CS100J Fall 2007 Assignment A7. Due Tuesday, 20 November

0. Introduction

This assignment deals with .jpg images. You will learn about how images are stored; write code to transpose images; and learn about filtering images. 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 *either* file a7.zip OR the files indicated on the course website and put them into a new folder. Two images are included. Put everything in the same folder. To get an idea of 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();` A dialog window will open. Navigate to a folder that contains a jpg file and select it. A window will open, with two versions of the image, some buttons, and a text area. The left image will not change; it is the original image. The right image will change as you click buttons.

(3) See what buttons `invert` and `horiz 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 buttons `transpose` and `filter` but they won't work until you write code to make them work.

We now explain a little bit about the classes in this assignment and about images. You don't have to learn all this by heart, but you would do well to study the code, being conscious of how precise the specs are and how well the Java code is written. Section 7 explains what you have to do for this assignment.

1. Separation of concerns

It is important to organize the parts of a program in a logical coherent way, in which it is clear what each part is responsible for and in which interactions are kept reasonable. The larger and more complicated the task of the program, the more important it is to have a good organization.

This program has two main functions: manipulating an image and providing a GUI. These two issues should be separated as much as possible in the program.

An object of class `Image`, in an API package, maintains an image. Our own class `ImageMap` provides more useable methods for manipulating an image, allowing one to view it as a two-dimensional table or as a one-dimensional array of pixels, kept in row-major order (the elements of row 0, then the elements of row 2, then the elements of row 3, etc.). It has a field that contains the `Image` itself.

Class `ImageMaintainer` provides methods for manipulating the image, maintained as an ImageMap. `ImageMaintainer` knows nothing about the GUI; it simply calculates.

Class `ImagePanel` provides part of the GUI. A subclass of `JPanel`, it can display an image through its method `paint`. When an image is changed in any way, the corresponding `JPanel` object has to be notified so that it can revise the size of the panel; that is the purpose of method `formImage`.

Class `ImageFrame` provides the GUI. It places buttons and `ImagePanels` in the window, and it "listens" to button clicks and acts accordingly, calling appropriate methods in `ImageMaintainer`, then calling on an `ImagePanel` to revise its view, and then repainting the GUI.

2. Class Image and class ImageMap

An instance of class `Image` can contain a jpg 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, 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 the elements in row-major order:

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

Class `ImageMap` provides an internal representation of an image —array `map`— along with methods for dealing with it. You can change the image by calling its methods `getPixel(row,col)`, `setPixel (row,col,v)`, and `SwapPixels(a,b,i,j)`. So, for a variable `im` of class `ImageMap`, instead of writing something like `im[h, k]= v`; to set an element of an image to `p`, write `image.setPixel(h, k, v)`; . Note also that you can reference pixels in row-major order in a one-dimensional array. For example, there are methods `getPixel(p)` and `setPixel(p,v)`. That's all you need to know to manipulate images in this assignment.

Here's more info on class `ImageMap`. 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 stores the pixels of the image in array `map`.

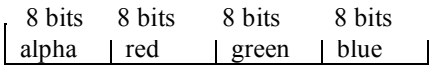
Once an `ImageMap` is created for an `Image`, methods `ImageMap.setPixel`, `ImageMap.getPixels`, and `ImageMap.SwapPixels` can be used to manipulate the image.

3. Pixels and the RGB system

Your monitor uses the RGB (Red-Green-Blue) system for images. Each RGB component is given by a number in the range 0..255 (8 bits). 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 RGB colors is 2^{24} =16,777,216.

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 this calculation is inefficient. Java has an instruction that *shifts* bits to the left, filling the vacated spots with `0`'s. We give three examples, 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 |                    0011 |
        0000000010111110                      1010
is      0110111110111110              is      1011
```

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

```
(alpha << 24) | (r << 16) | (g << 8) | b
```

Take a look at method `ImageMaintainer.invert`. For each pixel, 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 ImagePanel

We suggest you read this section with class `ImagePanel` open in DrJava. 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`.

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 size and "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 wants to redraw the panel (perhaps it was covered and is now no longer covered), the system calls method `paint`, which calls `drawImage` of the graphics to draw the image.

Finally, method `formImage` is called by our own program whenever it determines that the image has been changed, e.g. after inverting the image. The method gets the image from ImageMap map and resets the size and preferred size.

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

5. Class ImageJFrame

A `JFrame` is a window on your monitor. In this assignment, we want a window that contains two versions of an image. Therefore, class `ImageJFrame` extends `JFrame`. Take a look at these components of class `ImageJFrame` (there are others, which you need not look at now).

Fields original`Panel` and currentPanel contain the panels for the original and manipulated images.

Constructors: There are two constructors. One is given an image. The other has no parameters; 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. The method puts the buttons into the JFrame —we'll learn about this later. It then adds a labeled text area, which you will use in assignment A8. Then, provided there is an image, it creates two panels with the image in it and adds them to the `JFrame`, using the call `add(BorderLayout.CENTER, imagebox);`. It creates an instance of class `ImageMaintainer`, which will contain methods to manipulate the object. Finally, it fixes the location makes the JFrame visible, and "packs" and repaints it.

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.

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 lectures on the *ProgramLive* CD.

Methods to manipulate the image. At the bottom of the class are methods to (1) restore, invert, and transpose the image; (2) to filter an image, and (3) to hide and retrieve a message in the image (you do not have complete these!). 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 several 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.

6. Class ImageMaintainer

Class `ImageMaintainer` provides all the methods for manipulating the image given to it in the constructor, as an `ImageMap`. The constructor stores the map in field `originalMap` and stores a *copy* of it in `currentMap`.

As the image is manipulated, object `currentMap` changes. It can be restored to its original by copying field `originalMap` into `currentMap`. That's what procedure `restore` (near the end of the class) does.

Procedures `invert`, `hreflect`, and `restore` are complete. Procedure `invert` inverts the image (makes a negative out of a positive, and vice versa). 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` as explained below.

7A. Implement method vreflect. This method should reflect the image around its vertical middle. Look at method `hreflect` to get an idea about how to do it.

In order to get this method to work from the GUI, you have to change the GUI, as follows. Look at function `ImageJFrame.setup`. (1) After the addition of button `buttonhreflect` to `buttonBox`, insert a statement to add button `buttonvreflect` to `buttonBox` (the button is already declared). (2) Take a look at the code that makes **`this`** (i.e. this object) a listener to the various buttons. Insert a statement to add **`this`** as a listener to `buttonvreflect`.

7B. Implement method `transpose`. Below is an array. To its right is its transpose: 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 `ImageMap` framework.

The comments in the body of `transpose` should help you write the body. In addition, use procedure `ImageMaintainer.hreflect` as a model for accessing the elements of an `ImageMap`.

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

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

8. Your task.

Start early, because you are sure to have questions! Waiting until the deadline will cause you frustration and lack of understanding, instead of the fun that should be felt in completing this assignment. Complete the bodies of procedures `vreflect`, `transpose`, and `filter` in class `ImageMaintainer` and change class `ImageJFrame` as discussed in task 7A. Do not change anything else —do not declare new fields in the class and do not change any of the other classes. You may, of course, add other methods to class `ImageMaintainer`. Submit your completed file `ImageMaintainer.java` and `ImageJFrame` on the CMS by the due date.