Computer Science Department

Mount Holyoke College

CS 101
Problem Solving and Object-Oriented Programming
Lab 4 - A Picture is worth 1000 Pixels
Due: October 16/17/18/19

Pre-lab Preparation

Before coming to lab, you are expected to have:

- Read Bruce chapter 7
- Write a program that draws a horizontal chain by drawing tiny overlapping circles. The circles (chain links) should be circles that are 12 pixels by 12 pixels and should overlap each other by 4 pixels. The chain should contain 12 links across. The upper left corner of the chain should be at coordinates (50, 10). The chain should look like the image to the right.
- Use the code you wrote for problem 1 above, to solve exercise 7.11.1 in the text.
- You should turn in just 1 program, the one from exercise 7.11.1. Please put this program in your Lab4 folder, in a BlueJ project called Knitting. To receive credit, the time on the file must be before 1:15 PM on the day of your lab.

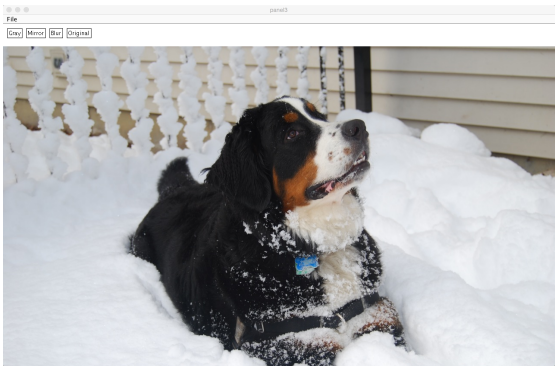
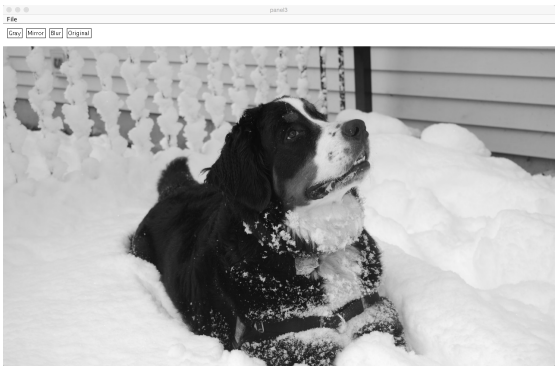
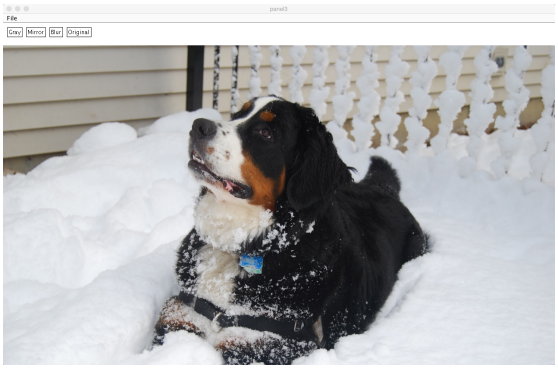
Goals:

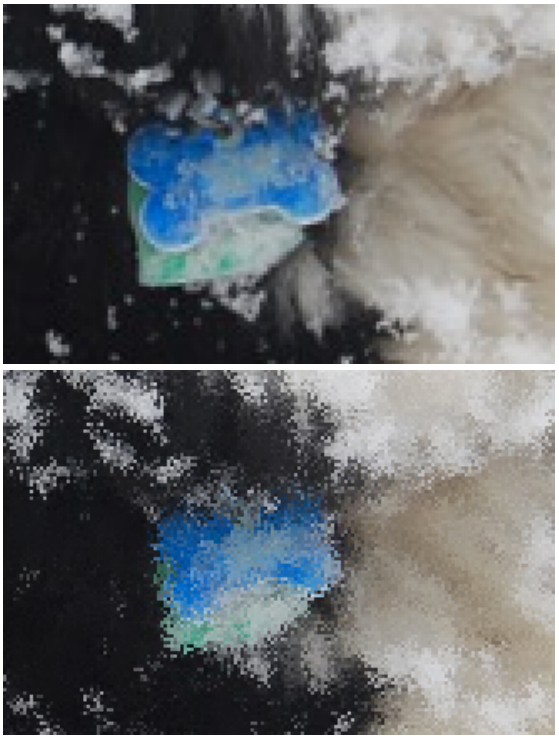
- To work with loops
- To gain some insight into how computers represent images on the screen

Introduction to the Assignment

In this lab, you will do some image manipulation. An image is a 2-dimensional grid of pixels. You will write some methods that will use loops to walk through the grid that makes up an image to manipulate the image. Specifically, you will write methods to convert an image to grayscale, to mirror an image and, for extra credit, to blur an image.

Here are some snapshots from this program:

	This is the original display. A picture is displayed and rectangles with text inside are placed across the top to serve as buttons that control the image manipulation.
	This is the image after clicking on the "gray" button. The image is replaced with a grayscale version of itself.
	This is the image after the original image is restored and the user clicks on the "mirror" button. The image is replaced with a mirror-image copy. That is, left and right are reversed.



This is a small portion of the image comparing that portion of the original image to the blurred image. Edges between things in the image are less well-defined. This is an **extra credit** step.

The Program Design

This project will have 3 classes: Picture, Button and Image Manipulator.

You can download Picture.java from the course website. This class is completely written for you. **YOU SHOULD NOT CHANGE THE PICTURE CLASS!**

The ImageManipulator class will be the only class that extends WindowController. It is responsible for defining the begin method and the event handlers.

- begin method - This method should draw the display consisting of the buttons and an image.
- onMouseClick method - This method should determine which, if any, button has been clicked on. Depending on which button is clicked on, it should then call a private method to carry out the requested image manipulation.

Button defines labeled rectangles. An example button is shown on the right. It is responsible for drawing itself with the text centered and determining if it contains a Location. This class is very similar conceptually to the HotAirBalloon class that you made for lab 3. You will also find it useful to be able to ask a Button where its right and bottom edges are in order to layout your window nicely (Hint: These should be methods added to the Button class). The Button class should define:

- A constructor - This should take parameters for the left and top coordinates of the button, the text to display, and the canvas.
- A contains method - onMouseClick should call this method to determine if the user clicked on a button
- A getBottom method - this method should return the y coordinate corresponding to the bottom of the button

- A getRight method - this method should return the x coordinate corresponding to the right edge of the button.

Writing the Program

Create a new BlueJ project called Lab4. You should download Picture.java from the course website and import it into your project by copying and pasting the file into your Lab4 folder and restarting BlueJ. **YOU SHOULD NOT CHANGE PICTURE.JAVA.** If you want to use the picture shown in this handout, use "http://www.mtholyoke.edu/~blerner/SnickersInSnow2.jpg" for the name of the file when you call the Picture constructor. If you want to use a different image on the Web, you can use its URL. If you want to use a picture that you have in a file, copy the file containing the picture to your ImageManipulator folder so that we will have access to it when we test your program.

Step 1: Displaying an image

For the start of this lab, we ask you to **ignore the buttons and begin just with displaying an image**. You should create the ImageManipulator class and say that it extends WindowController. You will need to declare the begin method and put statements in the begin method to display an image.

To work with images, you will need to use 2 different classes (that are already written):

- The **Picture** class allows you to load an image from a file and extract individual pixels from an image.
- The **VisibleImage** class allows you to display an image on the canvas.

Loading an image from a file

Place an image file inside the folder that contains your BlueJ project. To load the image from the file, call the Picture constructor passing in the name of the file containing the image. Here is the signature of the Picture constructor:

```
public Picture (String filename)
```

Remember that you are **calling** this constructor, not declaring it (Hint - you cannot call a constructor from outside of a method - so make sure that you have an instance variable named picture **before** calling the constructor). The above line of code should help you determine how to call it; it should not appear in your program as written above.

The filename should be placed inside quotes, so when you call the Picture constructor you would say something like:

```
picture = new Picture (“SnickersInSnow.jpg”);
```

You can also pass in an http URL instead of a filename to the Picture constructor.

Displaying an image

Calling the Picture constructor only reads the image from the file. To place the image on the screen, you call the createVisibleImage method in the Picture class, which has this signature (Remember to declare a VisibleImage instance variable before setting it equal to to the result of createVisibleImage):

```
public VisibleImage createVisibleImage(double left, double top,
                                       DrawingCanvas canvas)
```

The image should be at the left edge of the window and a constant number of pixels from the top, leaving space for the buttons that you will add later.

In the begin method of the ImageManipulator class you should call the resize method to make the window an appropriate size for the image you are using. To find out the size of your image, you can use the getHeight() and getWidth() methods that are part of the Picture class. The signatures for those are:

```
public int getHeight ()
public int getWidth ()
```

The signature of the resize method is:

```
public void resize(int width, int height)
```

The resize method is defined in WindowController and so you can call it without providing a variable to call it on, like this, but substituting in the size of your image using the methods shown above to find the size, but make sure to add some extra height to allow for the buttons and menu that appear above the image:

```
resize (1280, 920)
```

Step 2: Grayscale

When the user clicks anywhere in the window, the image should change to grayscale. (Later we will change this so that the user needs to click on the Gray button.) To do this, your onMouseClick method should call a private method that you will write named grayScale:

```
private void grayScale ()
```

An image is a 2-dimensional grid of pixels. Generally, pixels are too small to be distinguishable by the human eye. Here is a greatly magnified portion of an image to make the pixels visible:

Each colored square is an individual pixel. A digital photograph often has millions of pixels. The photograph I am providing has 1280 columns of pixels and 857 rows of pixels, for a total of 1280 x 857 = 1,096,960 pixels.

The Picture class provides 2 methods to allow you to manipulate an image. You can find out the color of a pixel by providing a row and column coordinate. These coordinates are relative to the top left corner of the image (different from the top left corner of the canvas). A second method allows you to change the color of a pixel:

```
public Color getPixel (double row, double col);
public void setPixel (double row, double col, Color color);
```

To manipulate an image, you will use nested loops to visit each pixel in the image, get its color, calculate a gray color as described below, and then set this gray color to be the new color of that pixel.

In Java, we normally use RGB color. In RGB, there is a red value, a green value and a blue value. When all 3 values are the same, we get a shade of gray. A simple way to convert an image to gray scale is simply to average the red, green and blue values. Then create a new color passing this value in for each of the three color components. The **Color class** provides 3 methods that you can call to get the red, green and blue values of a Color:

```
public int getRed()
public int getGreen()
public int getBlue()
```

They each return a value between 0 and 255.

After the Picture object is updated, you need to redisplay it. To do this, first remove the existing visible image from the display by calling:

```
public void removeFromCanvas()
```

on the VisibleImage. Then use the createVisibleImage method just like you did when you originally displayed the image.

Your grayScale method should contain the loops described above to convert the image to grayscale as well as the code to display the gray image.

Step 3: Displaying the buttons

The buttons should be displayed when the program starts. Therefore, you will need to modify the begin method and put statements in it to construct the 4 buttons.

Create a Button class to implement a button. Within the Button class, you will draw a button consisting of a FramedRect object with a Text object centered within it. The Button constructor should have the following signature:

```
public Button (double left, double top, String label, DrawingCanvas
canvas)
```

The text should be centered within the box, with a constant amount of whitespace between the button border and the text, like shown on the right:

The size of the rectangle depends on the size of the text. Unfortunately, you do not know how big the text will be until after you create it. So, you need to first create the Text object someplace other than its final location. Then, you can call the getWidth and getHeight methods on the text object to determine its size.

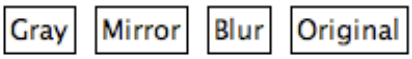
Using this information, knowing the amount of whitespace you want to have around the button's text, and the left and top parameters passed to the Button constructor, you should be able to create the rectangle.

Next, you need to move the text so that it is centered in the rectangle. To do this, use the normal centering equation (Hint: buttonCenterX and buttonCenterY) to determine where the text should be. Then, use the moveTo method, which is defined in the Text class, to move the text to its proper centered location. The moveTo method has the following signature:

```
public void moveTo (double x, double y)
```

To create 4 buttons, you will need to call the Button constructor 4 times from your Image-Manipulator's begin method. Each time you will pass in a different value for the left edge of the button and the label for the button. The buttons all line up nicely on the top edge, so they should all have the same value for the top parameter. You will also need 4 instance variables whose type is Button, one to hold each of the 4 buttons.

The row of buttons should look something like this:



Notice how there is a fixed amount of space between buttons. Where should the Mirror button go? It should be some constant number of pixels to the right of the Gray button. But where is that? To determine the answer to that question, declare a method in the Button class with this signature:

```
public double getRight()
```

This method should return the x coordinate of the right edge of a button. Now, your begin method should be able to figure out where the left edge of each button should be.

Step 4: Clicking on a button

Now, you should change your onMouseClick method so that it determines if the user has clicked on one of your buttons instead of always turning the image to grayscale. For now, the only functionality that you have implemented is grayscale, so the only button that should do anything when the user clicks on it is the Gray button.

To implement clicking on a button, think about how you decided if the user pressed the mouse button on a hot air balloon in last week's lab and do something similar this week to determine which button a user has clicked on, if any.

When the user clicks on the button labeled Gray, the image should change to grayscale.

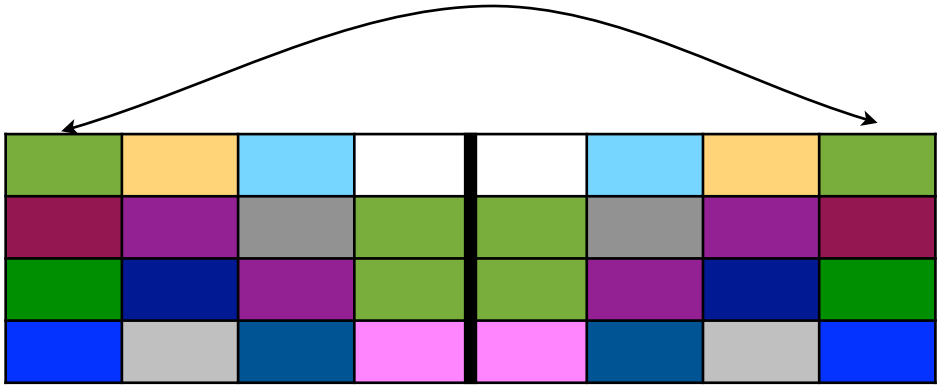
Step 5: Restoring the original

To restore the original image, you must remove the existing image from the display, reload the original image from the file and create a visible image. This should happen when the user clicks on the Original button.

Step 6: Mirroring the image

To mirror an image, you are swapping pixels on the right side of the image with those on the left. For example, a pixel that is 5 pixels from the left edge before swapping should be 5 pixels from the right edge after swapping.

If your image is 4 pixels wide and 4 pixels tall, like shown below, where should the pixel at row 0, column 0 be in the mirrored image? What about the pixel at row 2, column 2? In general, how can you calculate the location of a pixel in the mirrored image? (Hint: it depends on the location in the original image and the number of columns of pixels in the image.)



Mirroring an image is a little more complicated to write than producing grayscale. The reason is that the pixel that you are changing is in a different location than its original location. As a result, you will find it handy to have a blank picture that you can copy the mirror pixels into. Without a blank picture to copy pixels to, you may end up creating a symmetric picture containing just one half mirrored, looking something like this:

This is kind of cool or disturbing, actually, and **not what we are looking for!**

To perform this operation, you use two Picture objects. Picture provides a second constructor that creates a blank picture of a specific size. The signature of this constructor is:

```
public Picture (int imgWidth, int imgHeight)
```

One of your Picture objects will hold the image before the swap, the other will hold the image after the swap.

```
create a blank Picture that is the same size as the original
for each pixel in the first picture {
   determine the location of the corresponding pixel in the mirrored
     picture
   set the color of the pixel in the mirrored picture
```

```
}
remove the current image from the display
display the mirrored picture
```

Extra credit: Blurring the image

To blur an image, we randomly swap a pixel with a nearby pixel. Pick a maximum pixel distance, call it BLUR_DISTANCE, to use. (The demo uses 2.) Then for each pixel, swap it with another randomly chosen pixel that is within BLUR_DISTANCE pixels to the left, right, above and below. To keep your life simple, leave the pixels within BLUR_DISTANCE of the border unchanged.

As with mirroring, you will need to have 2 Picture objects. One will contain the original image while the other will contain the updated image.

Style Issues

Method design

It is good style to keep any individual method from becoming too long. If you are not careful, you will find that your onMouseClick method will be very long. It is better style for your onMouseClick method to be an if-statement that determines which button was clicked. Within each branch of the if-statement, you should call a private method to carry out the actions associated with that button. For example, one branch of your if-statement should look something like this:

```
if (grayButton.contains (point)) {
    grayscale();
}
```

Then, you should have separately defined a private method with a signature like this:

```
private void grayscale()
```

This method will contain the nested loops necessary to create the grayscale image and display it.

You should also keep your eye out for places where you repeat the same code several times, such as displaying the modified image. It is good to put that repeated code into a method and call the method from several places instead of repeating it.

Grading

2	Pre-lab work
10	Step 1: Displaying an image
10	Step 2: Grayscale
10	Step 3: Displaying the buttons
5	Step 4: Clicking on a button
5	Step 5: Restoring the original
10	Step 6: Mirroring the image
10	Compiles without error
5	Comments

4	Constant declarations
4	Indentation
5	Variable names
5	Variable scopes
5	If statements and conditions
5	While loops
5	Method design
100	**Total**
5	**Extra credit: Blurring the image**

Turning in Your Work

Your work will be automatically collected from your dev/cs101/Lab4 folder at the time that it is due. Please be sure your files are in the right place.

Over the course of the semester, you may have up to 5 late days total. Use them wisely! To request a late day, fill out the Google form on the course website so that we do not collect your assignment when it is due.