picture.
    getContentPane().add(mainPanel);
    pack();
    setVisible(true);

    //Request that the focus move to the text field where
    // the file name is to be entered.
    fileNameField.requestFocus();
    //--------------------------------------------------//

    //Register a listener on the text field. When the user
    // enters the file name in the text field, set
    // everything up properly so that the program will
    // function as an event-driven picture-manipulation
    // program until the user clicks the large X in the
    // upper-right of the GUI.
    fileNameField.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          //Disable the text field and its label to
          // prevent the user from entering anything else
          // into it and causing it to fire another event.
          fileNameField.setEnabled(false);
          fileNameLabel.setEnabled(false);

          //Get the file name from the text field and use
          // it to create a new Picture object.
          fileName = fileNameField.getText();
          backupPicture = new Picture(fileName);
          //Get information that will be used to write the
          // output files.
          String inputPath = new File(fileName).
                                        getAbsolutePath();
          int posDot = inputPath.lastIndexOf('.');
          outputPath = inputPath.substring(0,posDot);

          //Write the first copy of the output backup
          // file before any processing is done.
          backupPicture.write(outputPath
                       + "BAK" + writeCounter++ + ".bmp");

          //Get filename extension. It will be used later
          // to write the final output file.
          extension = inputPath.substring(posDot);

          //Decorate the GUI.
          setTitle("Copyright 2009, R.G.Baldwin");

          //Create the picture that will be used for
          // processing.
          //Note that the original image file is not
          // modified by this program.
          displayPicture = new Picture(backupPicture);

          //Display the picture.
          displayPicture.show();

          //Save a reference to the image. Also save the
          // width and height of the picture.
          theImage =
                (BufferedImage)(backupPicture.getImage());
          pictureWidth = backupPicture.getWidth();
          pictureHeight = backupPicture.getHeight();

          //Get and save a reference to the JFrame object
          // that contains the image.
          displayPictureFrame =
                   displayPicture.getPictureFrame().frame;

          //Get and save the insets for the JFrame object.
          leftInset =
                     displayPictureFrame.getInsets().left;
          topInset = displayPictureFrame.getInsets().top;
          rightInset =
                    displayPictureFrame.getInsets().right;
          bottomInset =
                   displayPictureFrame.getInsets().bottom;

          //Adjust the width of the GUI to match the width
          // of the display if possible. Then relocate the
          // GUI to a position immediately below the
          // display.
          //Establish the preferred size now that the
          // input file name has been entered.
```

```
          pack();
          int packedHeight = getHeight();
          int packedWidth = getWidth();
          if((pictureWidth + 7) >= packedWidth){
            //Make the width of the GUI the same as the
            // width of the display.
            setSize(pictureWidth + 7,packedHeight);
          }//Else, just leave the GUI at its current size.
          //Put the GUI in its new location immediately
          // below the display.
          setLocation(0,pictureHeight + 30);

          //Enable user input controls.
          zoom1.setEnabled(true);
          zoom2.setEnabled(true);
          zoom4.setEnabled(true);
          zoom8.setEnabled(true);
          zoom16.setEnabled(true);
          zoomButton.setEnabled(true);
          commitButton.setEnabled(true);
          writeButton.setEnabled(true);
          quitButton.setEnabled(true);

          //Disable the X-button on the display.
          displayPictureFrame.setDefaultCloseOperation(
                     WindowConstants.DO_NOTHING_ON_CLOSE);
          //--------------------------------------------//

          /*
          Note that the following listener registration is
          actually inside the action listener that is
          registered on the text field. The code in the
          registration block can't be executed when the
          GUI is first constructed because a Picture
          object does not exist at that point in time.
          This code is executed after the user enters the
          image file name, the file has been read, and the
          Picture object referred to by displayPicture has
          been constructed.
          */
          displayPictureFrame.addMouseListener(
            new MouseAdapter(){
              public void mousePressed(MouseEvent e){
                //Draw a new copy of the image on the
                // display each time the user clicks the
                // image with the mouse. This creates a
                // fresh copy of the displayPicture. Note
                // however that the image being copied
                // may contain changes made earlier when
                // the user clicked the Commit button.
                graphics = displayPicture.getGraphics();
                graphics.drawImage(
                       backupPicture.getImage(),0,0,null);
                displayPicture.repaint();

                //Prepare some working variables.
                //Note that the reported coordinates for
                // a mouse press on the upper-left corner
                // of the image will not be reported
                // as 0,0 due to the top and left insets
                // of the JFrame.
                anchorX = e.getX();
                anchorY = e.getY();
                //Draw a small white cursor to mark the
                // location of the mouse press on the
                // image.
                drawCursor();
              }//end mousePressed
            }//end new MouseAdapter
          );//end addMouseListener
          //------------------------------------------//
        //Now finish defining the action listener that is
        // registered on the text field.
        }//end actionPerformed
      }//end new ActionListener
    );//end addActionListener
    //----------------------------------------------------//

    //The event handler on this button copies the backup
    // picture into displayPicture and then calls the zoom
    // method to produce a zoomed version of
    // displayPicture. The zoomed version makes it easier
    // to place a lass on around the offending redeye
    // pixels.
    zoomButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          //Get a fresh image in order to prevent the
          // current cursor from showing in the zoomed
          // version.
          //Note however that if the user previously
          // clicked the Commit button when a cursor was
          // showing in the picture, that cursor will have
          // been permanently written into the backup
          // picture.
          graphics = displayPicture.getGraphics();
          graphics.drawImage(
                       backupPicture.getImage(),0,0,null);
          displayPicture.repaint();

          //Call the zoom method to do all the hard work.
          zoom();
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //----------------------------------------------------//

    //The event handler on this button calls a method that
    // changes the color of the offending redeye pixels
    // that are enclosed in a circular lasso.
    fixRedeyeButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          fixRedeye();
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //---------------------------------------------------//

    //The event handler on this button restores the zoomed
    // image to the correct size and recombines it with
    // displayPicture.
    //There is probably a way to combine the two images
    // directly without having to loop, but I haven't
    // found it yet.
    recombineButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          //Return if there is no zoomDisplayPicture to
          // recombine with displayPicture.
          if(zoomDisplayPicture == null) return;

          //Enable and disable various buttons.
          zoomButton.setEnabled(true);
          commitButton.setEnabled(true);
          writeButton.setEnabled(true);
          quitButton.setEnabled(true);
          recombineButton.setEnabled(false);

          //Scale the zoomed image back down to the
          // correct size.
          zoomDisplayPicture = zoomDisplayPicture.scale(
                           1.0/zoomFactor,1.0/zoomFactor);

          //Merge the modified image into displayPicture
          for(int col = 0;col < pictureWidth - 1;col++){
            for(int row = 0;row < pictureHeight -1;row++){
              if((col >= xMin) &&
                 (col < (xMin + zoomPictureWidth)) &&
                 (row >= yMin) &&
                 (row < (yMin + zoomPictureHeight))){
                //Get access to the pixel that will be
                // modified.
                pixel = displayPicture.getPixel(col,row);
                //Modify the color of the pixel.
                pixel.setColor(
                     zoomDisplayPicture.getPixel(
                       col - xMin,row - yMin).getColor());
              }//end if
            }//end inner loop
          }//end outer loop

          //Release the resources that are held by the
          // zoomed picture and repaint displayPicture.
          zoomPictureFrame.dispose();
          displayPicture.repaint();
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //--------------------------------------------------//

    //When the Commit button is clicked, the
    // displayPicture is committed for the long term by
    // copying it to the backup picture. Note that if this
    // button is clicked while a white cursor is showing
    // in displayPicture, that cursor will be committed
    // to the backup picture and cannot be removed
    // without starting all over.
    commitButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          backupPicture = new Picture(displayPicture);
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //--------------------------------------------------//

    //Register an ActionListener on the writeButton.
    // Each time the user clicks the button, a backup bmp
    // file containing the current state of displayPicture
    // is written into the directory from which the
    // original image was read. The five most recent
    // backup files are saved.
    //The names of the backup files are the same as the
    // name of the input file except that BAKn is
    // inserted immediately ahead of the extension
    // where n is a digit ranging from 0 to 4. The value
    // of n rolls over at 4 and starts back at 0.
    writeButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          displayPicture.write(outputPath
                       + "BAK" + writeCounter++ + ".bmp");
          //Reset the writeCounter if it exceeds 4 to
          // conserve disk space.
          if(writeCounter > 4){
            writeCounter = 0;
          }//end if
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //--------------------------------------------------//

    //The behavior of the event handler on the quit button
    // is the same as the behavior of the event handler on
    // the large X in the upper right corner of the GUI
    // that is explained below.
    quitButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          displayPicture.write(
                        outputPath + "FINAL" + extension);
          System.exit(0);
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //--------------------------------------------------//

    //Register a WindowListener that will respond when the
    // user clicks the large X in the upper-right corner
    // of the GUI. This event handler will write the final
    // state of displayPicture into an output file of the
    // same type as the original input file. The name will
    // be the same except that the word FINAL will be
    // inserted immediately ahead of the extension.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          displayPicture.write(
                        outputPath + "FINAL" + extension);
          System.exit(0);
        }//end windowClosing
      }//end new WindowAdapter
    );//end addWindowListener
    //--------------------------------------------------//


  }//end constructor
  //----------------------------------------------------//

  //This method creates a new Picture object that contains
  // a rectangular subimage of the pixels surrounding the
  // pixel selected by the user before clicking the Zoom
  // button.
  //The amount of zoom is determined by the radio button
  // that is selected. If the selected pixel is not close
  // to the edge of the image, the size of the zoomed
  // image will be 464x464 pixels. If the selected pixel
  // is close to one of the edges of the image, the zoomed
  // image will be smaller.
  private void zoom(){
    //Enable and disable various buttons.
    zoomButton.setEnabled(false);
    commitButton.setEnabled(false);
    writeButton.setEnabled(false);
    quitButton.setEnabled(false);
    recombineButton.setEnabled(true);

    //Establish the amount of zoom that will be applied.
    if(zoom1.isSelected()){
      zoomFactor = 1;
    }else if(zoom2.isSelected()){
      zoomFactor = 2;
    }else if(zoom4.isSelected()){
      zoomFactor = 4;
    }else if(zoom8.isSelected()){
      zoomFactor = 8;
    }else{
      zoomFactor = 16;
    }//end else
    //The numerator in the following fraction needs to be
    // divisible by 16 to maintain the best picture
    // quality when the image is later restored to its
    // original size.
    zoomPictureWidth = 464/zoomFactor;
    zoomPictureHeight = 464/zoomFactor;

    //Compute the coordinates for the upper-left corner
    // along with the width and height of the rectangular
    // sub image.
    xMin = anchorX - 232/zoomFactor - leftInset;

    if(xMin < 0){//Avoid negative coordinate values.
      xMin = 0;
    }//end if

    if((xMin + zoomPictureWidth) > (pictureWidth - 1)){
      zoomPictureWidth = pictureWidth - xMin - 1;
    }//end if

    yMin = anchorY - 232/zoomFactor - topInset;

    if(yMin < 0){
      yMin = 0;//Avoid negative coordinate values.
    }//end if

    if((yMin + zoomPictureHeight) > (pictureHeight - 1)){
      zoomPictureHeight = pictureHeight - yMin - 1;
    }//end if

    //Get the subimage that will be scaled up to create
    // the zoomed image.
    BufferedImage subImage = ((BufferedImage)(
                  displayPicture.getImage())).getSubimage(
                                     xMin,
                                     yMin,
                                     zoomPictureWidth,
                                     zoomPictureHeight);
    //Use the subimage to create two new Picture objects,
    // one for backup, and the other for a working
    // picture. Both are scaled up by zoomFactor. Display
    // the working picture.
    zoomBackupPicture = new Picture(subImage);
    zoomBackupPicture = zoomBackupPicture.scale(
                                 zoomFactor,zoomFactor);
    zoomDisplayPicture = new Picture(zoomBackupPicture);
    zoomDisplayPicture.show();

    //Get and save a reference to the JFrame object that
    // contains the working picture.
    zoomPictureFrame =
               zoomDisplayPicture.getPictureFrame().frame;

    //Disable the X-button on the zoom display.
    zoomPictureFrame.setDefaultCloseOperation(
                 WindowConstants.DO_NOTHING_ON_CLOSE);

    //Position the zoomed working picture at the top of
    // the screen immediately to the right of either
    // displayPicture or the GUI, whichever is wider.
    int zoomLocationX = 0;
    if((pictureWidth + leftInset + rightInset) >
                                     this.getWidth()){
      zoomLocationX =
                pictureWidth + leftInset + rightInset;
    }else{
      zoomLocationX = this.getWidth();
    }//end else

    zoomPictureFrame.setLocation(zoomLocationX,0);

    //Get and save a reference to the object on which the
    // circular lasso will be drawn.
    g2d = (Graphics2D)(zoomPictureFrame.getGraphics());
    zoomImage =
            (BufferedImage)(zoomBackupPicture.getImage());

    //---------------------------------------------------//
    //Register a mousePressed listener and a mouseDragged
    // listener on the zoomPictureFrame. This is done here
    // because it couldn't be done before the zoomed
    // display picture was created.
    //---------------------------------------------------//

    // This mousePressed event handler creates a fresh
    // zoomed image using the zoomed backup picture and
    // establishes the anchor point for the ellipse that
    // will be drawn.
    zoomPictureFrame.addMouseListener(
      new MouseAdapter(){
        public void mousePressed(MouseEvent e){
          graphics = zoomDisplayPicture.getGraphics();
          graphics.drawImage(
                  zoomBackupPicture.getImage(),0,0,null);
          zoomDisplayPicture.repaint();

          zoomAnchorX = e.getX();
          zoomAnchorY = e.getY();
          zoomDeltaX = 0;
          zoomDeltaY = 0;

          //Enable the button that will be used to call
          // the method that modifies the colors of the
          // pixels enclosed by the lasso.
          fixRedeyeButton.setEnabled(true);

        }//end mousePressed
      }//end new MouseAdapter
    );//end addMouseListener
    //---------------------------------------------------//
    //This mouseDragged event handler will call a method
    // to draw a lasso when the mouse is dragged in the
    // zoomed image.
    zoomPictureFrame.addMouseMotionListener(
      new MouseMotionAdapter(){
        public void mouseDragged(MouseEvent e){
          drawLasso(e.getX(),e.getY());
        }//end mouseDragged
      }//end new MouseMotionAdapter
    );//end addMouseMotionListener
    //---------------------------------------------------//

  }//end zoom method
  //-----------------------------------------------------//

  //This method draws a circular lasso that always
  // touches the anchor point established by the
  // mousePressed event handler.
  //This method is called each time the mouseDragged
  // event handler fires a MouseEvent.
  private void drawLasso(int x,int y){

    //The parameters x and y contain the coordinates of
    // the mouse pointer when the event was fired. Update
    // the diameter of the circular lasso.
    zoomDeltaX = x - zoomAnchorX;
    zoomDeltaY = y - zoomAnchorY;
    diameter = (int)Math.hypot(zoomDeltaX,zoomDeltaY);

    g2d.setColor(Color.GREEN);

    //Copy the entire image from the backup picture
    // stored in memory to erase any lassos drawn
    // earlier. This causes a single circular lasso to
    // appear to track the mouse position.
    g2d.drawImage(zoomImage,
                  zoomPictureFrame.getInsets().left,
                  zoomPictureFrame.getInsets().top,null);

    //Get the angle in radians that a line joining the
    // anchor point and the current mouse location makes
    // with a horizontal line going through the anchor
    // point. This is the angle that will be used in the
    // computations required to rotate the circular
    // ellipse around the anchor point while continually
    // touching the anchor point.
    angle =
       Math.atan2((double)zoomDeltaY,(double)zoomDeltaX);

    //Create and draw a circular ellipse that touches the
    // anchor point at all times.
    ellipse = new Ellipse2D.Double(
          //Compute and specify the coordinates of the
          // upper left corner of a box that will contain
          // the circular ellipse.
          zoomAnchorX-(
                  diameter/2-Math.cos(angle)*diameter/2),
          zoomAnchorY-(
                  diameter/2-Math.sin(angle)*diameter/2),
          //Specify the width and the height of the box.
          diameter,
          diameter);

    //Draw the ellipse.
    g2d.draw(ellipse);

  }//end drawLasso
  //--------------------------------------------------------//

  //This method executes an algorithm that attempts
  // to exclude any pixels contained in the lasso that
  // aren't offending redeye pixels, and change the
  // color of the remaining pixels contained in the lasso
  // to gray while maintaining some of the highlights.
  // The method also smooths the edges of the lasso.
  private void fixRedeye(){
    //Protect against clicking the button before drawing
    // a lasso.
    if(ellipse == null) return;

    //Don't allow another event to be fired while this
    // method is being executed.
    fixRedeyeButton.setEnabled(false);

    //Working variables.
    Color color = null;
    int red = 0;
    int green = 0;
    int blue = 0;

    for(int col = 0;
              col < zoomDisplayPicture.getWidth();col++){
      for(int row = 0;
             row < zoomDisplayPicture.getHeight();row++){
        //Change the color of the offending redeye pixels
        // inside the ellipse.
        //Note: It is necessary to compensate for the top
        // and left insets of the JFrame.
        if(ellipse.contains( col + leftInset,
                             row + topInset)){
          //The pixel is inside the lasso. Consider
          // changing its color.
          //Get a reference to the pixel.
          Pixel pixel =
                    zoomDisplayPicture.getPixel(col,row);
          //Test to see if the override button is
          // selected or the color of this pixel meets
          // the red eye criterion. If so, modify the
          // color. The new color is gray constructed
          // by averaging weighted versions of the
          // existing colors in order to reduce the red
          // content and also maintain the highlights.
          if((overrideButton.isSelected()) ||
                                       (isRedeye(pixel))){
            //This pixel is believed to be an offending
            // redeye pixel.
            color = pixel.getColor();
            red = color.getRed()/4;//Decrease red by 4.
            green = color.getGreen();
            blue = color.getBlue();
            //Create a gray from the average of the
            // modified red, the green, and the blue
            // color values.
            int avg = (int)((red + green + blue)/3.0);
            color = new Color(avg,avg,avg);//gray
            pixel.setColor(color);
          }//end if
        }//end if

      }//end inner loop
    }//end outer loop

    //Call a method to smooth the edges of the area
    // contained in the lasso.
    smoothEdges();

    zoomDisplayPicture.repaint();
  }//end fixRedeye method
  //----------------------------------------------------//

  //This method is called to test a pixel to determine if
  // it meets the redeye criterion. This is done by
  // identifying a 3D quasi-triangular wedge in the HSB
  // color model cone. Any pixel whose color is inside
  // the wedge is deemed to be an offending redeye pixel.
  private boolean isRedeye(Pixel pixel){
    //The following constants control the selectivity of
    // the algorithm relative to the decision as to
    // whether or not the color of an individual pixel
    // should be modified. Increasing the values towards
    // an upper limit of 1.0 makes the algorithm more
    // selective, in which case some problem pixels may
    // not have their color modified. Decreasing the
    // values can cause the program to modify the color
    // of pixels that aren't a part of the redeye
    // problem. Therefore, the values of these constants
    // are critical. Note, however, that they may not
    // work well for all digital photos, and an upgrade
    // with sliders that allow the user to specify the
    // values might be useful.
    final float hueThreshold = (float)0.95;
    final float saturationThreshold = (float)0.43;
    final float brightnessThreshold = (float)0.2;

    //Get pixel color as defined by the RGB color model.
    Color color = pixel.getColor();
    double red = (double)color.getRed();
    double green = (double)color.getGreen();
```

```
    double blue = (double)color.getBlue();

    //Get the color as defined by the HSB color model.
    float[] hsbvals = new float[3];
    Color.RGBtoHSB(
                  (int)red,(int)green,(int)blue,hsbvals);

    //For debug and test.
    /*
    System.out.println(hsbvals[0] + " " +
                       hsbvals[1] + " " +
                       hsbvals[2]);
    */
    //Test to see if the pixel color is in the wedge.
    //Recall that the hue values are cyclical so it is
    // necessary to test the hue as being close to 1.0
    // and  also as being close to 0.0;
    if(((hsbvals[0] > hueThreshold) ||
        (hsbvals[0] < (1.0 - hueThreshold))) &&
        (hsbvals[1] > saturationThreshold) &&
        (hsbvals[2] > brightnessThreshold)){
      return true;
    }//end if
    return false;
  }//end isRedeye
  //--------------------------------------------------------//

  //The purpose of this method is to break up the harsh
  // outline of the circle used to change the color of
  // the pixels contained in the lasso.
  //This method averages the colors of a small group of
  // pixels at the left and right boundaries of the
  // lasso and sets the color of each pixel in the
  // group to the average color. The number of pixels in
  // the group is equal to the zoomFactor. The
  // probability is high that when the zoomed image is
  // restored to its original size, only one of these
  // pixels will be preserved.
  private void smoothEdges(){

    if(ellipse == null) return;

    //Smooth the left side of the circle.
    for(int row = 0;
             row < zoomDisplayPicture.getHeight();row++){
      for(int col = 0;
              col < zoomDisplayPicture.getWidth();col++){
        if(ellipse.contains(col + leftInset,
                            row + topInset)){
          averagePixels(col,row);
          //Break out of the inner loop and process the
          // nest row.
          col = zoomDisplayPicture.getWidth();
        }//end if
      }//end inner loop
    }//end outer loop
    //Smooth the right side of the circle. This code is
    // essentially the mirror image of the code used to
    // smooth the left side.
    for(int row = 0;
              row < zoomDisplayPicture.getHeight();row++){
      for(int col = zoomDisplayPicture.getWidth() - 1;
                                          col >= 0;col--){
        if(ellipse.contains(col + leftInset,
                            row + topInset)){
          averagePixels(col,row);
          col = -1;//Break out of inner loop.
        }//end if
      }//end inner loop
    }//end outer loop

  }//end smoothEdges
  //--------------------------------------------------------//

  //This method computes the average color of a group of
  // contiguous pixels on the same row and sets the color
  // of all of the pixels in the group to the average
  // color.
  //The number of pixels in the group is equal to the
  // zoomFactor.
  private void averagePixels(int col,int row){
    //Avoid coordinate out of bounds problem.
    if((col - zoomFactor/2) < 0){return;}

    //Initialize accumulators.
    int redTotal = 0;
    int greenTotal = 0;
    int blueTotal = 0;

    //Compute the sums of the color values.
    for(int cnt = col - zoomFactor/2;
                         cnt < col + zoomFactor/2;cnt++ ){
      pixel = zoomDisplayPicture.getPixel(cnt,row);
      redTotal += pixel.getRed();
      greenTotal += pixel.getGreen();
      blueTotal += pixel.getBlue();
    }//end for loop

    //Now set the color of the pixels to the average
    // color.
    for(int cnt = col - zoomFactor/2;
                         cnt < col + zoomFactor/2;cnt++ ){
      pixel = zoomDisplayPicture.getPixel(cnt,row);
      pixel.setRed(redTotal/zoomFactor);
      pixel.setGreen(greenTotal/zoomFactor);
      pixel.setBlue(blueTotal/zoomFactor);
    }//end for loop
  }//end averagePixels
  //--------------------------------------------------------//
```

```
  //This method draws a small white cursor when the user
  // clicks displayPicture with the mouse.
  private void drawCursor(){
    Graphics graphics = displayPicture.getGraphics();
    graphics.drawLine(anchorX - leftInset - 1,
                      anchorY - topInset,
                      anchorX - leftInset + 1,
                      anchorY - topInset);
    graphics.drawLine(anchorX - leftInset,
                      anchorY - topInset - 1,
                      anchorX - leftInset,
                      anchorY - topInset + 1);
    displayPicture.repaint();
  }//end drawCursor
  //----------------------------------------------------//
}//end class RedEye05
```

## Copyright

## About the author

**Richard Baldwin** *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is object-oriented programming using Java and other OOP languages.*

*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*

-end-