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 how a text message can be hidden in an image. You will see a little 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 a6.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 ImageGUI 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 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 try buttons transpose, ver reflect, and filter out, but they won’t work until you write code to make them work.

You can use procedure ImageGUI.writeImage to save the modified pictures on your hard drive.

We discuss the classes in this assignment and also 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, and Section 8 explains what you have to turn in.

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 maintains an image. Our own class ImageArray maintains an image as a one-dimensional array of pixels, storing the pixels of the real 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 so that it can revise the size of the panel; updating the image and providing this notification are the purposes of method changeImageTo.

Class ImageGUI provides the GUI. It places buttons and ImagePanels 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 view, 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 $E_{ij}$ 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 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 `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:

```java
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 to 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 $g \times 2^8 + b$, 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 \ll 1 is 0000000011011110
0000000001101111 \ll 2 is 0000000110111100
0000000001101111 \ll 8 is 0110111100000000
```

Secondly, operation $|$ can be used to “or” individual bits together:

```
0110111100000000 | 0011 | 0000000010111110
is 0110111110111110
```

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`. 

```java
8 bits     8 bits       8 bits        8 bits
| alpha | red   | green | blue |
```
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 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 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 — 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 manipulate the image. At the bottom of class ImageGUI are methods to (1) restore, invert, and transpose the image; (2) filter an image, (3) hide and retrieve a message in the image, and (4) write an image to your hard drive. They are placed here to make it easy to perform these functions using the Interactions pane. They work by calling methods in class ImageProcessor, several of which you have to write.

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 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, and restore are complete. Procedure invert inverts the image (makes a negative out of a positive, and vice versa). Notice how it processes each pixel — first 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.

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 transpose. Below is an array. To its right is its transpose: each row $k$ of the original array becomes column $k$ in its transpose.

<table>
<thead>
<tr>
<th>array</th>
<th>transpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 02 03 04</td>
<td>01 06 11</td>
</tr>
<tr>
<td>06 07 08 09</td>
<td>02 07 12</td>
</tr>
<tr>
<td>11 12 13 14</td>
<td>03 08 13</td>
</tr>
<tr>
<td></td>
<td>04 09 14</td>
</tr>
</tbody>
</table>
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. 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 Interactions 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 ImageArray framework.

The comments in the body of transpose should help you write the body. READ THEM CAREFULLY. In addition, use procedure ImageProcessor.hreflect as a model for accessing the elements of an ImageArray.

7C. 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. Use mnemonic names rather than constants, so that the program remains readable.

7D. 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.

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, \((\text{int}) \ 'B' \) evaluates to the integer 66, and \((\text{int}) \ 'k' \) 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 \( 'k' \):

<table>
<thead>
<tr>
<th>Original pixel</th>
<th>Pixel with ( 'k' ) hidden</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red  Green  Blue</td>
<td>hide ( 'k' ), which is 107</td>
</tr>
<tr>
<td>199     222     142</td>
<td>191     220     147</td>
</tr>
</tbody>
</table>

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 \( \leq 255 \). 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 7D

(a) Decide how you will solve the problems mentioned in points 1..3 given 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.

A note on static inner classes

In class a few times, we created a class simply to collect a few variables together as fields and to provide methods to manipulate them. Class Pair, which contains fields “hits” and “misses”, was one example of this.

If you feel the need to use such a class when writing part 7D, then declare it as a private static class at the end of class ImageProcessor — after the last method but before the “}” that ends the body of the class. Yes, you can declare “inner classes” this way, but make them static, so there is only one copy of it, and private, so that other classes can’t use it.

Please introduce such a class ONLY if necessary, and talk to an instructor or a TA before you do it. Part 7D can be done without it, and most people will not need this additional class.

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, transpose, filterOut, hide, and reveal in class ImageProcessor. Don’t change anything else (other than writing your private helper methods and, if necessary and after discussion, any static inner classes) — 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 7D) into file ImageProcessor and submit the file on the CMS.

Writing, testing and debugging hide and reveal

Methods hide and reveal can be difficult to debug. Here some hints to help you.

1. We encourage writing other methods (perhaps helper methods) besides 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.

2. It will be difficult to use a JUnit testing class here, 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. Do not assume that you can debug simply by calling hide and then reveal to see whether the message comes out correctly. The best thing to do is to write method hide and debug it before going on to reveal.

5. In the methods that you write, insert 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. When you submit the assignment, please remove these statements as a courtesy to the graders (who already have a lot of code to read).

6. 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.

7. Try hiding and revealing a long message — 1000, 2000, 5000 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.