Lab 7

Digital Image Processing

Due October 4/5, 11PM

In this week's lab, you will construct two programs to implement two digital image processing tasks:

1. reducing the number of distinct colors used to display an image, and
2. adjusting the intensity of the colors in an image.

As we did last week, we encourage you to work with a partner while completing these programs.

The interface that the first of these programs should display is shown on the left. Two buttons are located at the top of the program's window. The first loads a new image. When it is pressed, a dialog box opens to allow the user to select an image file. That image is then displayed in the center of the program's window. The second button causes the computer's built-in camera to take a picture and display it in the window.

Once an image has been chosen, the user can adjust the number of distinct brightness levels used to display the image by changing the setting of the slider displayed at the bottom of the window. The window on the left shows an image displayed using 256 brightness levels (the standard for grayscale images). The window below show the image displayed using only 3 levels. (The couple in the images shown on this page are one of your instructor's grandparents. Can you see the family resemblance?)

The interface for the second program will be quite similar, but the slider will control the overall brightness of the image displayed rather than the number of levels used.

When your lab is complete, it will consist of one class that displays a window with the control buttons and one additional class for each of the image processing operations.

This handout is intended to function as a tutorial. It provides detailed instructions on how to construct the program we have described, introduces the details of several new library classes and methods, and explores two new feature of the Java language itself.

The new topics introduced include:

- A feature of Java called arrays. Arrays are tables of values that we will use to manipulate the collections of values that describe the colors and brightesses of the pixels that make up an image.
- The use of inheritance in Java programs. Many of the

classes you have defined in previous weeks have included the phrase "extends GUIManager". This week, we will see that it is possible to extend other classes.

- A new class named `SImage` that supports the manipulation of images.
- The ability to use a `JLabel` to display an image.
- A new class named `Camera` that provides access to images taken by the computer's iSight camera.
- A new Swing class of GUI components named `JSlider` that provides a convenient way for a user to specify a numeric value by sliding a virtual knob back and forth.
- Another Swing class named `JFileChooser` that makes it easy to create a dialog box that can be used to navigate around the computer's file system to choose files including image files.

Step 0: Getting Started

While working on this lab, you will need some image files to use as samples. We have provided a folder of such images as a .zip file which you can download from the course web page. To make it easy for the program you will write during lab to find these images, they should be stored within the "Documents" folder for your account (i.e., the folder that contains your BlueJ project folders), but **not** inside your BlueJ project folder for this week. Therefore, to start your work:

- Launch BlueJ and create a new project within your "Documents" folder whose name contains "Lab7" and your name (and, as usual, no blanks).
- Launch Safari and go to the "Labs" page of the CS 134 web site. (You can use another web browser if you prefer, but the following instructions are written for Safari.)
- Find the link that indicates it can be used to download the `AllImages.zip` file for this lab.
- Point at the link. Hold down the control key and depress the mouse button to make a menu appear.
- Select the "Download Linked File As ..." item from the menu.
- Navigate to your "Documents" folder and save the `AllImages.zip` file in that folder. (Click on the arrow to the right of the "Save As:" text field to make the navigating process easier.)
- Return to the Finder, locate the `AllImages.zip` file in your "Documents" folder, and double-click on it to create an `AllImages` folder and then drag the .zip file into the trash.

Step 1: Accessing and Displaying an Image File

As a start, let's construct a program that displays just one of the buttons that will appear in your final program's control window, the "Load Image" button.

- Use BlueJ to add a new GUIManager class named `ImageViewer` to your project.
- Delete all of the GUI event handling methods that BlueJ provides in its `GUIManager` template except for the `buttonClicked` method.
- Add the two import statements

```
import java.awt.*;
import java.io.File;
```

  to the beginning of the new file.
- Add code to display two GUI components: `JButton` with the label "Load Image", and a `JLabel` that displays a short message ("Hi" will do). Associate instance variables with both the `JButton` and the `JLabel`. Even though there is only one button now, place it in a `JPanel` since you will eventually have two buttons and want them to all be displayed together. Add the `JPanel` to the content pane before the button.
- Run the program to make sure it displays the label and button as desired.

Now, we want to add code to your `buttonClicked` method to load and display an image. Ultimately, the person running the program should be able to select any image file, but to start let's write code that will always load the image stored in the "`AllImages/MMM.gif`" file.

There are two steps you need to perform to get this image displayed in your program's window.

First, you have to tell Java to convert the data in the image file into an object that can be used to describe the image within your program. You can do this by constructing an object of a new class named `SImage`. You can construct a new `SImage` by including the name of a file on your disk as a parameter in the construction. You can therefore load the image into your program by declaring a variable of type `SImage` and adding a statement of the form:

```
someImageVariable = new SImage( "../AllImages/MMM.gif" );
```

to the `buttonClicked` method. The `SImage` variable should be declared as an instance variable. Although you will only use it in the `buttonClicked` method at this point, as you complete this class definition you will need to access the image in other methods.

Second, you have to tell Java to display the image within a GUI component in your program's window. That is why we had you include a `JLabel` in your program's interface. `JLabels` can display images as well as text. Just as you have used the `setText` method to make a `JLabel` display a piece of text, you can use a method named `setIcon` to make a `JLabel` display an `SImage`. Thus, an instruction that looks something like:

```
someJLabel.setIcon( someImageVariable );
```

should be included in your `buttonClicked` method to tell Java to display the image you loaded.

- Add the statements and the `SImage` instance variable described above to your program.
- Run the program and click on the "Load Image" button.

If you did everything right, your program should display the picture shown on the first page of this handout. Don't they make a lovely couple?

Even if you did everything right, you will notice that your program's interface has some obvious shortcomings. If you stretch the window to make it larger and smaller you will notice that when it is too small, there isn't enough room to display the image and when it is too large, you can see the text you displayed in the `JLabel` in addition to the image. Make the following changes to address these issues.

- Change the text displayed in the `JLabel` to be the empty string ("").
- Place the `JLabel` within a new `JScrollPane` so that scroll bars will be provided when the window is too small. That is, use an instruction of the form:

  ```
  contentPane.add( new JScrollPane( yourLabel ) );
  ```

  to add it to the display.
- Change the layout manager associated with the content pane to be a border layout manager. when you use a border layout manager, you have to add a second parameter to each invocation the content pane's add method to specify the area in which the component being added should be placed. There are 5 choices: `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.EAST`, and `BorderLayout.CENTER`. Only one item can be placed in each of these five areas. The item in the center is stretched to fill any space not used by the other four regions. So, you must perform all three of the following steps before you attempt to run your program again:

  1. Add the instruction

     ```
     contentPane.setLayout( new BorderLayout() );
     ```

     to your program's constructor before any instructions that add components to the `contentPane`.

2. Place the `JLabel` in the center of the window by modifying the command to add it to the content pane by adding a parameter specifying its desired position:

```
contentPane.add(  ... , BorderLayout.CENTER );
```

3. Similarly modify the command that adds the `JPanel` containing the "Load Image" button so that the panel is placed at the top of the window (`BorderLayout.NORTH`).

Run the revised program. Things should look pretty good now. The only remaining issue is that when the window is wider than required to display the image, the image will be left-justified rather than centered in the window. You should fix this problem by adding a second parameter to the constructor for your `JLabel` so that it looks like:

```
new JLabel( "", SwingConstants.CENTER )
```

This additional parameter tells the label to center its contents. If you run the program after making this change it should display the image just right.

Step 2: Using a `JFileChooser`

Now, we would like to make your program flexible enough to be able to display any image file. Swing provides a class named `JFileChooser` that makes it fairly easy to do this.

- Add an instance variable declaration of the form

```
private JFileChooser chooser =
        new JFileChooser( new File( System.getProperty("user.dir")) +
                          "/../AllImages" );
```

  to your `ImageViewer` class.

- Place the statement

```
chooser.showOpenDialog( this );
```

  before the code in your `buttonClicked` method that loads the `MMM.gif` picture.

- Replace the String literal "`../AllImages/MMM.gif`" used as a parameter for the `SImage` construction with the expression:

```
chooser.getSelectedFile().getAbsolutePath()
```

- Run your program and click on the "Load Image" button. A dialog box should appear.
- Select any image file you want using the dialog box.

The instance variable declaration we told you to add tells Java to create a file chooser. When you first create a `JFileChooser`, nothing appears on the screen. Invoking the `showOpenDialog` method on this object causes it to display a dialog box and wait for the user to select a file. The parameter

```
new File(System.getProperty("user.dir")) + "/../AllImages"
```

provided when we create the chooser tells Java that the dialog displayed should start in the folder containing all the sample images we had you download from the course web page (in a file name, "`..`" means to go up one level in the hierarchy of folders). The invocation

```
chooser.getSelectedFile().getAbsolutePath()
```

returns a string describing the file selected by the user.

When you ran the program, you might have noticed one issue you must address to make your program more robust. The dialog box a file chooser displays includes a "Cancel" button. If the user presses cancel, then the `getSelectedFile` method will not be able to return a description of the file. To enable you to

handle this situation, the `showOpenDialog` method returns a value indicating whether or not a file was selected by the user. The value returned when a file is selected is associated with the name `JFileChooser.APPROVE_OPTION`. Therefore, you should combine the line that invokes `showOpenDialog` and your code to get the image to create an if statement of the form:

```
if ( chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION ) {
        // ... code to get and display image
}
```

- Add such an if statement to your `buttonClicked` method. (Note: After you do this the invocation of `showOpenDialog` in the if statement should be the only invocation of this method in your code.)
- Run the program again after making this change to ensure that it handles the cancel button correctly.

Step 3: Say Cheese!

While the AllImages folder we provided for this lab contains many interesting images, nothing could possibly compare with a picture of your own smiling face. To give you the pleasure of seeing that sight, we will now add the ability to take pictures using the computer's built-in iSight camera to your program.

First, you need to add a button to tell the computer to take a picture. This new button should say something like "Take Snapshot". It should appear at the top of the program's window with the "Load Image" button as shown in the sample windows on the first page of this handout. Both buttons will need to be associated with instance variable names so that you can tell which one was clicked. You will also need to modify your `buttonClicked` method to respond differently to the two buttons.

- Add the new button to the display as described above.
- Add an `if` statement in your `buttonClicked` method so that the code to load an image will only be executed if the "Load Image" button was pressed.
- Run your program to verify that the buttons appear as expected, that the "Load Image" button still works, and that the "Take Snapshot" button does nothing. Correct your code and re-run if necessary.

The Squint library provides a class named `Camera` that makes it easy to take pictures. The constructor for the class does not require any parameters, so you can create a `Camera` by evaluating a construction of the form:

```
new Camera()
```

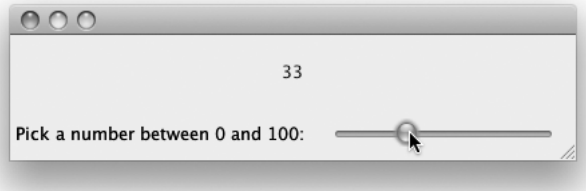
The class provides a method named `getImage` that returns an `SImage` representing the latest picture captured by the camera. So, to see your smiling face all you have to do is:

- Add a new instance variable of type `Camera` to your program.
- Initialize this variable to refer to a new `Camera` either in its declaration or in your constructor.
- Add a branch to the `if` statement in `buttonClicked` to respond to the "Take Snapshot" button by invoking `getImage` on your camera and displaying the picture in the center of your program's window by using `setIcon` to replace any image currently displayed. You should also associate the new image with the SImage instance variable used when an image file is loaded.
- Run and test your program.

Ham it up for a while before going on to the next step.

Step 4: Slippery Sliders

In the images on the first page, we showed windows containing a GUI component we have not previously used, the `JSlider`. A `JSlider` is a component that provides a simple way for a user to graphically select a value by sliding a knob within a bounded interval. The slider in the example window on the first page of



this handout allows the program's user to determine how many shades of gray will be used to display an image. To introduce you to sliders, we would like you to implement a program that uses a slider in a somewhat less sophisticated way.

The interface for the program we want you to write is shown on the next page. Its purpose is simply to let a user select a number between 0 and 100 using a `JSlider` and to display the number currently selected. The number selected is determined by the relative position of the slider's knob. In the figure on the right, the knob is about one third of the way from the left end of the slider, so the value displayed is 33, one third of 100.

This program displays three components. The number in the middle of the screen that shows the current value of the slider is a `JLabel`. The text "Pick a number between 0 and 100:" that appears before the slider is not part of the `JSlider` itself. It is another independent `JLabel`. The slider that appears on the bottom right of the window is a member of the class named `JSlider`.

The `JSlider` constructor expects three integer parameters. The first two parameters specify the range of values to be associated with the positions of the slider's control. We want your slider to describe a number between 0 and 100, so these values will be used as the first two parameters. The last parameter specifies the initial value for the slider. We will start out with the slider centered. As a result, you will use a construction of the form

```
new JSlider( 0, 100, 50 )
```

to create your program's slider.

While we want you to write this as a standalone program, by the end of the lab we will incorporate its code as part of a program that allows its user to reduce brightness levels used to display an image as described on this handout's first page. Accordingly, you should name this program `AdjustBrightness`.

When used in this larger program, we will want the window's content pane to be managed by a border layout manager. Rather than wait, we would like you to start using a border layout manager now.

- Use the "New Class.." button to create a new class that extends `GUIManager`. Name the new class `AdjustBrightness`.
- Delete all of the event handling methods in the template that BlueJ provides for this class.
- Add an import statement for `java.awt.*`.
- Write a constructor that displays the basic elements of the program's interface without actually making them work. You should associate instance variable names with all three of the GUI components used.
  - Set the layout manager for the program's `contentPane` to be a new `BorderLayout` (We provided detailed instructions on how to do this for you `ImageViewer` class a bit earlier in this handout).
  - Place the `JLabel` used to display the slider's current value at `BorderLayout.CENTER` and add the value `SwingConstants.CENTER` as a second parameter in its construction.
  - Place the slider and its label at `BorderLayout.SOUTH`. Actually, you can only put one component in the `SOUTH` of a panel with a `BorderLayout`. Therefore, you will have to create a `JPanel` to hold the `JSlider` and the `JLabel` that displays the range of the `JSlider` and then place this additional `JPanel` in the `SOUTH` region of the content pane.
- Run your program to verify that you have created and correctly displayed all the desired components. Now, we just have to make the slider work.

Whenever a slider is adjusted, any code you place in a method named `sliderChanged` will be executed. You should define such a method. Its header will look like:

```
public void sliderChanged( ) {
```

Within its body you should place code to update the text of the `JLabel` displayed in the center of your program's window.

You can determine the value associated with a slider by using a method named `getValue`. This is an accessor method that returns the integer value associated with the current position of a slider's control. For example, the statement

```
int currentValue = someSlider.getValue();
```

could be used to associate the updated value of a slider with an integer variable.

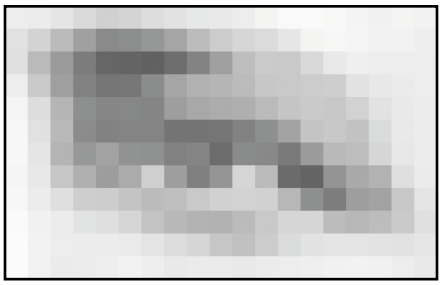
- Define a `sliderChanged` method that will update the number displayed each time the slider's knob is moved. You will need to concatenate the `int` value with `""` when you pass it to the `setText` method.

Step 5: Shades of Gray

To represent an image in a computer, the contents of the image must somehow be represented in binary. This is accomplished by dividing the picture into a grid of tiny squares called pixels and then using numbers encoded in binary to describe the color of each pixel. For color images, several numbers are used for each pixel. For grayscale (or monochrome) images, things are a bit simpler. A single number can be used to describe the brightness of each pixel. Small numbers describe dark pixels. Larger numbers are used to describe brighter pixels. The brightness numbers are ultimately encoded in binary using 8 bits. As a result, the largest number allowed is 255 which is one less than 2^8. The number 255 is used to describe white pixels. A black pixel's shade is 0. A dark gray pixel might have a brightness value of 80 and a light gray pixel's brightness might be 190.

For example, if you look closely enough at the left eye of the gentleman pictured on the front of this handout:

you will discover that it is really composed of distinct squares of varying shades of gray as shown below:

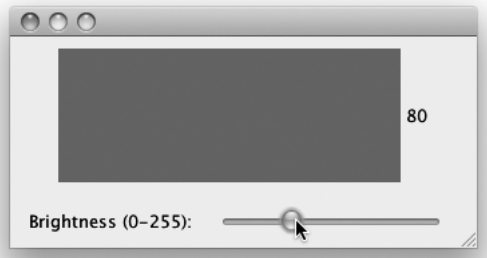


Within the computer, each of these gray squares is represented by a single number describing its brightness. The table below, shows all of the numeric values that would be used to describe the image of the eye.

233	232	223	210	198	202	214	219	226	229	233	240	248	248	247	241	238	242	239
218	201	173	143	120	122	131	148	173	194	204	212	224	242	246	242	239	239	235
218	172	137	106	83	81	78	91	114	143	177	190	190	208	229	234	232	233	233
234	188	142	110	99	100	120	148	162	167	174	188	189	192	196	219	226	229	233
242	213	166	123	115	116	115	141	151	158	160	173	185	192	189	201	219	229	231
245	218	172	120	112	112	113	97	97	98	111	124	147	184	189	182	210	228	232
247	223	165	134	146	129	126	109	113	91	120	118	104	136	172	175	204	225	232
246	226	186	161	139	155	197	141	108	131	204	173	89	80	130	153	165	215	230
246	231	215	196	194	183	173	178	189	199	201	198	168	125	107	139	147	187	222
248	238	233	220	215	204	183	168	163	164	174	188	205	205	175	168	175	191	217

250	239	233	231	231	230	221	213	205	188	181	193	211	219	221	222	221	219	224
250	241	230	231	231	239	234	228	230	230	228	231	231	234	238	239	238	232	227

Earlier, you constructed an `SImage` by providing a file name to the `SImage` constructor. You can also construct an `SImage` by providing a collection of numbers describing the brightness of the pixels of the desired image. In general, this can be difficult since many values may be required to describe even a relatively small image. There is, however, one special case where it is quite easy to do. This is the case where all of the pixels should have the same brightness.

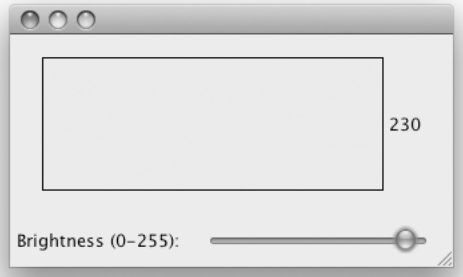


To illustrate this, we would like you to modify your `Adjust-Brightness` program so that it creates and displays a monotone image whose brightness is controlled by the slider displayed at the bottom of its window. A snapshot of what the program's window should look like is shown on the right.

- Change the range of the `JSlider` from 0-100 to 0-255.
- Change the wording of the `JLabel` that appears before the `JSlider` appropriately.
- Next, construct an `SImage` in the `sliderChanged` method specifying the current setting of the `JSlider` as the brightness value. This can be done using a construction of the form:

  ```
  new SImage( width, height, brightness )
  ```

  Make the `SImage`'s width be 256 and its height be 100.
- Associate the `SImage` with a local variable name. Then, use the `setIcon` method of the `JLabel` that appears in the center of your `AdjustBrightness` window to display the new `SImage` in your program's window.
- Compile and run your program. Test it by moving the slider back and forth.



Step 6: Thinking Inside the Box

We would like you to draw a frame around the gray images you create as shown on the right (the frame in this example surrounds a gray rectangle whose color closely matches the window background). This will require a four step process. First, you will have to create an monotone `SImage` as you did in step 4. Next, you will use a method provided with the `SImage` class to access a table containing all of the pixel values for the `SImage`. Then, you will write code to modify the values describing the edges of the image to make them black. Finally, you will make and display a new `SImage` using the modified table of pixel values.

This process involves using a new feature of the Java language called an *array*. Arrays in Java are objects that can be used to represent lists or tables of other values. The `SImage` class provides a method named `getPixelArray` that returns an array containing the integer values describing the brightnesses of all of the pixels in the `SImage`. For example, if `somePic` was a name associated with an `SImage`, you could use the assignment

```
pixels = somePic.getPixelArray();
```

to associate the name `pixels` with the collection of pixel brightness values. If you have an appropriate array of brightness values, it is also possible to create an image with these values using a construction of the form:

```
new SImage( pixels )
```

The notation used in Java and many other programming languages to work with arrays evolved from notations mathematicians use when working with matrices and vectors. For example, in a linear algebra textbook you are likely to find a definition that reads something like:

Definition 1.1 A rectangular collection of m·n values of the form

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \ldots & A_{1,n} \\ A_{2,1} & A_{2,2} & \ldots & A_{2,n} \\ \ldots & \ldots & \ldots & \ldots \\ A_{m,1} & A_{m,2} & \ldots & A_{m,n} \end{bmatrix}$$

is called an m by n matrix.

When working with a matrix, mathematicians use the name of the matrix, A, by itself to refer to the entire collection and use the name together with two subscript values, such as $A_{3,5}$, to refer to a particular element of the collection. Similarly, with Java arrays, we can use the name of an array, `pixels` for example, to refer to the entire collection, and the name together with subscript values to select a particular entry. In Java, however, the values that specify the position of the desired element are not actually written as subscripts. Instead, the values are placed in square brackets after the name. For example, we could write

```
pixels[3][5] = pixels[3][5] + 1;
```

to make the brightness value associated with the pixel at position 3,5 just a bit larger. Since they are not positioned as subscripts, we often refer to the values in square brackets as indices rather than subscripts.

Java's conventions for numbering the elements of an array are slightly different from those used with matrices. Java starts counting at 0, and, when dealing with images, the first index indicates the horizontal position within the image and the second index indicates the vertical position. That is, we might replace the tabular presentation of the matrix in Definition 1.1 with:

```
pixels = [ pixels[0][0]    pixels[1][0]    . . . pixels[m-1][0]
           pixels[0][1]    pixels[1][1]    . . . pixels[m-1][1]
             . . . . . .
           pixels[0][n-1]  pixels[1][n-1]  . . . pixels[m-1][n-1] ]
```

Finally, Java also uses the square brackets when describing the type of a matrix. We must describe types when we declare variables. For example, we use the word `int` to describe the type of `x` in the declaration

```
private int x;
```

The collection returned by the `getPixelArray` method is a table of `int` values rather than a single `int`. Therefore, to correctly declare a name like `pixels` that will be associated with such a table, we write

```
private int [][] pixels;
```

Therefore, the code:

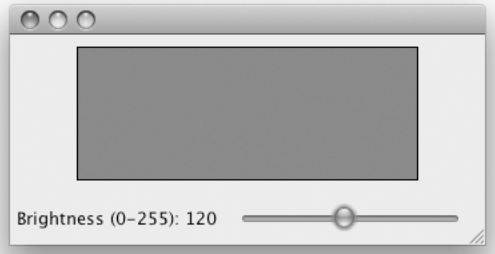
```
int [][] pixels = somePic.getPixelArray();

int y = 0;
while ( y < 100 ) {
       pixels[0][y] = 0;
       y = y + 1;
}
somePic = new SImage( pixels );
```

will take an existing `SImage` associated with the name `somePic`, and create a new image that is identical to the original except that it has a black line along its left edge. The first line simply associates an array containing the brightness values for the original image with the name pixels. The loop then sets each of the brightness values associated with the pixels with x coordinate 0 (i.e., the left edge of the image) to 0 (i.e., black). This does not actually modify the image since the array produced by `getPixelArray` is a copy of the brightness values. The last statement creates a new image with the modified brightness values and makes `somePic` refer to this new image.

- Edit your `sliderChanged` method to include code similar to the instructions shown above (replacing the name `somePic` with the name you used for your `SImage`).
- Compile and run this new class. It should draw a black line at the left edge of the rectangle produced when the slider is moved.

You can draw the other lines required to complete a rectangular frame around the image by writing four separate loops or just two loops. If you use just two loops, one will draw both horizontal lines and the other both vertical lines. You will discover one odd thing about the `y` coordinates used with an array of pixel values. The `x`-axis works as you would expect. The left edge of the image has `x` coordinate 0 and the right edge has `x` coordinate 255. On the other hand, the `y`-axis seems to be upside down. The `y` coordinate 0 is associated with the top of the image while the bottom of the image has `y` coordinate 99.

- Add the code required to draw the three lines needed to complete the box.
- Compile and test your program until it draws a nice box around your gray color sample.



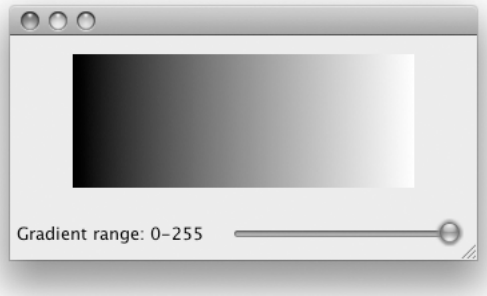
Step 7: A Little Housecleaning

One nice cosmetic change would be to move the brightness number that appears next to the image down next to the slider. that determines its value as shown on the right.

- Modify the way you set the text of your labels to place the current value of the slider just before the slider itself.
- Compile and test your program again.

At this point, the `sliderChanged` method of your `AdjustBrightness` class is getting a bit crowded. It contains quite a few lines of code to manipulate the pixel array. Before moving on, you should simplify this method's body by moving the code to make the pixel array into a private method within the `AdjustBrightness` class.

- Define a `makePixelArray` method that expects a single `int` (the setting of the slider) as a parameter and returns an "int [][]".
- Place the code you just wrote to create a gray rectangle surrounded by a black box in this new method.
- Modify the body of `sliderChanged` so that it invokes your `makePixelArray` method, constructs an `SImage` with the array the method returns, and displays the `SImage` as the icon of your `JLabel`.
- Compile and test the revised class.



Step 8: Making the Grade(ient)

Currently, the image produced by your `makePixelArray` method is pretty simple. It contains at most two shades of

gray, the black border and whatever shade is selected using the slider to fill the interior of the rectangle. To learn a little more about manipulating pixel values, let's make an image that contains every possible shade of gray. In particular, we want you to modify your `makePixelArray` method so that it will create a pixel array that describes a gradient like that shown in the window to the right.

The image drawn in this window contains all 256 possible shades of gray. The left edge of the rectangle is solid black. The right edge is white. Each of the lines between these two extremes is drawn with a color one shade lighter than the line to its left. (For now, we will just ignore the position of the slider. That is, initially your program should draw the same gradient no matter how the slider is set.)

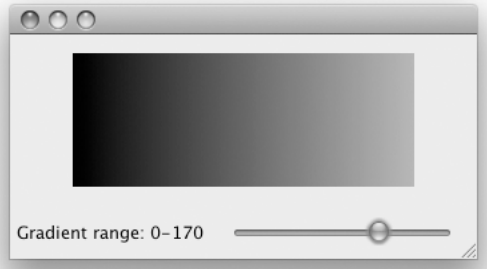
If you think about how we just described the gradient, you will realize that every pixel should have a brightness value equal to its `x` coordinate. The pixels on the left edge have `x` coordinate 0 and they are supposed to be black which corresponds to a brightness value of 0. The pixels on the right edge are supposed to be white. This is described by the brightness value 255 which happens to be the `x` coordinate of all of the pixels on the right edge (since earlier we told you to make the `SImage` 256 pixels wide). This means that you can set each pixel to the correct color by executing a statement like:

```
pixels[x][y] = x;
```

for all possible values of `x` and `y`.

To execute this statement for all possible `x` and `y` values you will use two while loops. This time, however, instead of coming one after another, one of the while loops should be nested inside of the other. One of the loops will start by setting a local variable named `x` to 0 and will then add 1 to `x` each time it executes until the value of `x` reaches the width of the `image`. The other loop will step through all values of the `y` coordinate from 0 to the height of the image. The inner loop, including the statement that sets y to 0, will sit within the outer loop and the statement that sets array elements equal to x will be placed within the inner loop.

These nested loops should replace the loops that draw the frame. Therefore, while you may want to keep the loops that draw the frame as examples while you write the loops that produce the gradient, once the code to draw the gradient is complete you should delete the loops that draw the frame.



- Write the nested loops that will draw the gradient.
- Compile and test your program until it draws the desired gradient.

Once you can draw a gradient that spans all the shades from black to white, we would like you to change the code you have written just a bit so that the gradient produced will go from black to the brightness value selected by the slider. In the figure on the right, for example, we have drawn a gradient that is black on the left edge but has the brightness value 170 on the right edge.

Drawing such a gradient is mainly a matter of scaling. In the sample window, the slider's knob is set two thirds of the way from the left. As a result, the brightness value of the right edge of the gradient would be 170 which is 2/3 of 255, the x coordinate of the right edge. In addition, we would want all of the other pixels in the image to have brightness values that were roughly equal to 2/3 their `x` coordinates. That is, the brightness of a pixel with `x` coordinate 90 should be 60 and the brightness of pixels with `x` coordinate 210 should be 140. This should be all that you need to know to modify the code you just wrote, but you will have to be a little careful writing the code because Java does arithmetic a little oddly.

Suppose, as suggested above, that the slider's value was set to 170 and you tried to compute the value of the Java expression

```
someSlider.getValue() / 255
```

According to the normal rules for division, this expression should produce the value 2/3 or .6666667. Unfortunately, in Java, it produces the value 0. This is because both of the values involved in the operation are integer values, and Java therefore believes it has to produce an integer answer. As a result, if you try to use the expression

```
( someSlider.getValue() / 255 ) * x
```

in your program, its value will almost always be 0 (the one exception is when the slider's value is 255).

Luckily, there is a simple way around this. Suppose that you instead evaluate the expression

```
( x * someSlider.getValue() ) / 255
```

This time, Java will first multiply the value of the slider and the value of `x`. Suppose that `x` is 100 and the slider's value is 150. The result will then be 15000. The value associated with 15000/255 following the normal rules of arithmetic is 58.823. When we evaluate this expression in Java, it still will give us an integer. Therefore the value produced will be 58. This isn't quite the right answer, but it is much closer than 0. Therefore, you should use an expression of this second form to create the desired gradient.

The lesson is to be very careful about the order in which you write operations when using division with integers in Java.

- Modify your code to scale the brightness values of the gradient based on the setting of the slider.
- Modify the code that sets the text displayed to the left of the `JSlider` so that it describes the range of the gradient currently displayed.
- Compile and test your program

The `SImage` class provides methods named `getWidth` and `getHeight`. You should used these methods to determine how many times the loops in your makePixelArray methods are executed (instead of coding numbers like 256 and 100 into the loop headers).

- Change your code to use `getWidth` and `getHeight` to determine how frequently the loops that create your gradient are executed.
- Compile and test your program again.

Step 9: The Meek Shall Inherit

While it may be fun to draw gradients, we suspect you are wondering whether we are ever going to get around to implementing the image processing operations we promised on the first page of the handout. Luckily, you are are actually very close. The next step in the process is to change the program in an interesting way so that it can get access to the images you want to modify.

Throughout the semester, most of the classes you have written have included the phrase "`extends GUIManager`" in their headers. Those that did not extend `GUIManager`, did not extend anything. Now, we will conduct a little experiment to prove that it is possible to extend things other than `GUIManagers`.

- Modify the header of your AdjustBrightness class so that it says "`extends ImageViewer`" instead of "`extends GUIManager`". (In case you forgot, `ImageViewer` is the name we told you to give to the other small program we asked you to write at the beginning of this lab.)
- Run your program. Resize the window a bit if what pops up isn't recognizable at first.

The result should resemble some some sort of deformed hybrid of the `ImageViewer` and `AdjustBrightness` programs. It isn't quite healthY, but the basic features of both programs should be evident

on the screen. The buttons that belong to the `ImageViewer` class should appear at the top of the window while the slider created by `AdjustBrightness` appears at the bottom.

Adding the phrase "`extends ImageViewer`" to the class header of `AdjustBrightness` tells Java that you want to use `ImageViewer` as a starting point for constructing the `AdjustBrightness` program. That is, Java will now perform all the steps it would have performed to create an `ImageViewer` before it performs the steps required to create an `AdjustBrightness` object. We say that `AdjustBrightness` *inherits* all of the features of an `ImageViewer`. In describing their relationship, we say that ImageViewer is a *superclass* of `AdjustBrightness` and that `AdjustBrightness` is a *subclass* of `ImageViewer`.

The reason the program does not work very well at the moment is that `AdjustBrightness` currently duplicates some of the initialization steps already performed by `ImageViewer`. In the constructors of both classes you invoke `createWindow` (probably with different sizes), set the layout to be a `BorderLayout` and place a `JLabel` to display images in the middle of the window. Now that `AdjustBrightness` extends `ImageViewer`, it does not need to perform all these steps itself. They will be taken care of for it by the constructor of the `ImageViewer` class. So, to make the program work a bit better:

- Remove the invocations of `createWindow` and `setLayout` from the `AdjustBrightness` constructor.
- Remove the invocation of `contentPane.add` to place a JLabel in the `CENTER` of the `contentPane` from the AdjustBrightness constructor (but do NOT remove the instance variable that refers to the `JLabel` or the construction of the JLabel quite yet).
- Compile and run the program. Try all of the controls (buttons and sliders) to see what works.

At this point, everything except the slider should work as expected. Even the slider works in the sense that the label displaying the slider's value changes as you adjust the slider. The gradient you create when the slider is moved, however, never appears on the screen.

First, it is important to understand why so many features are working. Your `AdjustBrightness` class has no buttons or `buttonClicked` method. When you run it, however, it displays buttons and responds when you click them. This is because `AdjustBrightness` inherits both the code in the `ImageViewer` constructor that creates the buttons and the definition of `buttonClicked` that specifies what to do when the buttons are clicked.

The reason the gradient you create when the slider is moved does not appear is that the `JLabel` for which the gradient is used as an icon is not the JLabel displayed in the window. `ImageViewer` has its own `JLabel` that is separate from the JLabel created in `AdjustBrightness`, and only the `ImageViewer`'s `JLabel` is visible in the window.

If you try to change your `AdjustBrightness` class to set the icon of the JLabel created in `ImageViewer`, you will discover that you cannot refer to it within the code placed in the definition of `AdjustBrightness`. This is because the instance variable that refers to the `JLabel` is declared private to `ImageViewer`. As a result, AdjustBrightness inherits the `JLabel`, but it does not inherit the name that refers to it. We could fix this by changing the variable's declaration to make it `public`, but there is a better way.

- Define a new "`public void`" method in `ImageViewer` named `updateDisplay`. This method should:
  - Expect an SImage as a parameter.
  - Use `setIcon` to place this image in the JLabel placed in the middle of the `ImageViewer`'s window.

Because this new method is public, it can be used in both the `ImageViewer` and the `AdjustBrightness` classes.

- Modify the code to handle both the "Load Image" and the "Take Snapshot" buttons so that they invoke updateDisplay rather than setting the icon of the JLabel directly.
- Modify the code in the `sliderChanged` method of the `AdjustBrightness` class so that it invokes `updateDisplay` to place the gradients it creates on the screen.
- Remove the `JLabel` that was declared in the original `AdjustBrightness` class.
- Compile and test your revisions by running AdjustBrightness until everything works as expected.

Step 10: A Little More Housekeeping

A little later in this lab, we will have you write another class that is very similar to the current version of the `AdjustBrightness` class. Therefore, at this point we would like you to save a copy of your `AdjustBrightness` class under a different name. To do this:

- Click on the "New Class" button and create a class named `Quantizer`.
- Copy and paste the entire code of your `AdjustBrightness` class so that it replaces the template Java provides for your new `Quantizer` class.
- Within the window for the new `Quantizer` class, edit the text by replacing the name "`AdjustBrightness`" that appears on the first line of the class and of its constructor with the name "`Quantizer`".
- Save the new class.

Now, close the `Quantizer` class window and go back to working with the `AdjustBrightness` class.

Step 11: Adjusting Image Brightness

The `sliderChanged` method in `AdjustBrightness` displays an image constructed using the pixel array created by your `makePixelArray` method. We want to change this method so that the array it produces is derived from an `SImage` loaded from an image file or taken by the computer's built-in camera. We would like the image displayed to be obtained by darkening the pixels of the original image by an amount specified by your slider.

Your current `makePixelArray` method starts by constructing a new `Simage` and associating that image's pixel array with a local variable.

- Delete the first of these two steps from the method.
- Declare what had been the local variable name for the SImage you created as a second formal parameter for the method. That is, instead of creating its own SImage, the method will be passed an `SImage` as a parameter.
- Change the name of the method to `adjustPixels`.

Instead of creating a gradient, we want you to scale the brightness value of each pixel of the image passed to your `adjustPixels` method just as you scaled the brightness values in your gradient based on the setting of the slider. That is, if a pixel's original brightness is "`b`" then we want to replace it with

```
( b * someSlider.getValue() ) / 255
```

- Using this tip, change the loop in your `adjustPixels` method so that it scales the brightness values in the pixel array based on the setting of the slider.

Now, we have to get the image loaded by the `ImageViewer` class into your new `adjustPixels` method. This image should be associated with a private instance variable in the `ImageViewer` class. Just as we defined a public `updateDisplay` method to give `AdjustBrightness` a way to access the JLabel declared within the `ImageViewer`, we can define a method to give us access to the `SImage`.

- Define a new public method named `getImage` in the `ImageViewer` class. This method:
  - Will expect no parameters.
  - Will simply return the last SImage retrieved when either the "Load Image" or "Take SnapShot" button was pressed.
- Now, modify the `sliderChanged` method in the `AdjustBrightness` class so that it invokes this method and passes the result as the second parameter to the `adjustPixels` method.
- Compile and test your work by loading an image with the `ImageViewer` class and then adjusting the slider to see if it darkens (and brightens) the image. Only work with grayscale images from our collection at this point.

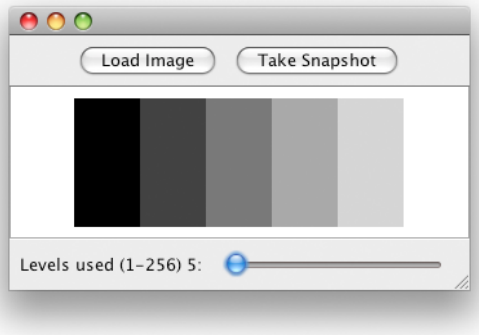
Step 12: Quantization

Early in the course, we emphasized that a defining feature of digital communication was that it depended on a finite set of discrete symbols to encode information. In the domain of images, this principle of digital communications manifests itself in the fact that we have been using 256 levels of brightness to describe images. Physically, brightness is a continuous phenomenon. For any two levels of brightness there are additional levels of brightness between them. In the images we have been working with, however, there are no brightness levels between 254 and 255 or between 40 and 41. Brightness has been quantized.

This means that a digital representation of an image cannot always be exact. If the actual brightness of a pixel in a scene falls between 40 and 41, we will have to approximate this with either the value 40 or 41. Of course, we can make our images more accurate by using more levels. Instead of distinguishing 256 levels of brightness we could use 1000 levels or 10000 levels. The more levels we use, however, the more bits will be required when the image is encoded in binary. There is a tradeoff, therefore, between the accuracy of a digital image and the number of bits required to store it or transmit it.

To explore this tradeoff, we would like you to write another class that will show how images would look if they were displayed using a smaller set of brightness levels. As you might have guessed, that is why we had you save a copy of your `AdjustBrightness` class under the name `Quantizer` a few steps ago.

The code we had you save as the `Quantizer` class creates a gradient of shades of gray such that the brightest shade displayed is determined by the setting of the slider in its window. If you forget how this works, create an instance of the class to remind yourself.



We want you to change the code in this class so that it display a different sort of gradient. The figure on the right illustrates what we have in mind. The slider is set at 5. The gradient shown, therefore includes 5 levels of brightness that might be used to represent images if we used only 5 levels of brightness instead of the usual 256. The brightness levels shown are determined by dividing the usual range of 256 brightness levels into 5 roughly equal subranges. While it might be best to use the brightness level at the middle of each range, for the sake of simplicity, we have instead used the smallest brightness value in each range. The five shades shown are therefore brightness 0, brightness 51, brightness 102, brightness 153, and brightness 204.

Suppose that we want to use `N` different shades of brightness. Then `256/N` will tell us how wide each band of our gradient should be. Now, suppose that we want to decide what color to use for the line of pixels at position `x`. If we divide `x` by `256/N`, Java will give us the integer portion of the result of this division. For the first `256/N` values of `x` this will be 0, for the next `256/N` values it will be 1, and so on. If we multiply this number by `256/N` the result will be the level of gray we should actually use for the x coordinate. That is, you should:

- Define a local variable with a name like `widthOfBrightnessBands` and initialize its value to be:

```
256/someSlider.getValue()
```

- For each pixels, set its brightness to its `x` coordinate divided by `widthOfBrightnessBands` and then multiplied by `widthOfBrightnessBands`.
- Change the range of the program's slider from 0-255 to 1-256 (since trying to draw an image using 0 colors is not a good idea) and change its label appropriately.
- Compile and test your program until it draws nice, segmented gradients. You will probably have to set the slider to a very small value to see the segments.

Once this works, we want you to transform your `Quantizer` class into a class that can be used to quantize the pixels of a grayscale image rather than to just draw a gradient. The process will be nearly identical to what you did with the `AdjustBrightness` class.

- Change the `makePixelArray` method into an `adjustPixels` method by making it accept an `SImage` as a parameter instead of creating its own `SImage`.
- Change the code in the loop within `adjustPixels` so that it quantizes values obtained from the pixel array instead of the `x` coordinates of the pixels.
- Modify the `sliderChanged` method so that it passes to `adjustPixels` the `SImage` obtained by invoking the `getImage` method defined in the `ImageViewer` class.

Compile and test this new class (using only grayscale images).

Step 13: A Horse of a Different Color

Let's add a little color to this program.

Each pixel of a color image is described using three values that describe the brightness (or intensity) of each of the primary colors that make up the actual color of that pixel. There is one value that specifies how much red is in the pixel, one value that specifies how much green, and one value that specifies how much blue. Each of these values falls between 0 and 255.

The `SImage` class lets you access the values that describe the intensity of each primary color in all of the pixels of an image as a single pixel array. To do this, you simply include a parameter that specifies which color you are interested in when you invoke the `getPixelArray` method. For example, to get the brightness values that describe how much red is used in each pixel of an `SImage` you could say:

```
someSImage.getPixelArray( SImage.RED )
```

Similarly, you can get pixel arrays describing the greenness or blueness of the pixels with invocations of the form

```
someSImage.getPixelArray( SImage.GREEN )
```

and

```
someSImage.getPixelArray( SImage.BLUE )
```

In reality, `SImage.RED`, `SImage.GREEN`, and `SImage.BLUE` are just names for int values used to indicate which color layer is desired.

In addition, if you have three pixel arrays that describe the rednesses, greennesses and bluenesses of the pixels of an image, you can make a new `SImage` out of these three arrays by saying:

```
new SImage( rednessArray, greennessArray, bluenessArray )
```

As a result, you can now fairly easily modify your `AdjustBrightness` and `Quantizer` classes so that they will work correctly on color images.

Currently, in your sliderChanged method you create an updated SImage by saying something like:

```
new SImage( adjustPixels( sliderValue, getImage() ) )
```

To work with colored images, you will redefine adjustPixels so that it takes both an image and an int specifying which color layer of the image to work on. That is, you will change the header of the method from something like

```
private int [][] adjustPixels( int level, SImage original ) {
```

to something like

```
private int [][] adjustPixels( int level, SImage original, int layer ) {
```

Then, at the start of the method, you will extract the array of brightness values to work with by saying

```
original.getPixelArray( layer )
```

instead of

```
original.getPixelArray()
```

Once these changes are made, you can replace the single invocation of `adjustPixels` in you `sliderChanged` method with three invocations whose results are used together to create the desired image. The revised construction will look something like

```
new SImage( adjustPixels( sliderValue, getImage(), SImage.RED ),
            adjustPixels( sliderValue, getImage(), SImage.GREEN ),
            adjustPixels( sliderValue, getImage(), SImage.BLUE ),
          )
```

With this in mind, in both the AdjustBrightness and Quantizer class you should:

- Modify `adjustPixels` to accept an additional int parameter which will have one of the values `SImage.RED`, `SImage.GREEN`, or `SImage.BLUE` as its actual value.
- Use this new parameter value to determine which pixel array to extract from the image passed to the method.
- Apply the `adjustPixels` method three times in your sliderChanged method using the three values `SImage.RED`, `SImage.GREEN`, and `SImage.BLUE` as the final parameter.
- Use the three arrays returned by these three invocation of `adjustPixels` as parameters to the construction of a new `SImage`.
- Display the image produced using the `updateDisplay` method of the `ImageViewer`.
- Compile and test the modified program.

Step 14: Want More Fun?

Now that you know how to use the tools, there are many ways you could extend this program. A simple improvement would be to let the user adjust the levels of the three primary colors in an image independently. That is, you could revise your `AdjustBrightness` class so that its interface looked like the window shown on the next page.

Another option that might be appealing is to have some fun with the built-in iSight camera. Snapshots are nice, but wouldn't you rather have video?

There are two things you need to know to construct a program that previews the images coming from the camera in video mode. First, you have to tell the camera to stay on rather than turning itself on and off

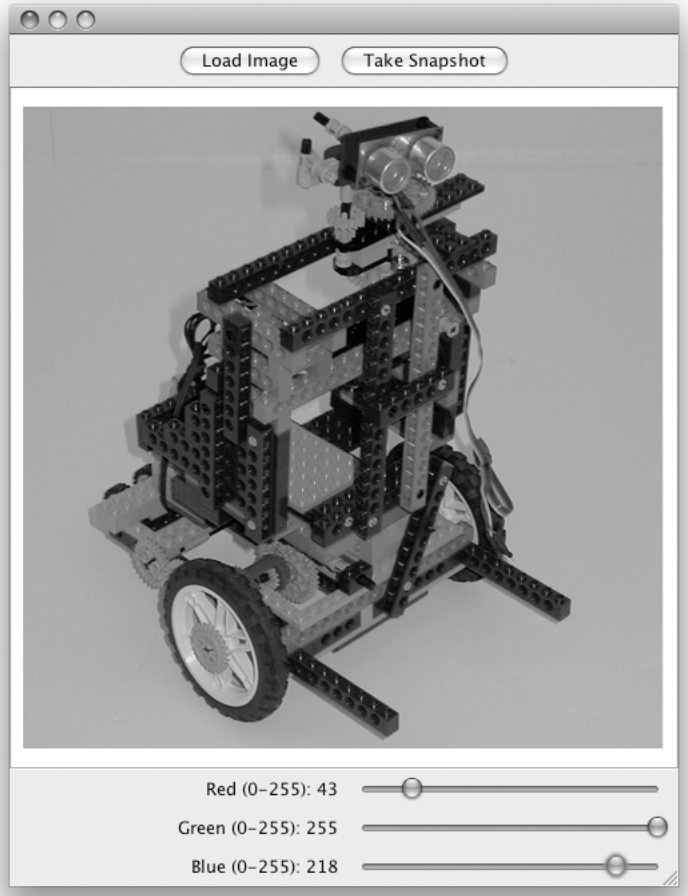
every time you invoke `getImage`. You do this by invoking a method of the `Camera` class called `activate`. This method requires no parameters.

The other thing you need to do is to get your program to perform a `getImage` at regular intervals and display the image on the screen. To enable you to write programs that perform actions at regular intervals, the Squint library includes a class named `PaceMaker`.

The `PaceMaker` constructor expects two parameters. The first parameter is a number specifying how much time should elapse between the times at which the repeated action is performed. The second parameter is the name of a `GUIManager` that will perform the necessary action. Thus, to create a `PaceMaker` that would remind you to display a new image every 1/10th of a second, you might use the construction

```
new PaceMaker( 0.1, this )
```

To use a `PaceMaker`, you define a method named tick within your `GUIManager`. The `PaceMaker` will arrange to execute the code you place within your `tick` method at the desired frequency. The `tick` method should not expect any parameter.



Finally, you might want to experiment with inheritance a bit more. The `Quantizer` class is very similar to the `AdjustBrightness` class. In particular, the `sliderChanged` methods in the two classes are probably identical. Could you eliminate some of the duplicated code by defining Quantizer as a class that "`extends AdjustBrightness`"? Quite frankly, the result you obtain may be a bit ugly, but you will learn more about inheritance in the process.

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 `FloydLab7`. If you are working with a partner, but both of your names on the folder.

- 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

This labs will be graded on the following scale:

++	An absolutely fantastic submission of the sort that will only come along a few times during the semester.
+	A submission that exceeds our standard expectation for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.
✓+	A submission that satisfies all the requirements for the assignment --- a job well done.
✓	A submission that meets the requirements for the assignment, possibly with a few small problems.
✓-	A submission that has problems serious enough to fall short of the requirements for the assignment.
-	A submission that is significantly incomplete, but nonetheless shows some effort and understanding.
--	A submission that shows little effort and does not represent passing work.

Completeness / Correctness

- All three classes display desired interfaces
- ImageViewer works as an independent class
- File dialog displayed when Load button pressed
- Load Image works correctly
- Take Snapshot works correctly
- `AdjustBrightness` reduces brightness of images
- `Quantizer` correctly requantizes images

Style

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

Appendix: Summary of New Library Features Used in this Lab

Creating Sliders

```
new JSlider( minimumValue, maximumValue, initialValue )
```

- The parameters determine the values that will be associated with the slider as its knob is moved by the user.

Accessing Slider Values

```
someSlider.getValue()
```

- Get the value associated with the current position of the slider's knob.

```
someSlider.getMinimum()
```

- Get the minimum value associated with the slider.

```
someSlider.getMaximum()
```

- Get the maximum value associated with the slider.

Event Handling for Changes in a Slider's Position

```
public void sliderChanged() { ... }
public void sliderChanged( JSlider which ) { ... }
```

- Instruction placed within the body of a `sliderChanged` method will be executed whenever the knob of a slider in a `GUIManager`'s content pane is moved.

Accessing or Creating an Image:

```
new SImage( someString )
```

- The String passes as an argument can be a file name or the URL of an image on the web.

```
new SImage( width, height, brightness )
```

- Creates a monotone grayscale image with the width, height, and brightness specified.

```
new SImage( somePixelArray )
```

- The image produced will be a grayscale image. The values in the pixel array are treated as the brightness values for the individual pixels of the image.

```
new SImage( somePixelArray, anotherPixelArray, oneMorePixelArray )
```

- The image produced will be a color image. The first array specifies the redness of the pixels, the second specifies the greenness, and the third specifies the blueness.

Accessing the dimensions of an SImage

```
someSImage.getWidth()
someSImage.getHeight()
```

- These methods return the requested size of the image measured in pixels.

Accessing a Pixel Array

```
someSImage.getPixelArray()
```

- Returns a pixel array describing the brightness of the pixels of the image.

```
someSImage.getPixelArray( someColor )
```

- Returns a pixel array describing the amount of red, green, or blue in each pixel of an image depending on the value of its parameter. The valid values of the parameter are 0 (for the redness values), 1 (for greenness), and 2 (for blueness). The names `SImage.RED`, `SImage.GREEN`, and `SImage.BLUE` are associated with the values 0, 1, and 2 respectively so that one can say things like

```
someSImage.getPixMap( SImage.RED )
```

Displaying images in JLabels

```
someJLabel.setIcon( someSImage );
```

- The specified image will appear within the JLabel.

Creating a Camera

```
new Camera()
```

- Provides a connection to the computer's built-in camera.

Accessing an Image from a Camera:

```
someCamera.getImage()
```

- Returns an SImage from the camera.

Using a Camera in Video Mode:

```
someCamera.activate();
```

- Tells the camera to remain active between invocations of getImage.

Creating a PaceMaker

```
new PaceMaker( delay-between-activation, this )
```

- The first parameter determine the time that should elapse between invocations of the second parameter's tick method.

Event Handling for a PaceMaker

```
public void tick() { ... }
public void tick( PaceMaker which ) { ... }
```

- Instruction placed within the body of a `tick` method will be executed at the frequency specified in the PaceMaker construction.