CS100J Fall 2005 Assignment A5. Due 4 November

1. Introduction

This is the first of two assignments that deal with .gif and .jpg images. You will learn about the RGB color system. You will learn something about how such images are stored, and you will write code to invert, transpose, and reflect such images. You will learn also a bit about how GUIS (Graphical User Interfaces) are constructed in Java. This assignment will give you practice with loops as well as one and two-dimensional arrays.

Here are the classes that you will need —download them from the class website and put them into a new folder: `ImageJFrame`, `ImagePanel`, `ImageMap`, and `ImageMaintainer`. The website also has a few images that you can download and use; put them in the same folder. To get an idea what the program does, do this:

(1) Open file `ImageJFrame` in DrJava.

(2) In the interactions pane, type this: `j= new ImageJFrame("first pic");` A dialog window will open. Navigate to a folder that contains a jpg or gif file and select it. A window will open, with the image in it.

(3) In the interactions pane, type this: `j.invert();` The picture should be inverted —it will look like a negative. Execute the method call again to get back the original image.

2. The methods you will write.

You will write three methods: to transpose an image, to reflect an image around its horizontal middle, and to reflect an image around its vertical middle. Below we show you what this means in terms of integer arrays. Below is an array. Then comes its transpose —put simply, each row `k` of the original array becomes column `k` in its transpose. Then comes its reflection about its horizontal axis. Here, if the rows are numbered `0..r-1`, row `k` of the original is row `r-1-k` in its horizontal reflection. Finally, if the columns are numbered `0..c-1`, then column `h` of the original is column `c-1-h` of its reflection around its vertical middle.

array	**transpose**	**hreflect**	**vreflect**
01 02 03 04	01 06 11	11 12 13 14	04 03 02 01
06 07 08 09	02 07 12	06 07 08 09	09 08 07 06
11 12 13 14	03 08 13	01 02 03 04	14 13 12 11
	04 09 14		

These three algorithms are fairly easy to write in terms of two-dimensional arrays. However, the algorithms will be a bit more complicated when the arrays involved are arrays of pixels making up an image. Most of this document is devoted to explain the Java classes you need to play with images.

We suggest that, before writing the code to manipulate the images, you write three static functions to produce the transpose, hreflect, and vreflect of a two dimensional integer array `b[0..r-1, 0..c-1]`, as well as a method to print (in the interaction pane) a rectangular array in order to help you debug your work. This practice makes code writing easier —you will then simply translate your code into the image framework.

To see exactly what you will do, open class `ImageMaintainer` and look at the four procedures `invert` (which is written for you and can be taken as a model for what you will write), `transpose`, `hreflect`, and `vreflect`. You have to complete the last three procedures. Write the reflection procedures first because they are easier.

3. Class ImageJFrame

You know that a `JFrame` is a window on your monitor. In this assignment, we want a window that contains an image —the window is an image. Therefore, we write a class `ImageFrame` to extend class `JFrame`. Take a look at these components of class `ImageFrame` (there are others, which you need not look at now).

Field `panel` is the variable that actually contains the image —you'll see this later.

Constructors: There are two constructors. One is given an image and a title for the window. The other is given only the title, and it gets the image using a dialog with the user; the user can navigate on their hard drive and choose

which image to work with. You don't have to know how to do this, but it is similar to obtaining a file to read, which you will learn about in a Lab.

Method `setUp`. Both constructors call this private method. First, it creates the panel with the image in it. Second, it adds the panel to the `JFrame` —this is what the call `add (BorderLayout.CENTER, panel);` does. One can add many things to the `JFrame` —buttons, labels, text fields into which a user can type information, etc. For more information on placing components in a `JFrame`, turn to Chapter 17 of the text. Listen to the appropriate lectures on the *ProgramLive* CD; that is the most efficient and enjoyable way to learn about GUIs.

The call of method `setDefaultCloseOperation` in `setUp` fixes the small buttons in the `JFrame` so that clicking the close button causes the window to disappear and the program to terminate. The next statement indicates where to place the window when it appears, and the call on `placeImage` does the necessary steps to get the image in the window. You don't have to look at `placeImage` now.

Methods to manipulate the image. At the bottom of the class are five methods to transpose, invert, reflect, and restore the image. They are placed here to make it easy to perform these functions in the interactions pane. They work by calling methods in another class to perform the operations. We have written the code to invert and to restore the image; the others are left for you to do.

4. Class Image and class ImageMap

An instance of class `Image` can contain a jpg or gif image (or some other formats as well). Just how the image is stored is not our concern; the class hides such details from us. Abstractly, however, the image consists of a rectangular array of pixels (picture elements), where each pixel entry is an integer that describes the color of the pixel. We show a 3-by-4 array below, with 3 rows and 4 columns, where each `Eij` is a pixel.

```
E11  E12  E13  E14
E21  E22  E23  E24
E31  E32  E33  E34
```

An image with `r` rows and `c` columns can be placed in an `int[][]` array `b[0..r-1][0..c-1]`. However, class `Image` provides us with something slightly different; it gives us the pixels in a one-dimensional array `map[0..r*c-1]`. For the 3-by-4 image shown above, array `map` would contain this:

```
E11, E12, E13, E21, E22, E23, E31, E32, E33
```

Thus, row 1 is first, then row 2, etc. This ordering of the array elements is called *row-major order.*

Our class `ImageMap` provides methods for dealing with array `map`. You can change the image by calling its methods `getPixel`, `setPixel`, and `SwapPixels`. So, for a variable `im` of class `ImageMap`, instead of writing something like `im[h, k]= p`; to set an element of an image to `p`, we write `image.setPixel(h, k, p);`. That's really all you need to know to translate your three methods into methods that change an image.

It might help to have a bit of understanding of this class. We provide a discussion here. Note that the class has a field `map`. The constructor has this in it:

```
map= new int[r*c];    // Create the array to contain the pixel-map
PixelGrabber pg=  new PixelGrabber(im, 0, 0, c, r, map, 0, c);
pg.grabPixels();
```

This statement sequence stores in `pg` an instance of class `PixelGrabber` that has associated image `im` with our array `map`. The third statement actually stores the pixels of the image in our array `map`. (Do not worry at this point about the try- and catch- thingamabobs; we'll explain them later in the course.)

Once an `ImageMap` is created for an `Image`, methods `ImageMap.setPixel`, `ImageMap.getPixels`, and `ImageMap.SwapPixels` can be used to manipulate the image.

5. Pixels and the RGB system

In maintaining images electronically, several different color systems are used. For example, most printers rely on the CYMK system, which uses the colors **C**yan, **Y**ellow, **M**agenta, and **B**lack. Black is needed because the best one

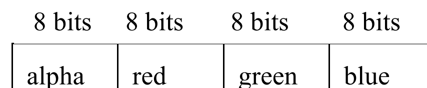
can do without it, in printing, is a muddy-looking brown. The system is not perfect. Only about 1 million colors are supported by the CYMK system, but it's a good system for printing.

Your monitor uses the RGB (Red Green Blue) system, and most images are stored using this system, too. The red component is given by a number in the range 0..255 (which takes 8 bits); green and blue components have the same range. Black is represented by (0, 0, 0), red by (255, 0, 0), green by (0, 255, 0), blue by (0, 0, 255), and white by (255, 255, 255). The number of different colors in the RGB system is 2**24 =16,777,216.

The assignment page of the CS100J website has a java program that you can download and run in DrJava in order to play with RGB colors. Do so to get a better understanding of the RGB system.

A pixel is stored in a 32-bit word. The red, green, and blue components take 24 of those bits. The last 8 bits are used for the "alpha channel", which is used as a mask to make certain areas of the image transparent —in those software application that use it. We will not change the alpha channel of a pixel in this assignment.

The elements of a pixel are stored in a 32-bit word like this:



Usually, 8-bit values are represented in the hexadecimal number system, which uses the 16 signs 0, 1, …, 9, A, B, C, D, E, F. For example, here are hexadecimal representations of some integers. 15 is 0F; 16 is 10, 31 is 1F, 254 is FE, 255 is FF. Hexadecimal F is 1111 in binary, and E is 1110.

Suppose we have the green component (in binary) `g = 01101111` and a blue component `b = 00000111`, and suppose we want to put them next to each other in a single integer, so that it looks like this in binary:

```
0110111100000111
```

This number can be computed using `g*2**8 + b`. But to calculate it that way is inefficient. Java has an instruction that *shifts* bits to the left, filling the vacated spots with `0`'s. We give three examples of this below, using 16-bit binary numbers.

```
0000000001101111 << 1   is
0000000011011110

0000000001101111 << 2   is
0000000110111100

0000000001101111 << 8   is
0110111100000000
```

Secondly, operation | can be used to "or" individual bits together:

```
0110111100000000 |
0000000010111110     is
0110111110111110

0011 |
1010     is
1001
```

Therefore, we can put an alpha component `alpha` and `red-green-blue` components `red`, `green`, and `blue` together into a single 32-bit **int** value —a pixel— using this expression:

```
(alpha << 24) | (red << 16) | (green << 8) | blue
```

Take a look at method `ImageMaintainer.invert`, which we have provided. For each pixel `(i, j)`, the method extracts the 4 components of the pixel, inverts the red, green, and blue components (e.g. the inversion of `red` is `255 – red`), reconstructs the pixel using the above formula, and stores the new pixel back in the image.

6. Class ImagePanel

We said earlier that a `JPanel` is a component that can be placed in a `JFrame`. We want a `JPanel` that will contain one Image. So, we define a class `ImagePanel` that extends class `JPanel`. We now discuss class ImagePanel. We suggest you read this section with class `ImagePanel` open in DrJava.

Class `ImagePanel` has two fields: one contains (the name of) the image object; the other contains (the name of) the `JFrame` object in which the `ImagePanel` object is placed. You can see how the constructor places values in these fields and also sets the "preferred size" of the `ImagePanel` to the dimensions of the image —this preferred size is used by the system to determine the size of the `JFrame` when it is shown.

Method `paint` is important. Whenever the system finds it necessary to redraw the panel (perhaps it was covered up and is now no longer covered up), the system calls method `paint`. The method calls a method `drawImage` of the graphics to draw the image.

Finally, method `changeImage` is called by our own program whenever it determines that the image has been changed. For example, after transposing or inverting the image, this method is called. If the new image is null, the method saves the new image and hides the window and returns. If not null, the method resets the preferred size and asks the `jframe` on which the panel has been placed to resize everything.

How does one learn to write all this code properly? Most people, when faced with doing something like this, will start with other programs that do something similar and modify them to fit their needs (as we did).

7. Class ImageMaintainer

Class `ImagePanel` has the basics for placing an image in a panel. However, there is more to do in maintaining an image —it should be capable of being transposed, inverted, etc. Class `ImageMaintainer`, which extends `ImagePanel`, provides these additional capabilities.

The class has three fields, which (1) contain the name of the file from which the image came, (2) the original `ImageMap` for the image, and (3) the current `imageMap` for the image.

As the image is manipulated, field `imageMap` changes. It can be restored to its original by copying field `originalMap` into `imageMap`. See procedure `restore` at the end of the class.

The constructor simply places its parameters in the fields.

Whenever field `imageMap` is changed, a new Image must be formed from it and the change must be reflected in the panel that is in the `JFrame`. This is accomplished by a call on procedure `formImage`. Take a look at it, but you need not understand its body completely.

Procedure `invert` is completed for you. It inverts the image, as discussed above. Look at its code to see how it processes each pixel —first retrieving it, then changing its parts, and finally placing the changed pixel back into `imageMap`.

Your job is to complete the bodies of procedures `transpose`, `hreflect`, and `vreflect`.

8. Your task.

Please complete the bodies of procedures `hreflect`, `vreflect`, and `transpose`. Do them in that order, because `transpose` is the hardest. Do not change anything else —do not declare new fields in the class and do not change any of the other classes. Submit your completed file ImageMaintainer.java on the CMS by the due date, which is 4 November.