CS1110 Fall 2010 Assignment A6. Images. Due Thurs Nov 18, 11:59pm on the CMS

0. Introduction

Please read this **entire** document carefully. Budget your time wisely. Don't wait until the day before the deadline to do this assignment. We advise starting now and doing a little work every day (the practice with loops and invariants will help you on the prelim).

This assignment deals with .jpg images. You will

1. Learn how images are stored
2. Write methods to manipulate images, getting practice with for-loops and 1- and 2-dimensional arrays
3. Write methods to "undetectably" hide and reveal a secret text message in an image file
4. Learn a little bit about constructing GUIS (Graphical User Interfaces) in Java
5. Get more practice reading and understanding previously-written code
6. Get practice debugging more involved code
7. Give you more practice with stepwise refinement, such as through writing your own helper methods
8. Shows an example of using constants (final variables) to represent "magic numbers" in a more mnemonic fashion

Download *either* file a6.zip *or* the other files for A6 given on the course website and put them into a new folder. Several images are included. Put everything in the same folder. To get an idea of what the program does, do this:

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

(2) In the Interactions pane, type this: `j= new ImageGUI();` 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`, `transpose`, and `hor reflect` do. After any series of clicks on these, you can always click button **`restore`** to get back the original file.

(4) You can use the `save image` button to save the modified pictures on your hard drive.

(5) You can try buttons `ver reflect`, `fuzzify`, `put In Jail`, `filter out`, and any other buttons, but they won't work unless you write code to make them work.

We discuss the classes in this handout. 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 the Java code is written. Section 7 explains what you have to do for this assignment, and Section 8 explains what you have to turn in.

Grouping. You may work with one other student. If you do, group yourselves on the CMS at least 1 week before the due date. You are expected to do the work together. Separating the work into two parts, doing the parts independently, and then merging the work is a violation of academic integrity. Any writing of Java code should be done while sitting together, with one person typing and the other person helping. Take turns doing the typing.

Academic Integrity. We expect the work you submit to be yours (or your group's) alone. It is a violation of academic integrity to be in possession of or to read the code produced by someone else, either in this class or in previous classes, or to show your code (in any format) to other students in the class. Please don't do that. If we find someone has done this, we will prosecute according to Cornell's Code of Academic Integrity.

1. Separation of concerns

It is important to organize the parts of a program in a logical and coherent way such that it is clear what each part is responsible for and such that interactions between parts are kept reasonable. The larger and more complicated the task of a program, the more important it is to have 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 `java.awt.Image` (note that this is an abstract class, so such objects always have real type that is a subclass of `java.awt.Image`) maintains an image. Our own class `ImageArray` maintains an image as a one-dimensional array of pixels, storing the pixels of the conceptually two-dimensional array in *row-major order*, i.e. first the elements of row 0, then the elements of row 1, then the elements of row 2, etc. `ImageArray` provides methods for manipulating the image, allowing a user to process the pixels of an image row by row, column by column, or without regard to the order.

Class `ImageProcessor` provides methods for transforming the image, maintained as an `ImageArray`. `ImageProcessor` knows nothing about the GUI; it simply calculates. It has fields that contain (the name of) the original and the transformed `ImageArray`.

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 to revise the panel; updating the image and providing this notification are done by method `changeImageTo`.

Class `ImageGUI` provides the GUI. It places buttons and `ImagePanel`s in the window, and it "listens" to button clicks and acts accordingly, calling appropriate methods in `ImageProcessor`, then calling on an `ImagePanel` to revise its image, and finally repainting the GUI.

2. Class Image and class ImageArray

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.

```
E00  E01  E02  E03
E10  E11  E12  E13
E20  E21  E22  E23
```

An image with `r` rows and `c` columns could be placed in an **`int`**`[][]` array `b[0..r-1][0..c-1]`. Instead, however, class `ImageArray` maintains the pixels in a one-dimensional array `rmoArray[0..r*c-1]`. For the 3-by-4 image shown above, array `rmoArray` would contain the elements in row-major order:

```
E00, E01, E02, E03, E10, E11, E12, E13, E20, …
```

Class `ImageArray` provides the representation of an image in its array `rmoArray`, along with methods for dealing with it. You can get the value of an individual pixel with `getPixel(row,col)`. You can change the image by calling its methods `setPixel(row, col, v)`, and `SwapPixels(a,b,i,j)`. So, for a variable **`im`** of class `ImageArray`, to set a pixel to `v`, instead of writing something like `im[h,k]= v;` write `im.setPixel(h,k,v);`. You can also reference pixels in row-major order in a one-dimensional array, using methods `getPixel(p)` and `setPixel(p,v)`. That's all you need to know in order to manipulate images in this assignment.

Here's more info on class `ImageArray`. The first constructor has these statements in it:

```
rmoArray= new int[r*c];
…
PixelGrabber pg= new PixelGrabber(im,0,0,c,r,rmoArray,0,c);
…
pg.grabPixels();
```

This code stores in `pg` an instance of class `PixelGrabber` that has associated image **`im`** with our array `rmoArray`. The third statement, `pg.grabPixels();`, stores the pixels of the image in `rmoArray`.

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 (memory location). 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:

8 bits	8 bits	8 bits	8 bits
alpha	red	green	blue

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, 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 `ImageProcessor.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

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 make `ImagePanel` extend `JPanel`.

Field `image` of `ImagePanel` contains (the name of) the image object. The constructor places a value in `image` 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 laying out the window.

Method `paint` is called whenever the system wants to redraw the panel (perhaps it was covered and is now no longer covered); our method `paint` calls `g.drawImage` to draw the image.

Finally, method `changeImageTo` is called whenever our program determines that the image has been changed, e.g. after inverting the image. Take a look at the method body.

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 ImageGUI

A `JFrame` is associated with a window on your monitor. Since we want a window (that contains two versions of an image), class `ImageGUI` extends `JFrame`. Take a look at the following components of class `ImageGUI` (there are others, which you need not look at now).

Fields `originalPanel` 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, where 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 learn about in a lab.

Method setUp. Both constructors call this private method. The method puts the buttons into the `JFrame` (using two arrays and a loop to streamline some of the repetitive stuff)—we'll learn about this later. It then adds a labeled text area, which you will use later. Then, provided there is an image, it creates two panels with the image in them and adds them to the `JFrame`, using the call `add (BorderLayout.EAST, imagebox);`. It creates an instance of class `ImageProcessor`, which will contain methods to manipulate the object. Finally, it fixes the window 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.

Read Chapter 17 of the text for more information on placing components in a `JFrame`. The most efficient and enjoyable way to learn about GUIs is to listen to lectures on the *ProgramLive* CD.

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 (although if you do, note the use of assert statements to make the programmer's job less prone to errors). We'll give some idea of how they work later.

6. Class ImageProcessor

Class `ImageProcessor` provides all the methods for manipulating the image given to it as an `ImageArray` in the constructor. The constructor stores the image in field `originalIm` and stores a *copy* of it in `currentIm`.

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

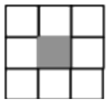
Procedures `invert`, `hreflect`, `transpose`, and `restore` are complete. Procedure `invert` inverts the image (makes a negative out of a positive, and vice versa). Note how it processes each pixel —retrieving it, then computing what its new color components should be, and finally placing the changed pixel back into `currentIm`.

7. The methods you will write. Implement the methods in class `ImageProcessor` as explained below.

But before you write any code, read the notes on writing, testing, and debugging on page 6.

*You **must** write invariants for every loop you write. Of course, in order for this to be most helpful to you, you should write each invariant **before** you write the corresponding loop.*

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.

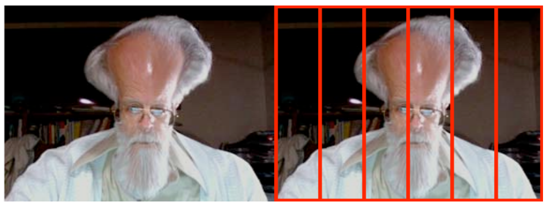


7B. Implement method `fuzzify`. This method replaces each pixel of the current image (except those on the edge) by the average of itself and its 8 neighbors —i.e. replace the red component by the average of the red components, and similarly for the blue and green components. *Don't* average the alpha compoment or otherwise change it. Follow the directions given in the java file. Look at method `invert` to see how to pick out the color components from a pixel and to build a new pixel.

As in transpose, the method will have two nested loops, and each iteration of the inner loop will process one pixel. Do not write all the code to process a pixel in the inner loop; instead, write another, private procedure that processes the pixel and, in the body of the inner loop, call the new procedure. This is done to keep each method simple and short. Also, you can initially stub in the private method (e.g., have it set the pixel to black) to facilitate constructing abd testing one bit of your code at a time. In a top-down, stepwise-refined fashion.

You must write a complete and thorough specification of any method you introduce. It is best to write it *before* you write the method body, because it helps *you* as you program. (It certainly helped us debug our own solutions!) This is standard practice in this course. It is a severe error not to do so, and points will be deducted for missing or inappropriate specifications.

When finished, open the eye-chart jpg file we give you; then hit the fuzzify button many times and watch the letters get fuzzier and fuzzier.



7C. Implement method putInJail. To the right you see the effect of clicking button *put in jail*: —drawing the red boundary and the vertical bars. This will happen when you implement method `putInJail` according to its specifications.

We have given you helper procedure drawHBar; use it. In the same way, implement a private procedure to draw a vertical bar. Don't forget its specification.

When finished, open a picture, click buttons *put in jail*, transpose, and then *put in jail* again for a nice effect.

7D. Implement method `filterOut`. In this method, you change each pixel of the image depending on the value of parameter `c`. You either filter out a single color (removing red, green, or blue), or remove all color by setting each of the three color components (red, green, blue) to the average of the original red, green, and blue values —this produces a gray-scale image. (Handy quick test: white pixels stay white and black pixels stay black.)

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.

In determining what parameter `c` is, do NOT use integer constants 0, 1, 2, and 3; instead use the static final fields `GRAY`, `RED`, `GREEN`, and `BLUE` (the names of variables holding constants are chosen to be all-capitals by convention). Use these mnemonic names rather than constants (the actual numbers), so that the program remains readable.

7E. Steganography, according to Wikipedia (en.wikipedia.org/wiki/Steganography), "is the art and science of writing hidden messages in such a way that no one apart from the intended recipient even realizes there is a hidden message." In contrast, in cryptography, the existence of the message is not disguised but the content is obscured. Quite often, steganography deals with messages hidden in pictures.

To hide a message, each character of the message is hidden in one or two pixels of an image by modifying the red, green, and blue values for the pixel(s) so slightly that the change is not noticeable to the eye.

Each character is represented using the American Standard Code for Information Interchange (ASCII) as a three-digit integer. We allow only characters that can be represented in ASCII —all the obvious characters you can type on your keyboard are ASCII characters. See page 6.5 of the ProgramLive CD for a discussion of ASCII.

For the normal letters, digits, and other keyboard characters like `$` and `@`, you can get its ASCII representation by casting the char to **`int`**. For example, `(`**`int`**`)'B'` evaluates to the integer 66, and `(`**`int`**`)'k'` evaluates to 107.

We can hide character `'k'` in a pixel whose RGB values are 199, 222, and 142 by changing each color component so that its least significant digit contains a digit of the integer representation 107 of `'k'`:

Original pixel				**Pixel with 'k' hidden**		
Red	Green	Blue	hide `'k'`, which is 107	Red	Green	Blue
199	222	142	$\longrightarrow$	19**1**	22**0**	14**7**

This change in each pixel is so slight that it will not —cannot— be noticed just by looking at the image.

Decoding the message, the reverse process, requires extracting the last digit of each color value of a pixel and forming the ASCII value of a character from the three extracted values. In the above diagram to the right, extract 1, 0, and 7 to form 107, and cast this integer to **`char`**.

Extracting the message does *not change the image*. The message stays in the image forever.

Three problems for you to solve. You will write code to store a message `m` in the pixels of an image in row-major order, starting with pixel 0, 1, 2, … Think about the following issues and solve them.

(1) You need some way to recognize that the image actually contains a message. Thus, you need to place something in pixels 0, 1, 2, … that has very little chance of appearing in a real image. You can't ever be *sure* that an image without a message doesn't start with those pixels, but the chances should be extremely small. This beginning marker should use at least two pixels.

(2) You have to know where the message ends. You can do this in several ways —place the length of the message in the first pixels in some way (how many pixels can that take?), put some unused character at the end of the message, or use some other scheme. You may assume that the message has fewer than one million characters.

(3) The largest value of a color component (e.g. blue) is 255. Suppose the blue component is 252 and you try to hide 107 in this pixel; the blue component should be changed to 257, but this impossible because a color components are ≤ 255. Think about this problem, come up with at least two ways to solve the problem, and implement one of them.

As you can see, this part of the assignment is less defined than the previous ones. *You* get to solve some little problems yourself. Part of this assignment will be to document and discuss your solutions.

Your task on part 7E

(a) Decide how you will solve the problems mentioned in points 1..3 above. As you design and implement this part, write a short essay that documents at least 2 solutions to each of the 3 problems mentioned above, discusses their advantages and disadvantages, and indicates what your solutions are. Advantages could be: easiest to implement, fewest number of pixels used in hiding an image, least time spent in hiding a message —whatever. When you are finished with this assignment, insert this essay as a comment at the beginning of class `ImageProcessor`.

Feel free to discuss points 1..3 with the course staff. They will not tell you *how* to solve these problems. But they will discuss your ideas with you.

(b) Complete the body of procedures `hide` and `reveal` in class `ImageProcessor`. These two methods will hide a message and reveal the message in the jpg image. When you design method `reveal`, make sure it attempts to extract the message only if its presence is detected. Feel free to introduce other methods as they are needed, but make them private because they are not to be called by a user, but only by your program.

Debugging hide and reveal can be difficult. We give you some hints on this at the end of this document.

8. What to submit

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`, `fuzzify`, `putInJail`, `filterOut`, `hide`, and `reveal` in class `ImageProcessor`. Don't change anything else (other than writing private helper methods that you write) —don't declare new fields in the class and don't change any of the other classes.

Insert your essay (see the beginning of part 7E) into file `ImageProcessor` and submit the file on the CMS.

9. Hints on writing, testing and debugging

1. For a method that consists of several steps, do not write the whole method and then compile and test. Generally you will have many syntax errors and you will waste time. Instead, compile after every line of code you write, and test what you have done if it is feasible to do so.
2. It will be difficult to use a JUnit testing class, because it will be difficult to get it to access a picture appropriately. You do not have to use one.
3. Use function `ImageArray.toString(pixel)` to get a readable String representation of a pixel.
4. In the methods that you write, use `System.out.println(…);` statements to print out information, so that you can see what your code is doing. This is the easiest way to determine whether your code is working correctly. But you will have to continually change these statements in order to keep the amount of output to something manageable. Before submitting the assignment, remove these statements as a courtesy to the graders.
5. Hide and reveal:
   a. We encourage writing helper methods to keep code simple and short in `hide` and `reveal`. You get to decide which ones to write as you design and implement this assignment. Be sure to specify each method appropriately in a comment before it.
   b. Do *not* assume that you can debug simply by calling `hide` and then `reveal` to see whether the message comes out correctly. Instead, debug method `hide` and then write and debug `reveal`.
   c. Start with extremely short messages to hide —1, 2, or 3 characters— and first check that the initial pixels, which are supposed to indicate that a message is hidden, are correct.
   d. When `hide` and `reveal` are done, try hiding and revealing a long message —1000, 2000, 5,000 characters. Notice any difference in the time to hide and the time to reveal? Can you figure out why? Can you change your code to make them the same (you don't have to do this). We'll discuss this in class at the appropriate time.