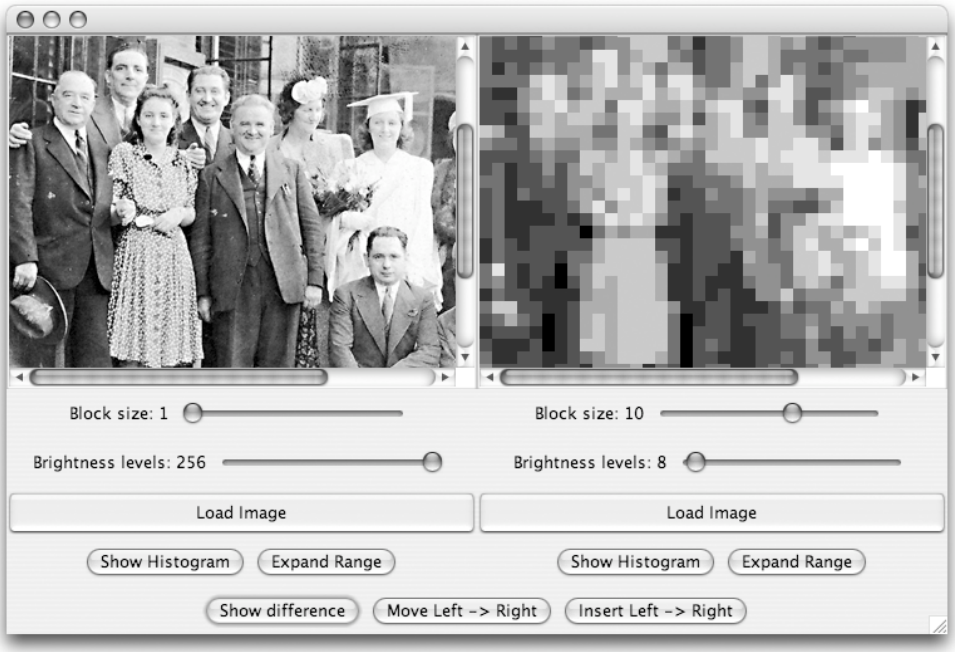
You Can Make a Difference!

Due April 11/12 (Implementation plans due in class on 4/9)

In last week's lab, we introduced some of the basic mechanisms used to manipulate images in Java programs. Now, we can use this knowledge to perform more sophisticated image manipulation operations.

The screen shot below shows the main window that the program you write this week should display.

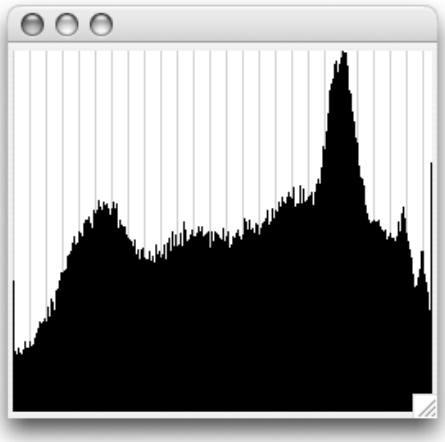


The program allows you to simultaneously manipulate two images. These images can be loaded from separate image files. The "Load Image" button on the left selects a new image for the left side of the display; the other "Load Image" button selects images for the right side.

In addition, the rightmost two buttons at the bottom of the window provide the means to either replace the image on the right with a copy of the image on the left or to add a copy of the image on the left to the upper left corner of the image on the right.

The sliders directly under the two images control the resolution with which the images are displayed in two ways. The first slider controls the effective spatial resolution with which an image is displayed. For example, the image on the left in the screen shot above is displayed with a block size of 1. This means every pixel from the original image is displayed separately. The image on the right, on the other hand, has its block size set to 10. This means that each block of 10 x 10 pixels from the original image is replaced in the display with a block of pixels whose color equals the average of the 100 pixels in the original block. If you look at this handout from far enough away, it should become clear that except for this difference in resolution the image on the right is the same as the one on the left.

The second slider under each image controls the color resolution with which the image is displayed. That is, it controls the equivalent of the color quantizer you implemented in the last lab. In the sample window shown above, the image on the left is displayed using 256 brightness values, while the image on the right is displayed using only 8 distinct brightness values.



Two additional buttons are found below each "Load Image" button. The "Show Histogram" button creates a new window containing a histogram of the brightness values associated with the displayed image's pixels. An example of such a histogram window is shown on the left.

The "Expand Range" button adjusts the brightness values of an image's color components so that they range all the way from 0 to 255. This operation has no effect on images with a histogram like the ones shown above since the pixel values already stretch from 0 to 255. This operation is valuable, however, on images that are either too dark (the largest brightness value is significantly less that 255) or to bright.

Finally, the only remaining button is the "Show Difference" button found at the bottom left of the window. If the two images displayed in the main window have identical dimensions, pressing this button creates an image with brightness values determined by the differences between the corresponding brightness values of the images displayed. This image is then displayed in a new, independent image viewer window like the one shown on the right. As an example, the differences between the full resolution and reduced resolution images shown in the sample window provided above are displayed in the window to the right.

Implementation Plan?

This week's lab handout is missing one very familiar feature. There is no "Implementation Plan."

This week, we want you to take a stab at making your own step-by-step plan for completing this program similar to the implementation plans we have presented in previous weeks. We will collect these plans in class on Monday. This is a dry run! We will provide you with copies of our implementation plan in class once you have turned in your own. The implementation plan that you turn in, however, will count as part of your final grade for this lab.

Your implementation plan should be about 1 or 2 TYPED pages. In preparing your implementation consider how you will test the correctness of the code you write in each step of the plan. That is, remember that one of your main goals in developing your plan is to ensure that as you develop the program there is a way to test the correctness of each addition you make before moving on to the next step.

Class Structure

The program you will write this week will be composed of 9 distinct classes.

GUIManager Classes

Your program will include three classes that extend GUIManager corresponding to the three type of windows described in the introduction of this handout. The main program window will be an instance of a class named `DualImageViewer`. The windows that display histograms will be instances of a class

named `DisplayHistogram`. The windows that display difference images will be instances of a class named `ImageViewer`. This last class will share many features with the `ImageViewer` class you wrote last week.

Like most of the main classes you have defined for other programs, the `DualImageViewer` class will have a constructor that expects no parameters. It will include a `buttonClicked` method and any private methods you find helpful.

The `ImageViewer` class will play a double role in this week's lab. If you compare the `DualImageViewer` shown in the first screen shot shown above to the `ImageViewer` used to display the difference image in the final screen shot, you might notice that the contents of the `DualImageViewer` look a bit like the contents of two `ImageViewer`'s positioned side-by-side. In fact, you will implement the `DualImageViewer` class by actually placing two `ImageViewers` next to one another within a single larger window. Just as you can say:

```
contentPane.add( new JTextArea( 10 ) );
```

or

```
contentPane.add( new JLabel( "Message:" ) );
```

you can also say

```
contentPane.add( new ImageViewer() );
```

because Swing recognizes that any class that extends `GUIManager` describes a type of GUI component.

When you reach the point that you want to place `ImageViewers` within a larger `DualImageViewer` window, there is one change you will have to make to your `ImageViewer` class. If you plan to place an `ImageViewer` in another window, it should not try to create a window of its own! Therefore, when you are ready to define your `DualImageViewer` class, you should remove the invocation of `createWindow` from the constructor of the `ImageViewer` class. You will still need a way to make the `ImageViewer` class create a separate window. You will want to do that when the user presses the "Show Difference" button. To accomplish this, you will create a new `ImageViewer` and then tell it to create a new window by invoking its `createWindow` method explicitly from outside the constructor. For example, you might say something like:

```
ImageViewer newDiffWindow = new ImageViewer();
newDiffWindow.createWindow(DIFF_WINDOW_WIDTH, DIFF_WINDOW_HEIGHT);
```

One interesting property of classes that extend `GUIManager` is that events associated with GUI components are handled by executing code within the class that describes the portion of the program's window immediately containing the GUI component. For example, in your final program, the "Load Image" buttons will still be created and added to the content pane within the code of the `ImageViewer` class. On the other hand, the "Show Difference" button that appears at the bottom of the `DualImageViewer` is created and added to the content pane within the constructor of the `DualImageViewer` window. As a result, when a user clicks on the "Show Difference" button, the code included in the `buttonClicked` method definition found within the `DualImageViewer` class will be executed. By contrast, when a user clicks on the "Load Image" button, the code found in the `buttonClicked` method defined within `ImageViewer` will be executed instead.

The `ImageViewer` class will therefore have to include `buttonClicked` and `sliderChanged` methods to respond to user GUI actions. In addition, it must provide two other public methods. When a user clicks on the buttons used to copy the image in the left `ImageViewer` to the right `ImageViewer`, the

code in the `buttonClicked` method of the `DualImageViewer` executes. To write appropriate code in this method, you will need some way to access the image displayed by the `ImageViewer` on the left and some way to set the image to be displayed on the right. As a result, the `ImageViewer` class must provide methods to get and set the images displayed.

We want you to be able to adjust the block size slider to reduce the resolution of an image and then set the block size back to 1 and see the original image. Similarly, it should be possible to undo the effect of the "Brightness levels" slider. Therefore, each `ImageViewer` should have two `SImage` variables. One variable should refer to the `SImage` originally placed in the `ImageViewer`. The other should refer to the `SImage` that is currently being displayed. Each time either of the sliders is changed you should start with the original image and compute a new image to display. When asked to display a histogram or expand the range of an image, the `ImageViewer` should show the histogram or expand the range of the displayed image rather than the original. The method used by the `DualImageViewer` to place a new image in an `ImageViewer` should set the "original." On the other hand, the method that gets an image from an `ImageViewer` should return the displayed image.

The final `GUIManager` class you will define is `DisplayHistogram`. This class depends on a separate `Histogram` class that holds the data describing a single histogram. The constructor of the `DisplayHistogram` class should take the `Histogram` to be displayed as a parameter. It will not have any public methods.

You will draw the lines of the histogram by setting the values in a pixel array much as you drew a border around the gray rectangles displayed briefly during last week's lab. The rectangle you drew last week was 256 pixels wide and 100 pixels high. Your histogram image should also be 256 pixels wide, but it will probably be best to make it taller than 100 pixels. A height of 200 pixels will probably work well.

Your code should draw one vertical line for each possible brightness value from 0 to 255. The height of the line you draw for brightness level *b* will depend on three values:

- the number of pixels in the image that had b as their value,
- the height you picked for the histogram image, and
- the number of pixels in the image that had the most common brightness value.

It is probably clear why the first two values matter. The third value matters because you will want the length of the vertical line you draw for the most common brightness value to extend from the top to the bottom of the image you create. Therefore, you will want to determine the length of the line you draw for brightness *b* by first multiplying the height of the image and the number of pixels of brightness *b* and then dividing the result by the number of pixels of the most common brightness level.

It will be much easier to draw the complete histogram if you define a method that takes a pixel array together with the x coordinate and the desired height of a line as parameters and modifies the array by "drawing" the desired line.

The Histogram Class

You should also define a class named `Histogram` to hold the array of pixel counts that provide the data used to draw a histogram. The constructor for the `Histogram` class should take a pixel array as a parameter. The constructor should create an array of 256 integer values. It should include a loop that will examine all the elements in the pixel array, setting each of the 256 elements of the histogram array so that the element at index b is equal to the number of entries in the pixel array that are equal to *b*.

The histogram class should provide two methods. One of the methods should take a brightness value, b, and return the count stored at position b in the histogram array. The other method should return the largest count stored in the histogram array.

Image Filter Classes

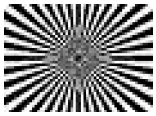
There will be five other classes included in your program that don't create windows of their own. In last week's lab, we had you write a separate class that extended GUIManager for each of the two operations (darkening and color quantization) we had you implement. Each of these classes not only contained code to perform the desired image operation, they also contained code to display the result of the operation. This week, when you perform an operation on an image displayed within an `ImageViewer`, the result will not appear in a new window. Instead, your code will replace the original image displayed within the `ImageViewer` with the transformed image. We will, however, have you write a separate class to hold the code that performs each of these operations. These classes behave somewhat like the interchangeable lenses and filters you can place on a high-quality camera. Accordingly we will refer to these classes as *image filters*. All of these classes will have a similar structure. In particular, they will all define a method named `filter` which takes an `SImage` as a parameter and returns a transformed version of that image.

In this lab, you will use five image filters (but luckily you will only have to write three of them):

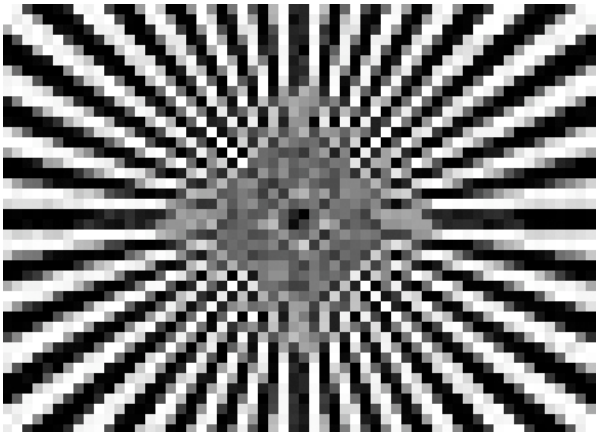
`Quantizer` - The `Quantizer` class will implement the operation you included in your `Quantizer` class in the last lab. We will provide you with the code for this class as an example of how all your filter classes might be implemented. The constructor for this class takes the number of levels of brightness that should be used in the transformed image as a parameter. More information on this class can be found in the section entitled "But the Meek Shall Inherit..." below.

`Blocker` - The `Blocker` class will divide an image into *n* x *n* blocks of adjoining pixels, compute the average color of each block of pixels and replace all of the pixels in the block with this average color. The number *n* to use as the size of the blocks will be provided as a parameter to the constructor. Applying this filter to an image has the effect of decreasing the spatial resolution of the image.

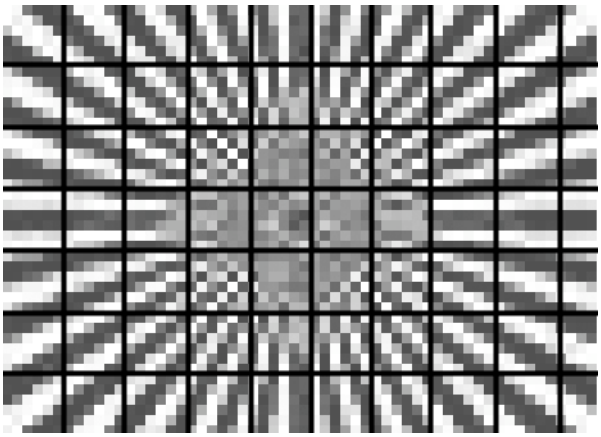
The idea is fairly simple. Suppose that we start with an image like the one shown below:



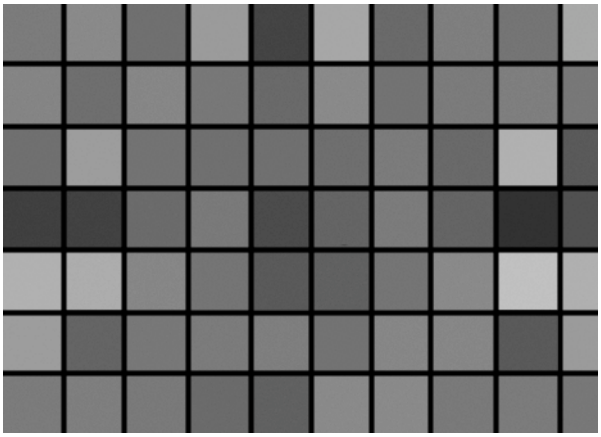
If we zoom in on this image a bit, we can see its individual pixels:



Now, to see how we can block this image, imagine placing a grid over the image to group the pixels into 6x6 squares (or, at the edges where we run out of pixels, rectangles of as many pixels as remain):



Next, compute the average brightness of the pixels in each cell of this grid and then replace the colors of all the pixels within each cell with the average brightness. Looking through our imaginary grid, the new image will look like:



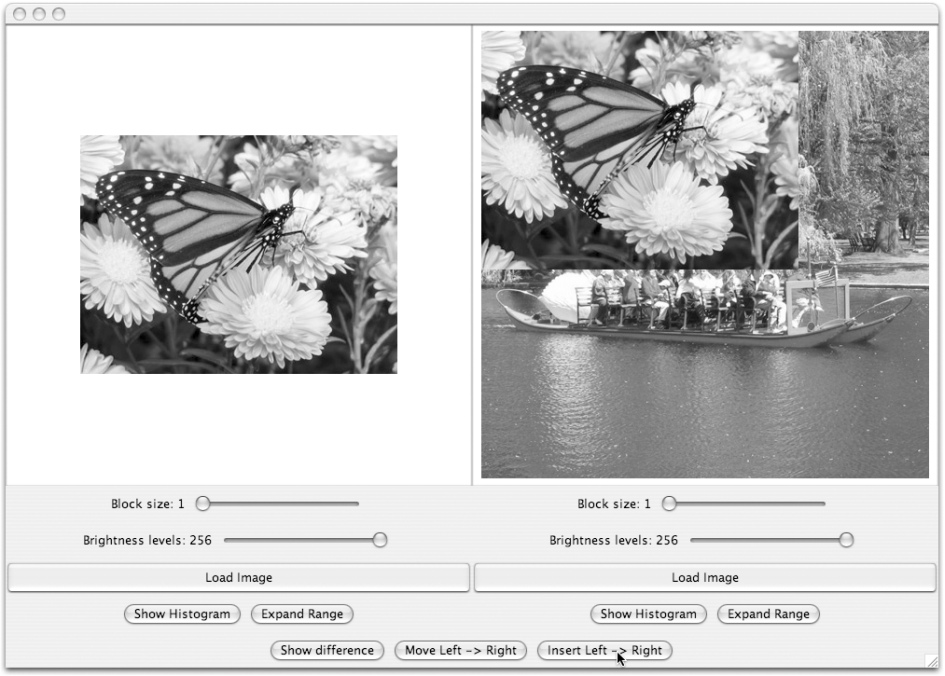
Without the grid and reduced to its original size, the resulting image might look like:

When defining this class, consider defining two helper methods. Both methods should expect a pixel array, the x and y coordinates of the upper left corner of a block and the desired block size as parameters. The first method should return an int equal to the average of the values found in the block. The second method should take a brightness value as an additional parameter and replace the values in the block with this new brightness value. In both of these methods, you should deal with the possibility that you are so close to the bottom or right edge of the image that there are not enough pixels to form a complete block.

`Expander` - The `Expander` class will adjust the brightness of the pixels of an image so that for each of the primary colors, the smallest brightness value is 0 and the largest is 255. To do this, the code in this class will have to determine the range of brightness values used within the original image.

The constructor for this class expects no parameters. You should write two helper method that find the minimum and maximum values stored in a pixel array as part of the definition of this class.

`Paster` - The `Paster` class implements a very limited form of the "paste" operation used when "cutting and pasting" with an image editor. The constructor for this class will take an `SImage` as a parameter. Then, when the `filter` method is applied to a second image, the resulting image will be formed by pasting a copy of the first image into the upper left corner of the image passed to `filter`. As an example, the window below show the result of pasting (for no apparent reason) a butterfly into a picture of a swan boat from the Boston Public Garden.



The implementation of `Paster` is somewhat different from the first three filter classes because it involves working with two images simultaneously. As a result, we will provide you with the code for this class as an example of how to implement such a filter. You will find this helpful while implementing...

`Differencer` - The `Differencer` class creates an image with pixel values equal to the difference between two other `SImages`. Its constructor takes an `SImage` as a parameter. Applying its filter method to a second image produces an `SImage` whose brightness values will be the differences between the brightness values of corresponding pixels of the images provided as parameters, as long as the two images have identical dimensions. If the images provided do not have identical shapes, then `filter` returns `null`.

The difference between corresponding pixels of two images can range anywhere from -255 to 255. The brightness values of an image are supposed to fall between 0 and 255. Since we will want to display the differences you compute as images, your `Differencer` should actually compute the absolute values of the differences between pixels. In Java, the absolute value of a value can be produced by evaluating the expression `Math.abs( value )`.

But the Meek Shall Inherit...

In last week's lab, you might have noticed that the `AdjustLevels` and `Quantizer` classes had very similar structures. For each class, you defined an `adjustPixelArray` method that performed the appropriate operation on one color layer. Then, in each class you defined a `sliderChanged` method that invoked the `adjustPixelArray` method on all of the color levels and combined the results to form a

new `SImage`. The `sliderChanged` methods in the two different classes were identical. Also, except for their names, the constructors for these classes were identical.

Java provides a mechanism to define classes like these that share sections of identical code without repeating the shared code. The mechanism is called *inheritance* or *extension*. The word `extends` in the phrase `extends GUIManager` tells Java that a class is being defined as an extension of the GUIManager class provided in the Squint library. Such a class is said to inherit shared features like the definition of the `createWindow` method from the class included in the library. In this lab, we would like to show you how to use inheritance to simplify the definitions of the five filter classes described above.

To understand how this can be done, consider the definition of the `Quantizer1` class shown below:

```
public class Quantizer1 {
    // Number of brightness levels desired
    private int levels;

    // Create a filter that will quantize to a specific number of brightness levels
    public Quantizer1( int theLevels ) {
        levels = theLevels;
    }

    // Given an image, produce a new image that uses the desired number of levels
     public SImage filter( SImage original ) {
        return new SImage( layerFilter( original, SImage.RED ),
                           layerFilter( original, SImage.GREEN ),
                           layerFilter( original, SImage.BLUE )
                           );
    }

    // Adjust the levels for the brightnesses for a specific color layer in an image
    public int [][] layerFilter( SImage orig, int layer ) {
        return pixelFilter( orig.getPixelArray( layer ) );
    }

    // Adjust the number of distinct brightness levels used in a pixel array
    public int [][] pixelFilter( int [][] shades ) {
        int bandWidth = 256/levels;
        int round = (bandWidth-1)/2;
        for ( int x = 0; x < shades.length; ++x ) {
            for ( int y = 0; y < shades[0].length; ++y ) {
                shades[x][y] = shades[x][y]/bandWidth*bandWidth + round;
            }
        }
        return shades;

    }
 }
```

The code in the `pixelFilter` method defined in this class should seem very familiar. It is basically the same as the `adjustPixelArray` method you defined in your `Quantizer` class last week. The only differences are that it accesses the number of levels desired as an instance variable rather than a formal pa-

rameter and it uses `shades.length` and `shades[0].length` to determine the dimensions of the pixel array.

Now, let's consider how this method gets used. When the `filter` method defined in this class is applies to an `SImage`, it invokes a method named `layerFilter` on the image three times, passing a different color level to `layerFilter` each time. The `layerFilter` method in turn extracts one of the three color pixel arrays from the image and passes it to `pixelFilter`. Thus, this class ultimately processes the image just as you did last week. It applies `pixelFilter` to each of the three color layers and combines the results to form a new image.

Suppose that we wanted to define a similar filter to replace the `AdjustLevels` class you implemented last week. We would have to define a new version of `pixelFilter`, but all of the other components of the new class would be identical to the `Quantizer1` class. With this in mind, consider the following alternate approach to defining `Quantizer` using inheritance.

First, we define a completely silly class with a filter method that always returns an `SImage` identical to the `SImage` it was given as a parameter:

```
public class ImageFilter {

    // Create a filter
    public ImageFilter( ) {
    }

    // Given an image, produce a copy
     public SImage filter( SImage original ) {
        return new SImage( layerFilter( original, SImage.RED ),
                           layerFilter( original, SImage.GREEN ),
                           layerFilter( original, SImage.BLUE )
                           );
    }

    // Extract the specified color pixel array
    public int [][] layerFilter( SImage orig, int layer ) {
        return pixelFilter( orig.getPixelArray( layer ) );
    }

    // Return the provided pixel array unchanged
    public int [][] pixelFilter( int [][] shades ) {
        return shades;
    }
 }
```

Note that although this class is silly, many of its parts are identical to parts of the `Quantizer1` class shown above. In particular, the `filter` and `layerFilter` methods are unchanged.

Although this class seems silly when examined in isolation, it becomes very useful when we want to define other, similar classes like `Quantizer`. Below, we show a second version of the definition of `Quantizer` that specifies in the class header that it `extends ImageFilter`.

```
public class Quantizer2 extends ImageFilter {
    // Number of brightness levels desired
    private int levels;

    // Create a filter that will quantize to a specific number of brightness levels
    public Quantizer2( int theLevels ) {
        levels = theLevels;
    }

    // Adjust the number of distinct brightness levels used in a pixel array
    public int [][] pixelFilter( int [][] shades ) {
        int bandWidth = 256/levels;
        int round = (bandWidth-1)/2;
        for ( int x = 0; x < shades.length; ++x ) {
            for ( int y = 0; y < shades[0].length; ++y ) {
                shades[x][y] = shades[x][y]/bandWidth*bandWidth + round;
            }
        }
        return shades;

    }
 }
```

When we say that one class extends another, we are telling Java to act as if the new class contains definitions of all the methods, variables, and constructors found in the existing class for which no replacement is provided in the new class. Since our new `Quantizer2` does not contain definitions of the filter or `layerFilter` methods, it will inherit the versions of these methods found in `ImageFilter`. On the other hand, since it does contain definitions for `pixelFilter` and a constructor, these will be used instead of the versions found in `ImageFilter`. We say that these definitions *override* the corresponding definitions in the `ImageFilter` class. The result is that `Quantizer2` and `Quantizer1` behave identically.

We will provide you with copies of the `ImageFilter` and `Quantizer` classes together with a version of `ImageViewer` that uses them in a starter project for this lab. You should mimic our approach to defining `Quantizer` as an extension of `ImageFilter` when defining your `Blocker` and `Expander` classes.

Our starter project also contains a definition of the `Paster` class described above that extends `ImageFilter`. This class overrides the `layerFilter` method rather than the `pixelFilter` method. This is because the filter this class defines depends on comparing pixel arrays from two distinct images. You should use a similar technique to define your `Differencer` class as an extension of `ImageFilter`.

Layout Issues

In last week's lab, we began using the `BorderLayout` manager so that the `JLabels` we used to display images would be stretched to fill the windows that contained them as we stretched those window's with the mouse. Your `DualImageViewer` class should also use a `BorderLayout` manager. In the `SOUTH` of the window, you will place a `JPanel` used to hold the "Show Difference" and "Left --> Right" buttons. In the `CENTER` you will place another `JPanel` used to hold two `ImageViewers`.

We want the `ImageViewers` to be stretched to fill the `JPanel` that holds them. To do this, you should associate a `GridLayout` manager with this `JPanel`. As its name suggests, a `GridLayout` manager divides the area within a `JPanel` into a grid and places each component added to the panel in one cell of the grid. When you construct a `GridLayout` manager, you tell it how many rows and columns to include

in the grid. Therefore, if `viewerPanel` is the name of the panel that is to hold your two `ImageViewers`, you should say

```
viewerPanel.setLayout( new GridLayout( 1, 2 ) );
```

before adding the `ImageViewers` to the `JPanel`.

We will also use a `GridLayout` within the `ImageViewer` class to arrange the sliders and buttons displayed by each `ImageViewer`. This grid will have one column and four rows, so we created it by saying

```
new GridLayout( 4, 1 )
```

The first row holds the slider that determines the spatial resolution with which the image should be displayed. Actually, since we want to place a label like "Block size: 1" together with the slider, the first row will hold another `JPanel` that will in turn hold both a `JLabel` and the `JSlider`. Similarly, the second row holds a panel containing the slider that controls the number of distinct brightness levels used in image pixel arrays together with its label. The third row holds the "Load Image" button, and the fourth row holds a `JPanel` that in turn holds the remaining buttons.

Getting Started

We will provide you with some of the code needed to complete this program in a starter project that you can download from the course web site. You should begin by downloading this starter project and opening it with BlueJ:

- Use a web browser to visit the labs section of the course web page. Download the .zip file for this lab into your Documents folder. The starter folder must be placed in your Documents folder because some of its code assumes that it will reside in the same folder where you place the `AllImages` folder from last week.
- Double-click on the .zip file to unpack it. This will create a folder named `Lab8Starter`.
- Change the folder's name to something like `Lab8Floyd` that includes your name and Lab8 (but no blanks).
- Open the renamed folder as a project with BlueJ.

Submission Instructions

As usual, make sure you include your name and lab section in a comment in each class definition. Find the folder for your project. Its names should be something like FloydLab8.

- Click on the Desktop, then go to the "Go" menu and "Connect to Server."
- Type "cortland" in for the Server Address and click "Connect."
- Select Guest, then click "Connect."
- Select the volume "Courses" to mount and then click "OK." (and then click "OK" again)
- A Finder window will appear where you should double-click on "cs134",
- Drag your project's folder into either "Dropoff-Monday" or "Dropoff-Tuesday".

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. The Mac will not let you submit again unless you change the name of your folder slightly. Just add something to the folder name (like the word "revised") and the re-submission will work fine.

Grading

Implementation Plan (5 points)

Completeness (12 points) / Correctness (6 points)

- Blocker class replaces blocks of pixels with averages
- Blocker class handles incomplete blocks at edges
- Expander determines minimum and maximum brightness values
- Expander adjust brightness values proportionately to stretch from 0 to 255
- Differencer computes absolute values of pixel color differences
- ImageViewer GUI interface displays correctly
- ImageViewer transforms images correctly
- DualImageViewer GUI interface displays correctly
- DualImageViewer GUI interface functions correctly
- Histogram class computes pixel counts correctly
- DisplayHistogram displays an mages of a histogram

Style (7 points)

- Commenting
- Good variable names
- Good, consistent formatting
- Correct use of instance variables and local variables
- Good use of blank lines
- Uses names for constants