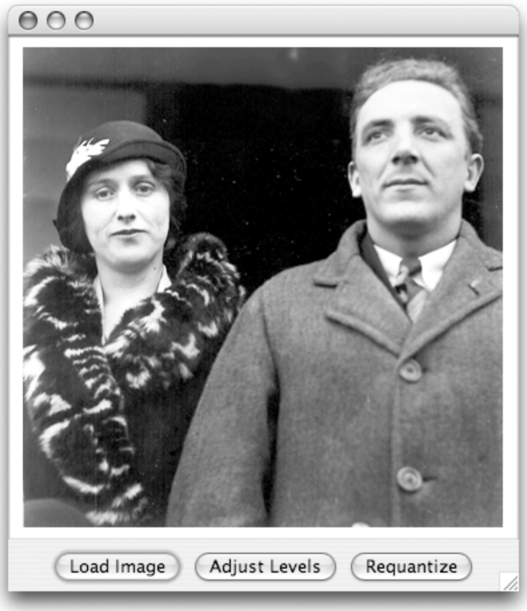
Lab 7

Digital Image Processing

Due April 4/5, 11PM

In this week's lab, you will construct a program that implements two digital image processing tasks:

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

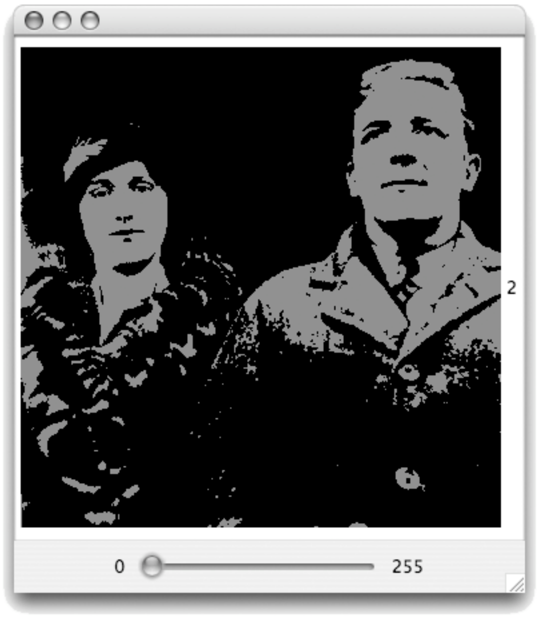


The interface your completed program should display is shown on the right. The program's window displays three control buttons. The first loads a new image for processing. When it is pressed, a dialog box opens to allow the user to select an image file.

Once an image has been chosen, the user can click either of the other buttons to perform an operation on the most recently loaded image.

When one of the processing buttons is pressed, a new window opens to display the processed version of the image. For example, if you press the "Requantize" button, a window like the one shown below should appear. This window displays a version of the image displayed using just two shades, black and gray (the role of the slider in determining how many shades are used will be discussed below).

When your program is complete, it will consist of one class that displays the window with the control buttons and one additional class for each of the image processing operations. All these classes will extend GUIManager.



This lab is intended to function as a tutorial. The handout provides detailed instructions on how to construct the program we have described above, introduces the details of several new library classes and methods, and explores one new feature of the Java language itself. We describe a new class named `SImage` that supports the manipulation of images. We explain how to manipulate tables of values describing the individual pixels in an image using a feature of Java called arrays. In addition, we introduce several new features of the Java Swing library. We will show you how to use `JLabels` to display images. We will introduce a new Swing class named `JFileChooser`. When you press the "Load Image" button in your program's interface, we want your program to provide its user with the ability to navigate around the computer's file system to find image files. The `JFileChooser` class makes this easy to do. Finally, the image processing operations we want you to implement will be controlled by GUI components known as sliders, so we will also introduce the `JSlider` class in this lab.

Step 0: Gettting 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 possible for some of the programs you will write during lab to find these images, they must be stored within your "Documents" folder for your account (i.e., in the same folder that contains the folder BlueJ creates for your project). Therefore, to start your work:

- First, start 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 "Lab Schedule and Handouts" 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` folder for this lab.
- Point at the link. Hold down the control key and depress the mouse button to make a menu appear.
- While still holding the control key and the mouse button, depress the option key. The item that had said "Download Linked File" should now read "Download Linked File As ..." Select this item.
- Using the dialog box that appears, navigate to your "Documents" folder and save the `AllImages.zip` file in that folder. (You may want to 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 throw away the .zip file.

Step 1: Accessing and Displaying an Image

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

- First, 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.
- Next, add code to display two GUI components: a `JLabel` that displays a short message ("Hi" will do) and a `JButton` with the label "Load Image". Associate instance variables with both the `JLabel` and the `JButton`. Even though there is only one button now, place it in a `JPanel` since you will eventually have three buttons and want them to all be displayed together. Add the label to the content pane before the `JPanel` that holds 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 that will actually load and display an image. Ultimately, the person running the program should be able to select any image file, but for now let's write simple code that will always load the image stored in the "`AllImages/MMM.gif`" file.

There are two steps you need to perform in `buttonClicked` 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 extend this class definition to support the "Adjust Levels" and other buttons, you will need to remember the image data between button clicks.

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 a picture of one of your instructor's grandparents. Can you see the family resemblance?

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 button 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 by performing all three of the following steps before you attempt to run your program again:
  1. Add the instruction

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

     to the beginning of your program's constructor.
  2. Place the `JLabel` in the center of the window by replacing the command to add it to the content pane with a command of the form:

     ```
         contentPane.add(  ...   , BorderLayout.CENTER );
     ```
  3. Change the command that added the `JPanel` containing the "Load Image" button so that the panel is placed at the bottom of the window (`BorderLayout.SOUTH`).

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 can 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 that you can display the `MMM.gif` file, 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. 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.

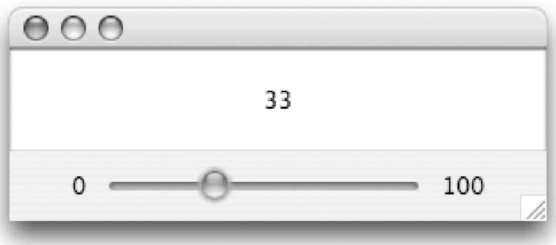
When you ran the program, you might have noticed one issue you must address if you want your program to be robust. The dialog box a file chooser displays includes a "Cancel" button. If the user presses cancel, then the `getSelectedFile` method will not be able to return a description of the file. To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

Run the program again after making this change to ensure that it handles the cancel button correctly.

Step 3: Slippery Sliders

In the image on the bottom of the first page, we showed a window 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 right. 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.

To begin, use the "New Class.." button to create a new class that extends `GUIManager`. While we want you to write this as a standalone program, by the end of the lab we will incorporate its code as a subcomponent of the image processing program described at the beginning of this handout. This subcomponent will handle the "Adjust Levels" button. Accordingly, you should name your new class `AdjustLevels`. You can delete all of the event handling methods in the template that BlueJ provides for this class. Add an import statement for `java.awt.*`.

As usual, start by writing a constructor that displays the basic elements of the program's interface without actually making them work. This program displays four components. The number in the middle of the screen that displays the current value of the slider is just a `JLabel` (set its initial contents to be "50"). The numbers "0" and "100" that appear at either end of the slider are not part of the `JSlider` itself. They are independent `JLabels`. The slider that appears between the labels "0" and "100" is a member of the class named `JSlider`. You should associate instance variable names with the `JLabel` that displays the current setting of the slider ("33" in the illustration) and the `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 should use a construction of the form

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

to create your program's slider. Associate an instance variable with the slider since later on you will have to apply methods to the slider to determine its current setting.

As mentioned above, we will adapt the code you are writing now for other purposes later in this lab. In particular, you will have to change the range of values associated with the slider in later versions of this program. As a result, when you create the `JLabels` that appear before and after the slider, you should not type something like

```
new JLabel( “100” )
```

since you would have to remember to replace the `String` “100” later in the lab. A better approach is to take advantage of the fact that the `JSlider` class provides methods named `getMinimum` and `getMaximum` that can be used to determine the range of a slider. Therefore, if `someSlider` was the name of your `JSlider`, your `JLabel` construction should look like:

```
new JLabel( “” + someSlider.getMaximum() )
```

The only tricky part about creating this program’s interface is getting the layout desired. As mentioned above, you will eventually incorporate this program as a part of your larger image processing program. When used in that context, we will want this window’s content pane to be managed by a border layout manager. Rather than wait until later, we would like you to start using a border layout manager now!

First, set the layout manager for the program’s `contentPane` to be a new `BorderLayout`. Then, place the `JLabel` used to display the slider’s current value at `BorderLayout.CENTER` and 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 another `JPanel` to hold the `JSlider` and the `JLabels` that display the range of the `JSlider` and then place this additional `JPanel` in the `SOUTH` region of the content pane. Finally, add the value `SwingConstants.CENTER` as a second parameter in the construction that creates the `JLabel` in the center of the window.

- Run your program to verify that you have created and correctly displayed all the desired components.

Now, we want 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();
```

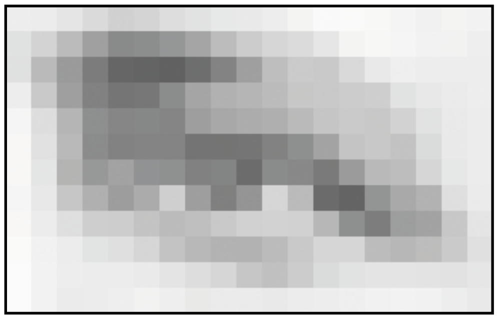
could be used to associate the updated value of a slider with an integer variable. Using this method, write a definition of `sliderChanged` that will update the number displayed by your program each time the slider’s knob is moved. You will need to concatenate the `int` value with `“”` when you use it as a parameter of the `setText` method just as we suggested you do in the `JLabel` construction shown above.

Step 4: Shades of Gray

When an image is represented within a computer, the colors of the parts of the image must 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. A black pixel’s shade is 0. 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 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:

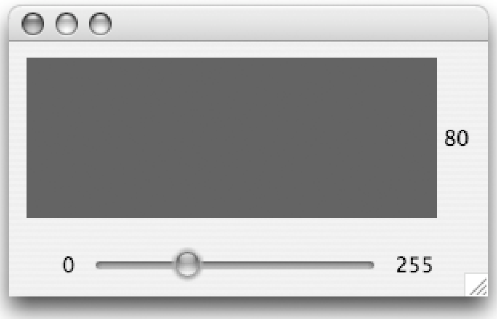


Within the computer, each of these gray squares is represented by a single number describing its brightness. The table below, shows all of the numeric values that would be used to describe the image of the eye.

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, we saw that we could construct 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 to do 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 the process of creating such an image, we would like you to modify your `AdjustLevels` program so that it creates and displays a monotone image whose brightness is controlled by the slider dis-



played at the bottom of its window. A snapshot of what the program's window might look like is shown on the left.

First, change the range associated with the `JSlider` from 0-100 to 0-255.

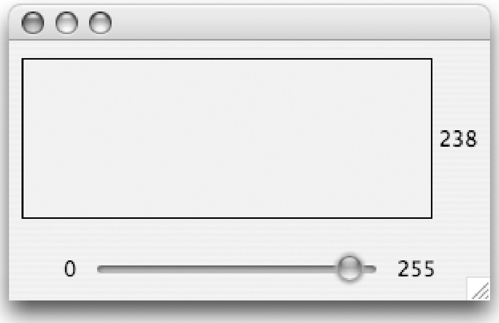
Next, in the `sliderChanged` method, construct an `SImage` specifying the current setting of the `JSlider` as the brightness value. This can be done using a construction of the form

```
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 `AdjustLevels` 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 5: Thinking Inside the Box



We would like you to draw a frame around the gray images you create as shown on the right (with the frame actually surrounding 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 brightness of the pixels at 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 or objects. 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.

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 $m \cdot n$ values of the form

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

is called an m by n matrix.

When working with such a matrix, we use the name of the matrix, A, by itself to refer to the entire collection and we use the name together with two subscript value, 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 itself, `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 of the array. 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 tiny bit larger. Since they are not actually 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:

$$\texttt{pixels} = \begin{bmatrix} \texttt{pixels[0][0]} & \texttt{pixels[1][0]} & \ldots & \texttt{pixels[m-1][0]} \\ \texttt{pixels[0][1]} & \texttt{pixels[1][1]} & \ldots & \texttt{pixels[m-1][1]} \\ \ldots & \ldots & \ldots & \ldots \\ \texttt{pixels[0][n-1]} & \texttt{pixels[1][n-1]} & \ldots & \texttt{pixels[m-1][n-1]} \end{bmatrix}$$

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 < somePic.getHeight() ) {
       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.

- Add this code to the instructions of your `sliderChanged` method.

- 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.

Now, add additional code to draw the other lines required to complete a rectangular frame around the image. You can do this 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.

Compile and test your program until it draws a nice box around your gray color sample.

Step 6: A Little Housecleaning

It would be nice if the gray rectangle and the black box appeared as soon as you ran the program. There is a simple way to do this. Add an invocation of the form:

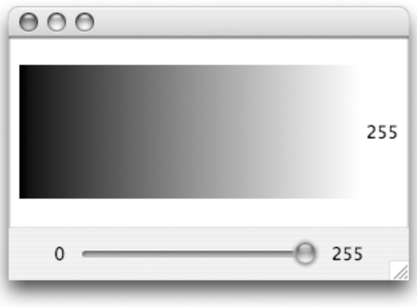
```
    this.sliderChanged();
```

as the last line in the `AdjustLevels` constructor. This tells Java to execute the code in the `slider-Changed` method (even though the slider hasn't actually been moved). Make this change and then compile and test again.

At this point, the `sliderChanged` method of your `AdjustLevels` 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 modify the pixel array into a private method within the `Adjus-tLevels` class. The method should be named `makePixelArray`. It should take an `int` value which will be equal to the setting of the slider as a parameter and return a pixel array describing an appropriate gray rectangle with a black border. Most of the code that had been in `sliderChanged` will be moved to this new method. Within the body of `sliderChanged`, you should invoke this method, construct an `SImage` with the array the method returns, and then display the `SImage` as the icon of your `JLabel`.

Make these changes and then compile and test the class.

Step 7: 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 cleverly 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:

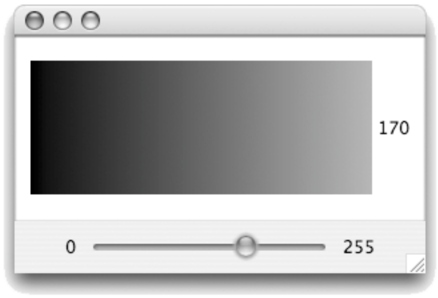
```
pixels[x][y] = x;
```

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 (probably 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 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. 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. Suppose that the slider's knob was set two thirds of the way from the left. Then the brightness value of the right edge of the gradient would be 170 which is 1/3 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, as long as you use an expression of this second form, you should be able 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.

Step 8: 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 `AdjustLevels` class. Before that, however, we want you to make some additional changes to `AdjustLevels`. Therefore, at this point we would like you to save a copy of your `AdjustLevels` class under a different name. To do this:

- Click on the "New Class" button and create a class that extends `GUIManager` named `Quantizer`.
- Copy and paste the entire code of your `AdjustLevels` 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 "`AdjustLevels`" 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 `AdjustLevels` class.

Step 9: Adjusting Image Levels

While it may be fun to have a program that can display gradients, we suspect you are beginning to wonder when we are going to get around to implementing the image processing operations we promised on the first page of the handout. Luckily, you are are now at the point where we can easily use the `AdjustLevels` class you have developed to adjust the brightness levels of an actual image. We will work with grayscale images first. Later, we will adapt the program to work with color images.

- First, open the window containing your `ImageViewer` class from the first two steps in this labs so that we can modify it to use the `AdjustLevels` class.
- Add a new button with the label "Adjust Levels" to the interface of your `ImageViewer`. Initially, this button should be disabled.
- Add an if statement in the `buttonClicked` method so that it only loads a new image when the "Load Image" button is clicked. Also add code to enable the "Adjust Levels" button when an image is loaded.
- Add an else to the if statement in `buttonClicked` so that the program creates a new `AdjustLevels` window when the "Adjust Levels" button is pressed.
- Compile and run the program to make sure that these changes work.

When you press "Adjust Levels", the window that appears will display a gradient. We would like it to instead display a modified copy of the image that was loaded within the `ImageViewer`. To do this, you will have to make a few more changes.

- Add an `SImage` instance variable to the `AdjustLevels` class.

- Change the `AdjustLevels` constructor to expect an `SImage` parameter and set the `SImage` instance variable equal to this parameter within the constructor body.
- Pass the image loaded by the `ImageViewer` class to the `AdjustLevels` constructor when the "Adjust Levels" button is pressed.

We are almost there. The `sliderChanged` method in `AdjustLevels` 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 the `SImage` passed to the `AdjustLevels` constructor.

- Your `makePixelArray` method starts by constructing a new `Simage` and associating that image's pixel array with a local variable. Delete these two line from the method and instead declare what had been the local variable name for the pixel array as a second formal parameter for the method. That is, instead of creating its own pixel array, the method will be passed a pixel array as a parameter.
- You probably used the expressions `someImage.getWidth()` and `someImage.getHeight()` within the conditions of the loops in your method. These expressions refer to the `SImage` you are no longer creating! To avoid this, add two integer parameters named `width` and `height` to your `makePixelArray` method and use these in place of `getHeight` and `getWidth` in your loop conditions.
- Change the name of the method to `adjustPixelArray`.

The `SImage` class provides methods named `getPixelArray, getWidth` and `getHeight`. You should use these methods to determine the three additional parameters passed from your `sliderChanged` method to your `adjustPixelArray` method..

- Using the `getPixelArray` method, modify your `sliderChanged` method so that it passes the pixel array of the `SImage` that was passed to the `AdjustLevels` constructor as a parameter to the `adjustPixelArray` method.
- Using `getWidth` and `getHeight`, modify your `sliderChanged` method so that it passes the width and height of the `SImage` that was passed to the `AdjustLevels` constructor as the third and fourth parameters to the `adjustPixelArray` method.

The last step is the fun one. We want you to scale the brightness value of each pixel of the image passed to `AdjustLevels` just as you scaled the brightness values in your gradient based on the setting of the slider. That is, if a pixel's brightness is "`b`" then we want to replace it with

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

- Using this tip, change the loop in your `adjustPixelArray` method so that it scales the brightness values in the pixel array based on the setting of the slider.
- Compile and test your work by loading an image with the `ImageViewer` class and then pressing the "Adjust Levels" button. Only work with grayscale images from our collection at this point.

Step 10: Quantization

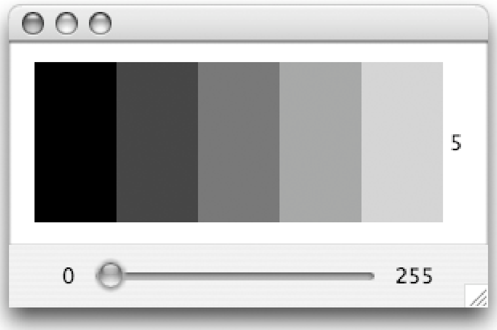
Early in the course, we emphasized that a defining feature of digital communication was that it depended on a finite set of discrete symbols to encode information. In the domain of images, this principle of digital communications manifests itself in the fact that we have been using 256 levels of brightness 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 `AdjustLevels` 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 shade 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, you should then set its brightness to its `x` coordinate divided by and then multiplied by `widthOfBrightnessBands`.

Before you attempt to run your program change the range of the slider from 0-255 to 1-255 since trying to draw an image using 0 colors is not a good idea.

- Compile and test your program until it draws nice, segmented gradients.

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 `AdjustLevels` class in Step 9.

- First, add a “Requantize” button to the `ImageViewer` class. Arrange for it to be enabled and disabled like the “Adjust Levels” button.

- Next, add code to the `buttonClicked` method in `ImageViewer` so that it will construct a new `Quantizer` when the "`Requantize`" button is pressed. It should pass the currently loaded image as a parameter in the `Quantizer` construction.
- Then modify the `Quantizer` class so that its constructor expects an `SImage` as a parameter and associates this `SImage` with an instance variable.
- Change the `makePixelArray` method into an `adjustPixelArray` method by making it accept a pixel array and the width and height of the image corresponding to the pixel array as parameters instead of creating its own monotone `SImage` and getting that image's pixel array.
- Change the code in the loop within `adjustPixelArray` 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 `adjustPixelArray` the values obtained by invoking the `getPixelArray, getWidth,` and `getHeigth` on the `SImage` that was passed to the `Quantizer` constructor.

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

Step 11: 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 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 )
```

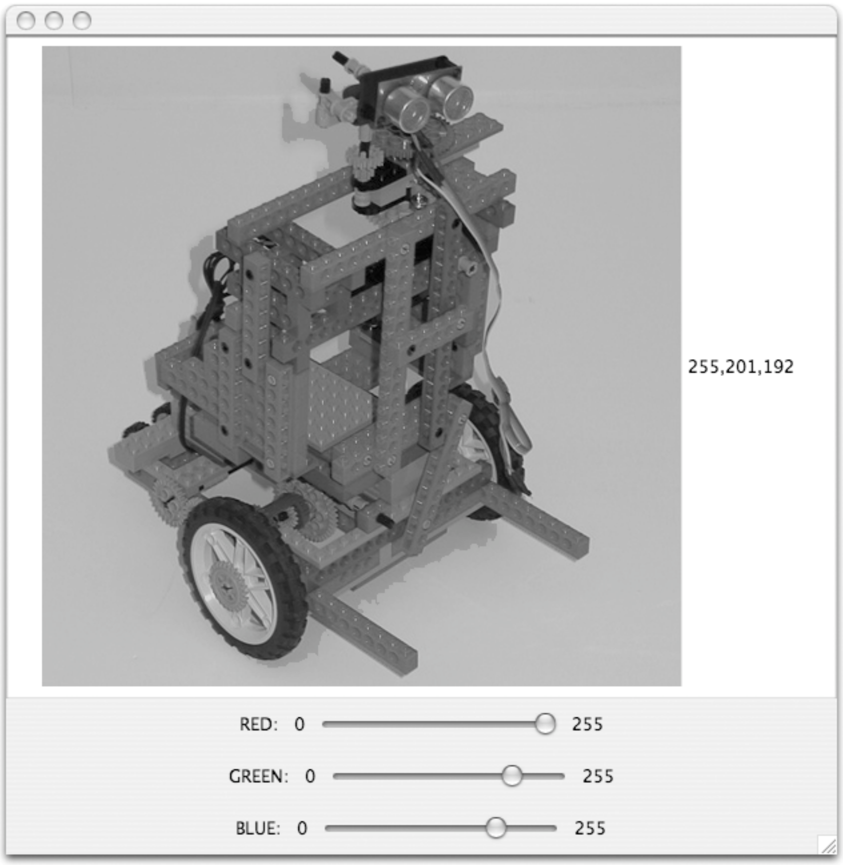
As a result, you can now fairly easily modify your `AdjustLevels` and `Quantizer` classes so that they will work correctly on color images. For each class:

- Apply the `adjustPixelArray` method you have already written to each of the three arrays obtained by invoking `getPixelArray` on the `SImage` passed to the constructor using parameter values `SImage.RED`, `SImage.GREEN`, and `SImage.BLUE`.
- Pass the three arrays returned by the three invocation of `adjustPixelArray` as parameters to the construction of a new `SImage`.
- Display the image produced using the `setIcon` method of your `JLabel`.
- Compile and test the modified class using the `ImageViewer` class.

Step 12: 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 `AdjustLevels` class so that its interface looked like the window shown on the right.

You could also add completely new operations to the `ImageViewer` class. Use your imagination (or if you are tired enough, just turn in what you completed in the first 11 steps).



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`.

- 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

Completeness (14 points) / Correctness (6 points)

- File dialog displayed when Load button pressed
- Images displayed correctly in `ImageViewer`
- `ImageViewer` buttons create appropriate processing windows.
- `AdjustLevels` reduces brightness of images
- `Quantizer` correctly requantizes images

Style (10 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

Appendix: Summary of New Library Features Used in this Lab

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 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.