

Lab 7

NotNotPhotoshop

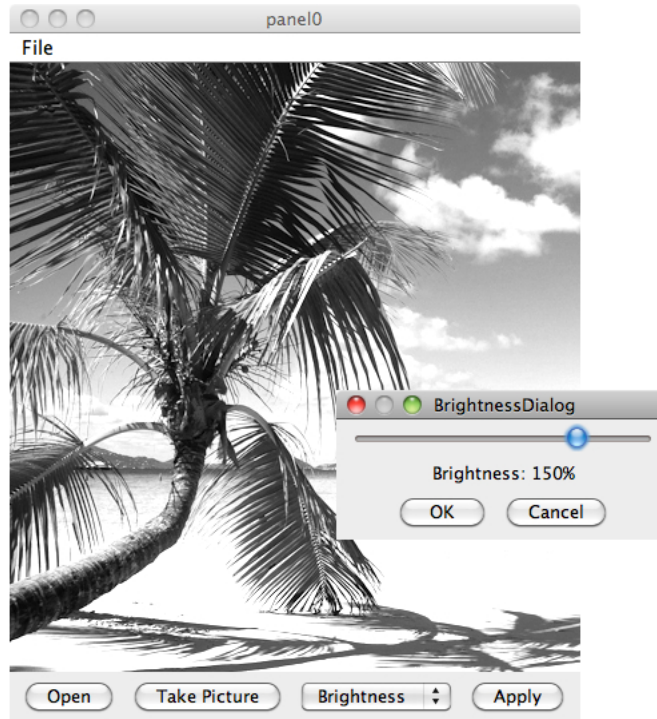
Objective. To gain experience using arrays, manipulating image data, and using inheritance.

As in the previous weeks, you are welcome and encouraged to work with a partner.

The Scenario. Your lab assignment this week is to implement a simple photo editing program. Your program will be able to open image files and perform a number of processing tasks on them. You will also have the chance to experiment with many other image transformations if you like. The interface to the program contains an image and a row of controls on the bottom:



The “Open” button opens a dialog box through which the user can load an image file from the hard drive. The “Take Picture” button uses the computer’s built-in camera to take a picture to display. The next component is a `JComboBox` that lists all of the operations we can apply to the image, and the “Apply” button applies the selected operation to the image. When an operation (or transformation) is applied, the image is updated to include the appropriate changes and a dialog box appears that enables the user to adjust the parameters of the transformation and either accept or cancel the change. For example, the following shows the result of clicking “Apply” when the “Brightness” transformation is selected:



The slider adjusts how much brighter (or darker) the image becomes. As it is dragged, the image in the original window is updated to reflect the new brightness level. The user can then either click “OK” to keep the new image or “Cancel” to revert the image to what it was before.

Before coming to lab, please read at least the next two sections. The first reviews the lecture material on images and arrays. The second describes the four transformations you will implement and specifies which parts of the program you must design before lab. We will come around and check the designs at the beginning of the lab session.

Image Representation and Manipulation

The contents of an `Image` are represented as a grid of pixels, where the color intensity of each pixel is encoded using integers. For color images, we describe the color of a pixel with three numbers for the intensity of red, green, and blue at that pixel, just as when creating `Color` objects. For gray-scale images, a single number from 0–255 can be used to describe the brightness of each pixel. We’ll stick to gray-scale images for the moment.

Consider the following image of an eye:



Zooming in more closely gives us a better view of the shades of gray assigned to each pixel:




The table below shows the numeric values for the gray-scale intensities of each pixel:

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

In Java programs, such tables are stored as two-dimensional arrays of integers. We can obtain the array of intensities for an image using the `Images.imageToPixelArray` method:

```
Image eye = getImage("eye.gif");
int [][] pixels = Images.imageToPixelArray(eye);
```

We can then change the pixel array and convert it back to an `Image`. For example, the following code darkens the eye image to appear as  and displays it on the screen:


```
Image eye = getImage("eye.gif");
int [][] pixels = Images.imageToPixelArray(eye);

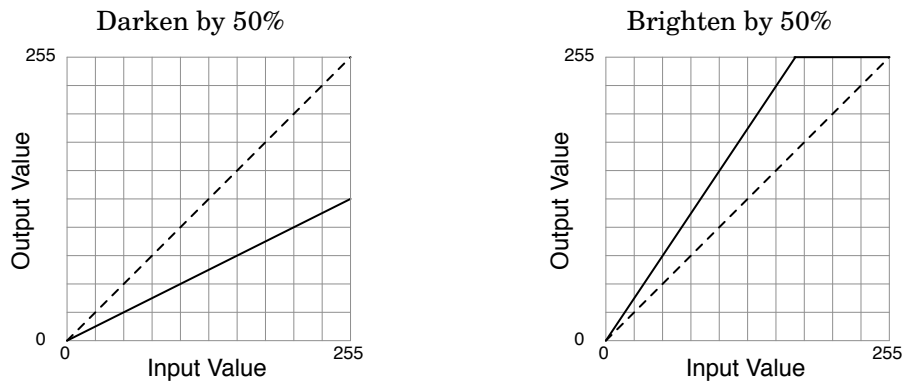
for (int x = 0; x < pixels.length; x++) {
    for (int y = 0; y < pixels[0].length; y++) {
        pixels[x][y] = pixels[x][y] / 2;
    }
}

Image darker = Images.pixelArrayToImage(pixels);
new VisibleImage(darker, 0, 0, canvas);
```

Note that individual elements in the `pixels` array are accessed using subscripts: the top-left pixel is

`pixels[0][0]`, the pixel immediately to its right is `pixels[1][0]`, the pixel immediately below is `pixels[0][1]` and so on.

More generally, we can think about the nested for loops above as mapping each input pixel value in an image to a new output pixel value that is half as large. The “adjustment curve” on the left below captures the relationship between input values and output values. (The dashed line on that plot represents the “identity” transformation that does nothing.) The plot on the right captures an adjustment that increases each pixel value by 50%, which would lead to the brighter eye . Note that any input values greater than 170 become 255 (pure white).



Remember that you can also create new arrays of any dimension you like, as in

```
int [][] pixels = new int[100][200];
```

which creates an array with 100 columns and 200 rows.

———— The Four Transformations ————

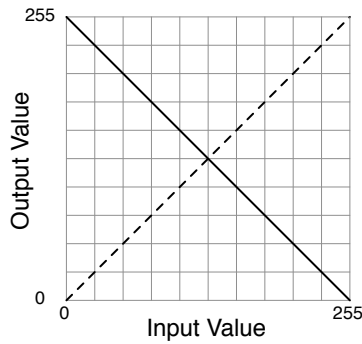
We’re now going to examine the four image transformations you will write as part of this lab. As in the lecture examples, the central part of each transformation will be a method

```
int [][] adjustPixelArray(int [][] pixels) {
    ...
}
```

whose parameter is the displayed image’s `pixels` array. It returns an array of pixels with the transformation applied.

Sketch the implementation of this method for each transformation before lab.

Inverting. Our first transformation inverts the pixel values for the image to make it look like a film negative. That is, a white pixel becomes black, a black pixel becomes white, and so on. We can think about `invert` as the following function from Input to Output Value. Applying this function to the beach image yields the image shown below:

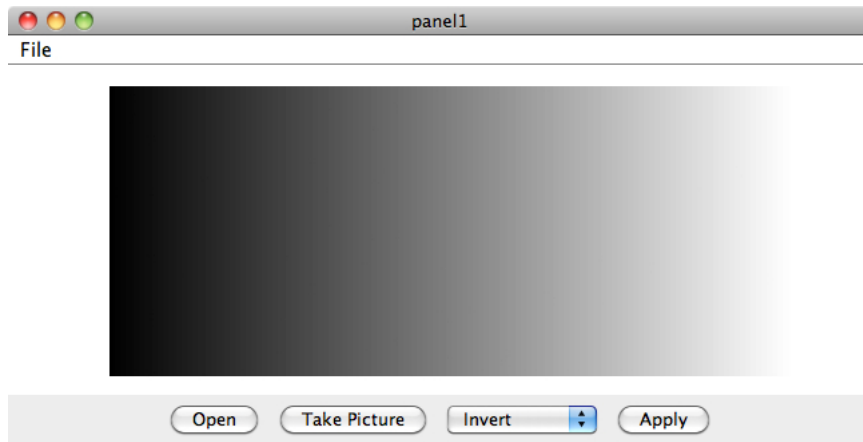


Brightness. Your brightness adjustment will apply a transformation similar to the one in the previous section that halved the intensity of each pixel. However, your brightness transformation should support darkening/brightening by a range of values rather than just darkening by 50%. As described below in the implementation plan, the desired percent change will come from a slider named `percent`, which will be in the range 0–200. A new pixel value is then computed by treating this number as a percentage change:

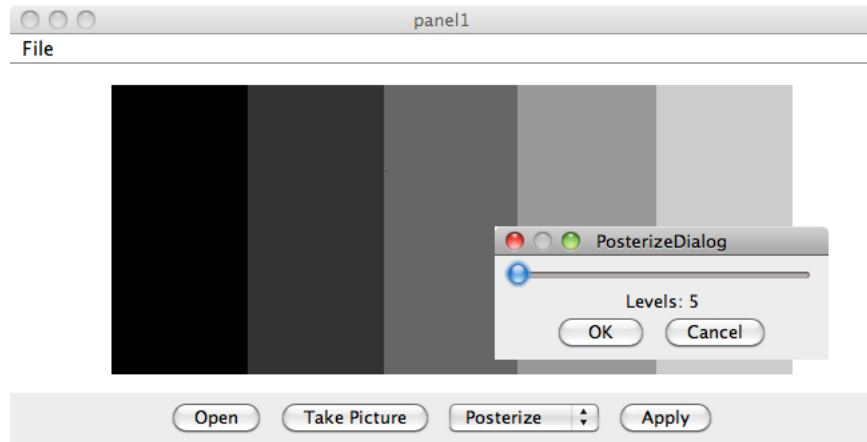
```
newPixelValue = percent.getValue() * oldPixelValue / 100;
```

Note that our method to convert pixel arrays to `Images` treats all pixel values greater than 255 as 255, so you don't need to worry about creating numbers that are too large when adjusting the brightness of each pixel. Similarly, all negative numbers are treated as 0.

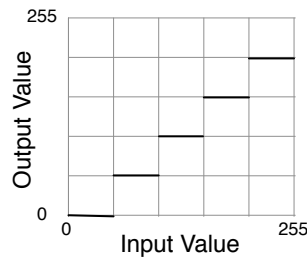
Posterizing. Posterization refers to the process of reducing the number of shades used in an image. The figure below shows an image that has a gradient from black to white, using all 256 pixel intensities from 0 to 255.



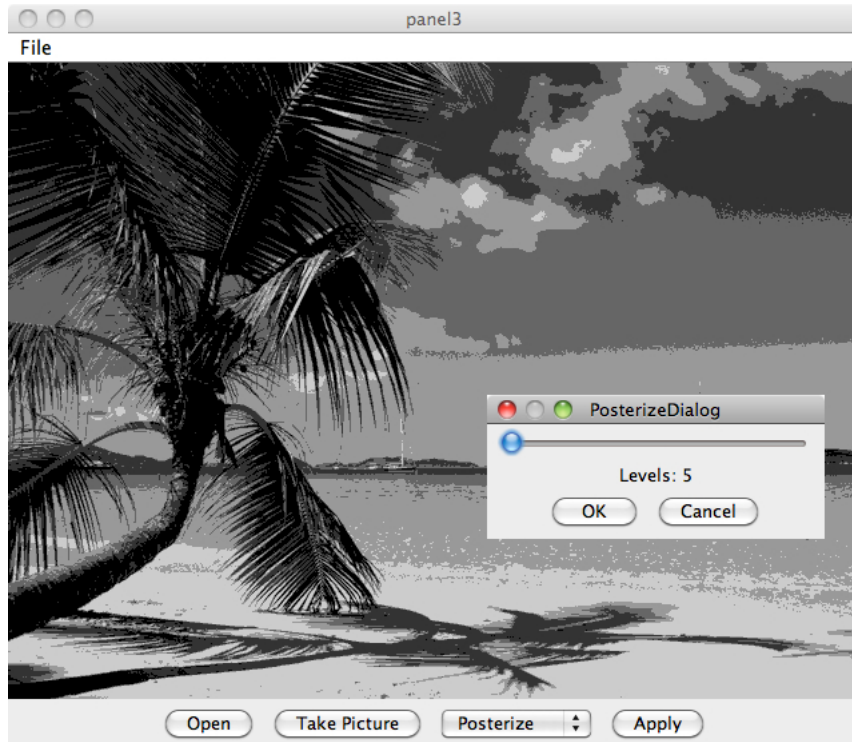
The following is a posterized version that uses only five levels of brightness.



The brightness levels shown are determined by dividing the 256 brightness levels into five 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 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. The mapping from original brightness to new brightness can be captured with the following diagram:



If we were to posterize our beach scene to just five shades, we would have:



Suppose we posterize to n different levels of brightness. Then $256/n$ will tell us how wide each band of our gradient should be. Now, suppose that we want to decide the new shade for a `pixels[x][y]` that originally has brightness b . If we divide b by $256/n$, Java will give us the integer portion of the result of this division. If b is in the range $0 - 256/n$, the result 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 new shade of `pixel[x][y]`. That is, in your `adjustPixelArray` method, you should:

- Define a local variable with a name like `widthOfBrightnessBands` and initialize its value to be: `256/levels.getValue()`, where `levels` is a `JSlider` your program will create.
- For each pixel, set its brightness to its original brightness divided by `widthOfBrightnessBands` and then multiplied by `widthOfBrightnessBands`.

Flipping. The Flip transformation adjusts an image by flipping it horizontally. The original and flipped beach are shown below:



Flipping is a little different than the previous transformations. To compute the new value for a pixel at column x and row y , those other transformations apply some function to the original value of `pixels[x][y]`. When flipping an image, the new value for the pixel at column x and row y is, in contrast, computed by looking at pixel values other than `pixels[x][y]`. That will affect how you write your `adjustPixelArray` method.

Design of the program

The Handouts web page contains a starter project as well as a collection of images for you to use. See the “Getting Started” Section at the end of the handout for instructions on how to set these up. The `NotNotPhotoShop` window controller creates an initial, simplified GUI that you will use for the first few steps below. The end of the handout contains the code for this class. It sets up two `JButton` instance variables, `openFileButton` and `snapshotButton`, for opening files and using the camera, and it loads an initial image into the `display` instance variable.

Part 1: Loading and Displaying Images. Your first step is to extend the window controller to load and display any image file when the user presses the “Open” button. Swing provides a class named

`JFileChooser` that makes it fairly easy to do this. In `actionPerformed`, you should perform the following steps when the source is the “Open” button:

- First, create a new chooser object:

```
JFileChooser chooser = new JFileChooser();
```

- Second, tell the `chooser` to let the user select a file to open:

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

- Finally, load the image file that was selected by the user. We do this with the following two lines:

```
String name = chooser.getSelectedFile().getAbsolutePath();  
Image newImage = Toolkit.getDefaultToolkit().getImage(name);
```

The first line computes a `String` describing the exact location of the file selected by the user (such as “/Users/freund/cs134/examples/beach.jpg”). The second line then uses a Swing library method to load the image file specified by that `String`. (This method is a bit more flexible than the `getImage` method we’ve used in the past, which is limited to only loading images out of the project directory.) Once loaded, you can then set the image being displayed to that `newImage` by changing the image showing in the `display` instance variable.

Run the program and be sure this works. When you do so, you may notice 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 the constant `JFileChooser.APPROVE_OPTION`.

Therefore, you will need an “if” statement of the form shown below to ensure you only load an image when one was selected.

```
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.

Part 2: Camera. While you may enjoy looking at a warm tropical beach, nothing could possibly compare to a picture of your own smiling face. To give you the pleasure of seeing that sight, we will now add the ability to take selfies using the computer’s built-in camera when the “Take Snapshot” button is pressed.

As we saw in lecture, capturing an image requires only two lines of code:

```
Camera camera = new Camera();  
Image newImage = camera.getImage();
```

Extend your program’s `actionPerformed` method to take a picture and update `display`’s image accordingly. Run your program and say “Cheese”. (Cameras may or may not work on your own computers but they will in our labs...)


```

public class InvertDialog extends JDialog implements ActionListener {
    // Location and size of dialog box
    private static final int LOCATION_X = 400;
    private static final int LOCATION_Y = 200;
    private static final int WIDTH = 250;
    private static final int HEIGHT = 140;

    private JButton okButton = new JButton("OK");
    private JButton cancelButton = new JButton("Cancel");

    // The image display in the canvas of the main window
    private VisibleImage display;

    // The Image representation of the displayed image prior to transforming it.
    private Image original;

    // Configure dialog box appearance, create ok/cancel buttons, add listeners.
    public InvertDialog() {
        this.setModal(true);
        this.setTitle(this.getClass().getName());
        this.setLocation(LOCATION_X, LOCATION_Y);
        this.setSize(WIDTH, HEIGHT);
        this.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);

        Container contentPane = this.getContentPane();
        JPanel buttons = new JPanel();
        buttons.add(okButton);
        buttons.add(cancelButton);
        contentPane.add(buttons, BorderLayout.SOUTH);
        contentPane.validate();

        okButton.addActionListener(this);
        cancelButton.addActionListener(this);
    }

    /*
     * Apply the image transformation and ask user to accept/reject.
     */
    public void applyAdjustment(VisibleImage theDisplay) {
        // remember the visible image we are to update, as well as the image currently being shown.
        display = theDisplay;
        original = display.getImage();
        this.adjustImage();
        this.setVisible(true);
    }

    /*
     * Handle both button events. Right now, we just close
     * the dialog box by setting it to not be visible.
     */
    public void actionPerformed(ActionEvent e) {
        this.setVisible(false);
    }

    // Applies the desired transformation to the image.
    protected void adjustImage() { /* change */ }
}

```

Figure 1: The InvertDialog class.

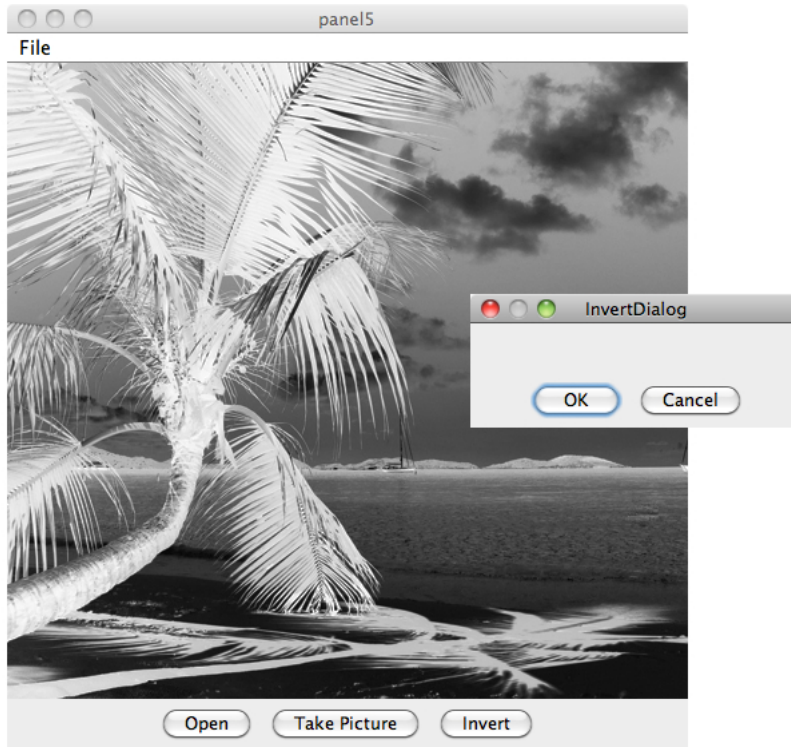


Figure 2: The result of applying the “invert” transformation.

Part 3: Invert. The next step is to implement “Invert”, our first transformation. The transformation itself is relatively straightforward, but we’re going to carefully design the code for it so that we may re-use it for the other transformations.

First, add an “Invert” button to your interface, next to your other buttons, and change `actionPerformed` to handle events generated by clicking on it as follows. To add some space between the two buttons, our sample solution creates a new `JLabel` with a long string of spaces and adds it to the panel between the buttons, as in:

```
bottomPanel.add(new JLabel("                "));
```

When the user clicks this button, your program will create a new object and ask it to do two things: 1) invert the displayed image and 2) display a dialog box to allow the user to either accept the adjustment or cancel and revert to the original image, as in Figure 2.

We have provided a partially completed `InvertDialog` class definition to help you get started (see Figure 1). It demonstrates how to create a new class that extends `JDialog`, the Swing class for dialog boxes. `JDialogs` are used in much the same way as `WindowControllers`. For example, they have a `contentPane` to which we can add components, and they can listen for events from components if they implement the appropriate listener. The `InvertDialog` constructor takes no parameters, and we create one using a command like:

```
InvertDialog invert = new InvertDialog();
```

You can then invoke `invert.applyAdjustment(display)` to ask the newly created object to transform the `VisibleImage display`, open up a dialog box, and wait for the user to either accept or cancel the change. If you look at the code we provide for that method, it just uses a helper method `adjustImage` to apply the transformation and then invokes `this.setVisible(true)` to make the dialog box be visible on the screen. Right now, the `InvertDialog` class doesn’t do anything interesting to the displayed image. That’s the next step...

The `InvertDialog` class contains two instance variables that are initialized in `applyAdjustment`:

- `display` is set to be the `VisibleImage` that we will adjust.
- `original` is the `Image` that `display` was showing prior to transforming it.

We will structure the transformation code in `InvertDialog` much like the programs we wrote in lecture. That is, you will write the helper method

```
protected void adjustImage()
```

that will get the pixel array for the original image, update the array to have the inverted pixel values, and then set the `display` to show an `Image` created with this updated array. You may notice we declared this helper method to be “protected” rather than “private”. You’ll see why shortly... The code you need to write for this method will be identical to the examples in lecture:

```
int[][] pixels = Images.imageToPixelArray(original);
int[][] newPixels = this.adjustPixelArray(pixels);
Image newImage = Images.pixelArrayToImage(newPixels);
display.setImage(newImage);
```

All you need to do is define and write the method

```
protected int[][] adjustPixelArray(int[][] pixels)
```

to compute the pixels in the inverted image. This method is also protected. Run the program and verify that the transformation works.

The final step is to enable the user to cancel the transformation. We can use the original image again, and simply set the image shown in `display` to be the original image if the user presses the “Cancel” button. Change `actionPerformed` to do this and test your `InvertDialog` on several images.

Part 4: Flip. We’ll now write our second transformation. Add a “Flip” button to your interface. When this button is clicked, your `actionPerformed` method should create a `FlipDialog` object and invoke `applyAdjustment(display)` on it.

Of course, we do need to define the `FlipDialog` class before we can actually use it. We want the behavior to be almost exactly like the `InvertDialog` class. In fact, the only difference will be how we adjust the pixel array. So rather than writing it from scratch, create the `FlipDialog` as follows:

- Open the class called `FlipDialog`.
- Copy the entire contents of the `InvertDialog` class and paste it into the `FlipDialog` class. Be sure to change the class name and constructor name to be `FlipDialog` and then save the file.
- Run the program and verify that you can create a `FlipDialog`, although it will behave like the other dialog and invert the image.
- Finish writing the `FlipDialog` class by changing its `adjustPixelArray` method to flip values in the pixel array. Test it to be sure it works.

Part 5: The Meek Shall Inherit. Before continuing on to the other transformations, let’s reflect on `InvertDialog` and `FlipDialog` for a moment. We seem to be violating one of our common goals for beautiful code: don’t write the same thing twice. However, with a few short editing steps and a new programming feature called *inheritance*, we can create a much more pleasing solution...

- Modify the header of your `FlipDialog` class so that it says “extends `InvertDialog`” instead of “extends `JDialog`”. That change tells Java that you want to use `InvertDialog` as the starting point for constructing the `FlipDialog` class. That is, `FlipDialog` now *inherits* all of the features of an `InvertDialog` without us having to repeat them. In describing their relationship, we say that `InvertDialog` is a superclass of `FlipDialog` and that `FlipDialog` is a subclass of `InvertDialog`.

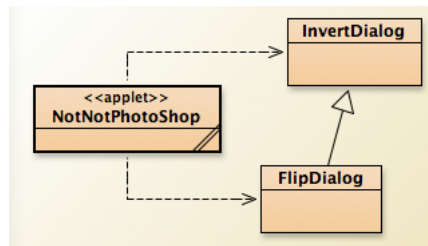
- Remove from `FlipDialog` anything that is the same as what is inherited from the superclass. That is, remove everything except `adjustPixelArray`, the one method that is different. This should leave us with a class that looks like the following:

```
public class FlipDialog extends InvertDialog {

    protected int [][] adjustPixelArray(int [][] pixels) {
        ... your code to construct the flipped array ...
    }
}
```

Run your program again. It should work exactly as it did before. Even though we didn't explicitly declare that `FlipDialog` has "Okay" and "Cancel" buttons, they appear and behave as before, because we inherit those instance variables and behaviors from the superclass and get to use them for free. Similarly, when we create a `FlipDialog`, it will adjust the image it was given using the same code as what appears in the superclass's `adjustImage()` method. However, when that code invokes `adjustPixelArray`, something happens that is different because we redefined the behavior of that method by providing a new definition in the subclass. Previously, we had declared that helper method to be `protected` to indicate that we would like subclasses to be able to use it and change its behavior, as we have done here.

If you look at the project window in BlueJ, this inheritance relationship is captured with the hollow arrows:



Inheritance is a very handy feature that we've been using all along. Think about our `WindowController` subclasses – they all inherit a lot of functionality from the `WindowController` superclass (such as having a `canvas` and a `window` on the screen), and we have always just defined what is different (such as how it should respond to mouse clicks).

Part 6: Housekeeping. As we write other transformations, we could continue to extend `InvertDialog` with other subclasses. However, from a design point of view, it is a little odd to have a transformation class like `FlipDialog` extend a class that already does something for us (i.e., inverts an image's pixels). A better design is to write a general class `AdjustmentDialog` that provides all the common scaffolding for writing transformations and have *every* real transformation be a subclass of it. To set it up in this way:

- Open `InvertDialog` and change the class name and the constructor name to `AdjustmentDialog`.
- Change `FlipDialog` to extend the `AdjustmentDialog` class.
- Also, create a new `InvertDialog` class that is a subclass of `AdjustmentDialog`. This is actually really easy — it looks almost exactly like `FlipDialog`, the other subclass we just finished. The only difference is what `adjustPixelArray` does. You can copy that method from the `AdjustmentDialog` class, where it is still defined to invert the image:

```
public class InvertDialog extends AdjustmentDialog {

    protected int [][] adjustPixelArray(int [][] pixels) {
```

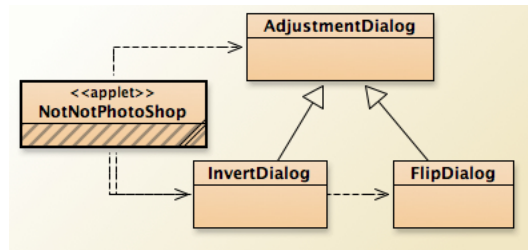
```

        ... code to invert pixel array. (Take from AdjustmentDialog) ...
    }
}

```

- Finally, it doesn't make sense for the `AdjustmentDialog` to invert the image anymore, so we can change the `adjustPixelFormat` in `AdjustmentDialog` to just return the `pixels` parameter without changing it.

The project window will now be similar to the following, in which the relationship between our general `AdjustmentDialog` class and its subclasses are shown with hollow arrows.



Run your program and verify that it still works correctly.

Before continuing, also change the window controller's interface to have a `JComboBox` listing the available transformations, and replace the "Invert" and "Flip" buttons with a single "Apply" button. When the "Apply" button is clicked, your `actionPerformed` method should get the selected item from the `JComboBox` and create the right transformation.

Part 7: Brightness. The next step is to implement a `BrightnessDialog` subclass of `AdjustmentDialog`. To get things started, add "Brightness" to the `JComboBox` in the window controller and extend your `actionPerformed` to create a `BrightnessDialog` object when the "Apply" button is clicked with "Brightness" selected.

There is one big difference between this adjustment and the ones you've written. It requires a slider to adjust the how much brighter the image will be. Let's call that slider `percent`. A good range for the slider is 0 to 200.

Create a new `BrightnessDialog` that is a subclass of `AdjustmentDialog`. Add the `JSlider` instance variable `percent`, and a constructor that creates and adds the slider to the dialog box's content pane, just as you would in a `WindowController`. The initial value for the slider should be 50.

Run the program. It should create the brightness dialog box, but it will not do anything yet. Write its `adjustPixelFormat` method to perform the brightness adjustment to the array.

If you run and test your program at this point, you will notice that the brightness is adjusted to 50%, but we don't get to see the image change brightness as we move the slider back and forth. To complete the `BrightnessDialog`, make that class listen for `ChangeEvent`s on the slider, and whenever the slider changes, reapply the brightness adjustment to the original image by invoking `this.adjustImage()`. Since that helper method was declared to be protected in the superclass, we are able to invoke it in the subclass.

You may wish to add a `JLabel` showing the current value of your slider to your dialog box at this point as well. (If you wish to change the size of the dialog box to fit more components, you can simply call `this.setSize(width, height)` in your constructor.

Part 8: Posterize. Create a `PosterizeDialog` following the same steps as we just performed to create the `Brightness` dialog. The only two pieces that will be a little different are the range of values for the slider (which we'll call `levels` this time), and what exactly happens in `adjustPixelFormat`. A good range for `levels` is 2–256.

Write the posterizing `adjustPixelArray` method you designed ahead of time, and test your program on the “gradient.gif” provided in the “cs134-images” folder until it draws nice, segmented gradients. You will probably have to set the slider to a very small value to see the segments.

(Optional aside: you may notice that our posterize adjustment never uses the brightest white colors, since each band is shown as the darkest shade mapping to that band. How would you change your code to use the full range of shades (0 to 255) for the bands? That is, the darkest and brightest bands should appear as black and white.)

Part 9: Now in Color. So far, our adjustments have all been applied to a pixel array representing gray-scale intensity values obtained by invoking `Images.imageToPixelArray(original)`. However, we can also extract color information from color images. In particular, we can ask for an array representing the intensities for any of the three color components (or channels) by invoking, for example, `Images.imageToPixelArray(original, Images.RED_CHANNEL)`.

To make all of our transformations work on color images, we only need to make one change. In particular, we will rewrite the `adjustImage()` method in `AdjustmentDialog` to transform these three arrays separately, and then build a new image based on the three new color-specific arrays using `Images.pixelArraysToImage`, which takes three arrays as parameters:

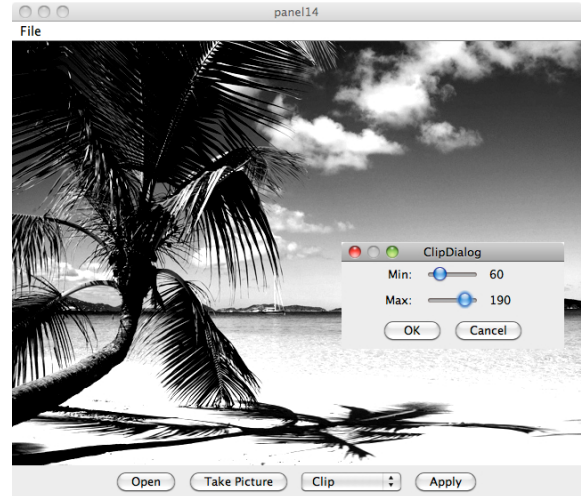
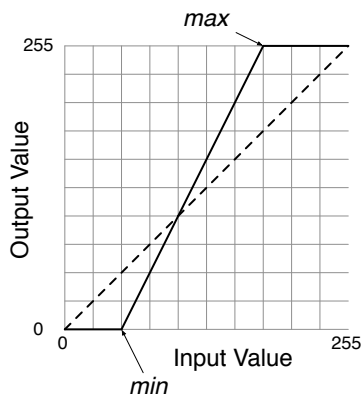
```
int[][] red = Images.imageToPixelArray(original, Images.RED_CHANNEL);
int[][] green = Images.imageToPixelArray(original, Images.GREEN_CHANNEL);
int[][] blue = Images.imageToPixelArray(original, Images.BLUE_CHANNEL);

Image newImage = Images.pixelArraysToImage(this.adjustPixelArray(red),
                                           this.adjustPixelArray(green),
                                           this.adjustPixelArray(blue));
```

All subclasses will then use this new definition, and all will be able to work on color images essentially “for free”. Modify your `AdjustmentDialog` in this way and test all your transformations in color. The samples folder contains several color images for you to use.

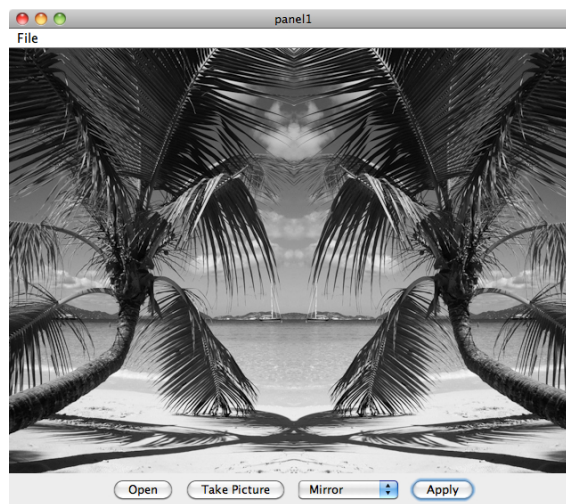
Part 10: More Image Fun... There are many other transformations you can apply to images. Feel free to add any that you’d like to experiment with. They can be as simple or as sophisticated as you like, and, as always, you may leverage anything from the lecture examples. Playing with the options in PhotoShop’s “Filters” menu may give you many ideas. Here are a few to get you started:

- **Border:** Add a black border around the edge of an image. (Possibly add a slider to change the thickness of the border.)
- **Vertical Flip:** Implement a transformation to perform vertical flipping of images. (Or one class that can do both horizontal and vertical...)
- **Clip:** Convert all pixel values below a minimum threshold to black, and all pixel values above a maximum threshold to white. Everything in between is scaled proportionally:

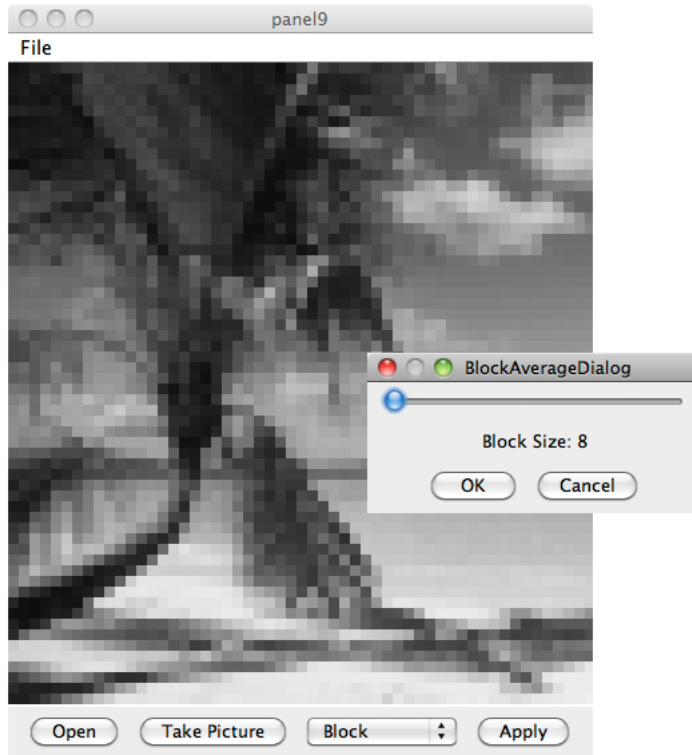


You will need two sliders to capture the minimum and maximum cutoff.

- **Mirror:** Make the right half of the image a mirror image of the left half:



- **Edges:** Implement edge detection from lecture.
- **Blur:** Make an image blurry. An easy way to do this is to compute the new value for `pixels[x][y]` as the average of that pixel and the eight pixels immediately surrounding it.
- **Block Average:** Divide the image into small, $n \times n$ blocks, compute the average pixel value of all the pixels in the block, and use that average as the value of every pixel in the block in the new image. Here's an example:



Getting Started

We have provided a starter project for you to use. To download the starter project, visit

<http://www.cs.williams.edu/~cs134/>

and then follow the link to the Handouts page. Click on the link for “Starter Code” under Lab 7.

This will download a file archive called `Lab7Images.tar.gz` to your `cs134` folder. The files of this archive will then be extracted to a folder in your `cs134` folder called `Lab7Images`. (If this does not happen automatically, simply double-click on the downloaded `Lab7Images.tar.gz` file to extract the archive.) You may delete the `Lab7Images.tar.gz` file at this point. Rename the “`Lab7Images`” folder to include your last name and open the project folder in BlueJ.

The Handouts page also contains an archive called `cs134-images.tar.gz`. Download this file and extract it, as above. This folder includes some images that will be useful for testing.

Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder. You can do this by clicking on the smiling Macintosh icon in your dock.
- From the “Go” menu at the top of the screen, select “Connect to Server...”.
- For the server address, type in “`afp://Guest@fuji`” and click “Connect”.
- A selection box should appear. Select “Courses” and click “Ok”.
- You should now see a Finder window with a “`cs134`” folder. Open this folder.
- You should now see the drop-off folders for the three lab sections. Drag your “`Lab7Images`” folder into the appropriate lab section folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click “OK”.
- Log off of the computer before you leave.

You can submit your work up to 11 p.m. on Wednesday if you’re in the Monday night lab; up to 6 p.m. on Thursday if you’re in the Tuesday morning lab; and up to 11 p.m. on Thursday if you’re in the Tuesday afternoon lab. If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before your lab deadline. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like the word “revised”) and the re-submission will work fine.

The NotNotPhotoShop WindowController

```
public class NotNotPhotoShop extends WindowController implements ActionListener {
    // Size of window when created
    private static final int WINDOW_WIDTH = 640;
    private static final int WINDOW_HEIGHT = 560;

    // Buttons to open a new image file and take a picture with the camera
    private JButton openFileButton;
    private JButton snapshotButton;

    // The image we are editing
    private VisibleImage display;

    /*
     * Create the GUI components and load the original image from the hard drive.
     * You will change this method when adding new componets to the interface.
     */
    public void begin() {
        this.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

        // Create the buttons and add them to the window
        openFileButton = new JButton("Open");
        snapshotButton = new JButton("Take Picture");

        JPanel bottomPanel = new JPanel();
        bottomPanel.add(openFileButton);
        bottomPanel.add(snapshotButton);

        // Add the components to the window
        Container contentPane = getContentPane();
        contentPane.add(bottomPanel, BorderLayout.SOUTH);
        contentPane.validate();

        // Make me listen for button clicks
        snapshotButton.addActionListener(this);
        openFileButton.addActionListener(this);

        // load the beach scene at startup
        display = new VisibleImage(getImage("gray-1.jpg"), 0, 0, canvas);
    }

    /*
     * Handle events from the buttons.
     */
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == openFileButton) {
            // add code here to load file
        } else if (e.getSource() == snapshotButton) {
            // add code here to use camera
        }
    }
}
```