CS100J Fall 2005 Assignment A6. Due 22 April

1. Introduction

This assignment deals with .gif and .jpg images. You will learn about the RGB color system and how images are stored; write code to transpose images; learn about filtering images; and see how an image can be broken up into blocks. You will learn a bit about how GUIS (Graphical User Interfaces) are constructed in Java. Finally, you will have practice with loops and one- and two-dimensional arrays.

Download these course website and put them into a new folder: EITHER file a6.zip OR the files indicated on the website. Two images are included. Put everything in the same folder. To get an idea what the program does, do this:

(1) Open file `ImageJFrame` in DrJava and compile it.

(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 and several buttons.

(3) See what buttons `invert`, `horiz reflect`, `vert reflect` do. After any series of clicks on these, you can always click button `restore` to get back the original file.

(4) You can try the other buttons (`transpose`, `filter`, and the space buttons) but they won't work —until you write code to make them work.

We will now spend some space explaining a little bit about the classes in this assignment and about images. You don't have to learn all this by heart, of course. In section 7, we explain what you have to do for this assignment.

2. 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 could be placed in an **`int`**`[][]` array `b[0..r-1][0..c-1]`. However, class `Image` provides us with something 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`, write `image.setPixel(h, k, p)`; . That's all you need to know to manipulate images in this assignment.

It might help to have a bit of understanding of class `ImageMap`. 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.

3. Pixels and the RGB system

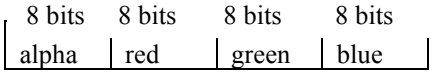
In maintaining images electronically, several different color systems are used. For example, most printers rely on the CYMK system, which uses the colors **C**yan, **Y**ellow, **M**agenta, and **B**lack. (The black is referred to as K for **key** – a shorthand for the printing term **key plate**.) Black 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.

Open class `ColorChooserExample` and execute `ColorChooserExample.main(null);` in order to play with RGB colors in order to get a better understanding of the RGB system.

A pixel is stored in a 32-bit (4 byte) word. The red, green, and blue components each take 8 bits. The remaining 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 applications 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:



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      0000000001101111 << 2  is
0000000011011110                0000000110111100


0000000001101111 << 8   is
0110111100000000
```

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

```
0110111100000000 |                    0011 |
0000000010111110      is               1010     is
0110111110111110                       1011
```

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`. 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.

4. Class ImageJFrame

You know that a `JFrame` is a window on your monitor. In this assignment, we want a window that contains an image. Therefore, we write a class `ImageJFrame` to extend class `JFrame`. Take a look at these components of class `ImageJFrame` (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. This is similar to obtaining a file to read, which you learned 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. The rest of this method has to do with adding the buttons to the panel. You don't have to look at this stuff now; we'll explain it a bit in later lectures. For more information on placing components in a `JFrame`, turn to Chapter 17 of the text.. The most efficient and enjoyable way to learn about GUIs is to listen to the appropriate lectures on the *ProgramLive* CD.

The call of method `setDefaultCloseOperation` near the end of `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.

Methods to manipulate the image. At the bottom of the class are methods to (1) restore, invert, and transpose the image; (2) to set a block of an image to a pixel value; (3) to swap two blocks of an image; and to filter an image. They are placed here to make it easy to perform these functions in the interactions pane. They work by calling methods in class `ImageMaintainer`. You have to write four of these methods.

Methods to make the buttons available to the program. A set of methods are used to connect the clicking of a button on the window to the program. You don't have to look at these. We'll give some idea on how they work later.

5. Class ImagePanel

We have said 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).

6. 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 contain (1) 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`. That's what procedure `restore`, which is near the end of the class, does.

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.

Procedures `invert`, `hreflect`, `vreflect`, and `restore` are completed. Procedure `invert` 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`.

7. The methods you will write.

Implement the methods in class `ImageMaintainer` that are explained in this and the next section.

7A. Implement method `transpose`. Below is an array. To its right is its transpose —put simply, each row `k` of the original array becomes column `k` in its transpose.

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

The transpose algorithm is 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.

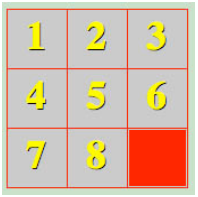
We suggest that, before writing the code to manipulate the images, you write a static function to produce the transpose of a two dimensional integer array `b[0..r-1, 0..c-1]`, as well as a procedure to print (in the interaction pane) a rectangular array in order to help you debug your work. This practice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

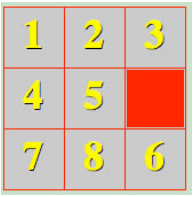
Use procedure `ImageMaintainer.invert` as a model for what you will write in procedure `transpose`.

7B. Implement method `filter`. In this method, you change each pixel of the image to consist only of grey, red, green, or blue color, depending on the value of parameter color. Producing a "grey-scale image" is done by making all three components —red, green, blue— equal to the average of the original red, blue, and green values. Filtering it through red (and similarly through green and blue) is done simply by making the green and blue components 0.

This manipulation requires that you extract the alpha, red, green, and blue components from each pixel and then construct the new pixel value and store it. Look at procedure `invert` to see how this is done.

7C. Implement methods `setBlock` and `swapBlocks`. You know that old fifteen shuffle puzzle. One is given a rectangle of blocks with one block empty (in this case, red), as on the right. One can shift one of the adjacent blocks to the empty position, as on the second picture to the right, where block 6 has been shifted down into the empty position. The idea is to start with a random situation and shift blocks into the empty space until a picture emerges.





Take a look at class `BlockGame`. It extends `ImageJFrame`, because a block game is an image JFrame with additional methods.

Suppose you have finished methods setsBlock and swapBlocks. Then, executing the following statement in the interactions pane causes a navigation window to open; you can open any image you wish. Its top left block will be empty.

```
shuffle= new BlockGame("shuffle game",3,3);
```

Then, executing these method calls in the interactions pane will cause blocks to be shifted as indicated (if the move is legal):

```
shuffle.move(0);  // shift a block upward into the empty space.
shuffle.move(1);  // shift a block downward into the empty space.
shuffle.move(2);  // shift a block left into the empty space.
shuffle.move(3);  // shift a block right into the empty space.
```

Please write the bodies of methods `ImageMaintainer.setBlock` and `ImageMaintainer.swapBlocks`. The important piece of work will be to figure out what pixels are in a block with top left corner `(row0, col0)`. This is based on the number of vertical and horizontal blocks there are in the image, as given by parameters `nr` and `nc`.

Note that if the number of rows `r` (say) in an image is not divisible by `nr`, then `r % nr` rows at the bottom of the image are not considered part of a block and will never be changed. Similarly for the columns. That is OK.

File `scramble1.jpg` contains an image that already has been scrambled as 3 x 3 block. When you are finished with the two procedures for this part, see whether you can use your program to unscramble the picture.

8. Your task.

Start early, because you are sure to have questions! Waiting until the deadline will only cause frustration and lack of understanding. Complete the bodies of procedures `transpose, filter, setBlock, and swapBlocks` in class `ImageMaintainer`. 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, 22 April.