Computer Laboratory 9

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday April 8th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

We've already explored computer-generated images a bit in previous labs. However, in those labs we looked at images generated from basic shapes. While this is a very natural way to think about images, it does not actually match the low-level truth of computer images. At a low level, computers represent images as a 2-d grid of colors.

You may already know this, or have some basic intuition, but to a computer, a 100x100 size picture, is simply a 100 by 100 grid of colored squares (called pixels, short for "picture elements"). If you zoom in enough on an image (or put your face really close to a low-resolution screen) you can even see the individual colored squares that make up the image. Normally, however, the images are big enough (and the pixels small enough) that you can't see them, and your eyes are fooled into seeing trees, coffee cups, cats, and every other image your computer may show you.

In this lab, we will be representing images using a custom class (provided) that stores images explicitly as 2d grids of colors. We will implement various image manipulations – simple transformations such as making the image into greyscale, or putting a border around the image. To make matters more interesting, however, these transformations will each be implemented as their own class. This is a software design pattern known as the "functional object" – in which a *class* is used to represent one or more *behavior* rather than simply representing data. In this case, we will take this pattern one step further – making each transformation class inherit from a parent class. This will allow us to make a polymorphic function that applies *several* transformations all at once.

From an educational standpoint, in this lab you will:

- Write code with a clear inheritance structure.
- See an example of a function that makes deliberate use of polymorphism.
- See code written in the "function object" design pattern – and see how this works well with advanced features like polymophism.

3 Software environment setup

This lab will be done using a java programming environment. We recommend the IntelliJ IDE running on your own personal computer. See lab 6 for instructions about how to set this up.

A few reminders:

- Source code goes in the `src` folder in java. Java is VERY particular about where files go, and what they are called.
- Lab06 has syntax guides if you still find translating to java difficult.
- Do not have any folders inside src – these are called packages and will make your code fail the autograder.
- Download *every* provided file and move them all to the src folder before starting.
- Code style elements like comments and javadocs are **important** and **easier to do as you go**. Don't write hard-to-read code and think "I can fix it later".

4 Files

This lab will involve the following **provided** files.

- `Transformation.java`
- `RGBImageUtil.java`
- `RGBImage.java`
- `RGBColor.java`
- `Tester.java`

This lab also involves the following files **you will create**:

- `Greyscale.java`
- `Stamp.java`
- `AddBorder.java`
- `ColorPallet.java`
- `Brighten.java`
- `TransformationUtils.java`

5 Instructions

Before beginning you should:

1. Setup an IntelliJ project
2. Download the provided source-files and place them in the src folder
3. Download the zip file full of provided source images. You will need to uncompress this zip file and extract the images before java will be able to see them. These files must be placed correctly. These should generally go in the root folder of the project (NOT in src) – if these are misplaced you will have trouble getting tests to run
4. Download the provided expected images – but maybe place them in their own folder so they're less in the way – there's going to be enough files to manage on this project without the clutter.
5. Make sure you have a reasonable way to look **in detail** at the image files – we're looking for pixel-to-pixel color matches here so you might need to zoom in quite a bit as you test your code. IntelliJ's built-in image viewer does a good job of this on many computers.
6. From there – begin working on the 5 Transformation classes, then work on the TransformationUtils class at the end.

6 Formal Requirement: Code reading

There are four provided classes for this assignment. Two of these classes are relatively simple and two are a bit more complicated.

RGBColor This class is a simple class representing a color as three numbers: red, green, and blue. These numbers take a value between 0 and 255 with 0 meaning "none" and 255 meaning "a lot". For example the color red would be (255, 0, 0) (full red, no green or blue), and the color purple would be (255, 0, 255) (full red, no green, full blue). This is a common way to represent colors inside a computer, and it's entirely possible you've read about, or used this before. If you have not, you may find it useful to google "RGB color" briefly – there are plenty of quick guides online that can explain the idea in more depth.

The RGBColor class is immutable – meaning once a color is made, there is no way in the public API to change the red, green, or blue values. There are two static methods you may find useful at the end of the class, one is used to make sure an int representing one of these color components is within the acceptable range (0 to 255). The other is used to compute a distance between the two colors – the idea is that two colors should be visually less similar if the distance is large, and two colors should be visually similar if the distance is small.

RGBImage The RGB image class represents a full image as a 2D grid of RGBColors. There are a few basic functions for accessing and changing the individual color values. These functions are relatively self-explanatory, so I won't spend more time explaining them here.

However, if you find you are unsure about these functions after reviewing the class – please do reach out to course staff for more information.

RGBImageUtil This class's code is pretty complicated. You are not advised to read this class' code. However **you are required** to read the public API of this class – the public functions, their names and parameters, and the javadocs contained. In particular, make sure you know how to use this class to

- load an image
- save an image
- attempt to show an image.

This final function is untested on most environments. You are encouraged to try this function out – it can be very useful when debugging – but you may find that it doesn't work on your platform. In this case, remember that you can always save the image and then view it in intellij, or your image viewer of choice.

Transformation This class is a provided parent class for the various child classes you have to make. It has a relatively simple design with one core method for overloading: do_transform. The do_transform method takes a color and returns a transformed version of the color, based on the specified transformation. To provide context (for various transformations) you will also have access to the x and y position of the color, and the original image (which you can use to get the width/height of the image)

You should not need to edit the Transformation class, but will need to understand it's basic function and design to easily and quickly create sub-classes.

7 Formal Requirements: 5 Transformations

The first 5 classes you should program are all subclasses of the Transformation class. While that may sound like a lot, many of them require very little work to program if you make good use of inheritance. In particular, all Transformation subclasses are expected to leave the inherited transform method alone, instead focusing on overloading the protected do_transform function. Overloading this one method should be enough to change the behavior of the transformation as a whole.

In addition to overloading the do_transform function, some of these classes will need additional instance variables and a constructor. While some classes (Greyscale) take no constructor parameters, many of the other transformations are configurable – the AddBorder transformation, for example, can be built to add different border widths, or colors. These constructor parameters may seem strange at first – it will likely feel like these should be function parameters, not constructor parameters. This is a part of the nature of the "functional object" design of the Transformers – each instance of a Transformation class represents a transformation function. Therefore "configurations" to that transformation (what color

to make the image border etc.) are necessarily constructor parameters (and presumably associated private variables – so that data is available when you later need it.)

For each of the 5 separate Transformations, we will list the constructor, and what transformation is intended to be performed, as well as an example or two. As a general requirement for all 5 classes, these must inherit from the Transformation class, and should overload the do_transform function. To make this way of communicating more clear, we will start by presenting the Transformation class (which is provided)

7.1 Transformation

(Presented as an example – this is given to you already)

Constructor The Transformation class should have a 0-argument constructor. As Transformation transformation is not "configurable" – it does not need constructor parameters to modify the transformation process.

Transformation The Transformation transformation is quite simple. It "scrambles" the color channels of the image. This is easily done – at each pixel create a new color so it's red value is the original color's green value, the new green value is the original blue value, and the new blue value is the original red value. This will create an image which is still quite identifiable, but which is colored in a weird way.

Example

7.2 Greyscale

Constructor : The greyscale class should have a 0-argument constructor. As the greyscale transformation is not "configurable" – it does not need constructor parameters to modify the transformation process.

Transformation : The greyscale transformation is the most simple transformation we will implement. This is easily done – at each pixel create a new color that is a "grey" version of the original color. This can be done by setting the new color's red, green, and blue values to the same number (the average of the 3 values from the original color.) (colors that have red, green, and blue set to the same number look to be a shade of grey). As a detail for the mathematical computation: use simple integer division (so always round towards 0).

Example

7.3 Brighten

The brighten transformation will make images lighter (or darker) by a selected amount. This is easily achieved by increasing or decreasing the red, green, and blue values.

Constructor : The Brighten class should have a 1-argument constructor.

- amount – an integer. This indicates how much to change the brightness of images this object transforms. A positive value would mean "make images brighter", a negative value would indicate "make images darker".

Transformation : The transformation is relatively simple to implement. Independently, for each pixel, change the color by adding the amount (see constructor) to red, green, and blue. If this would cause the red/green/blue value to go over 255 set that value to 255. If this would cause the red/green/blue value to go below 0, set that value to 0.

Example
Brighter by 110

Darker by 40 (amount = -40 – note this is most notable if you look at originally bright parts of the image)

7.4 AddBorder

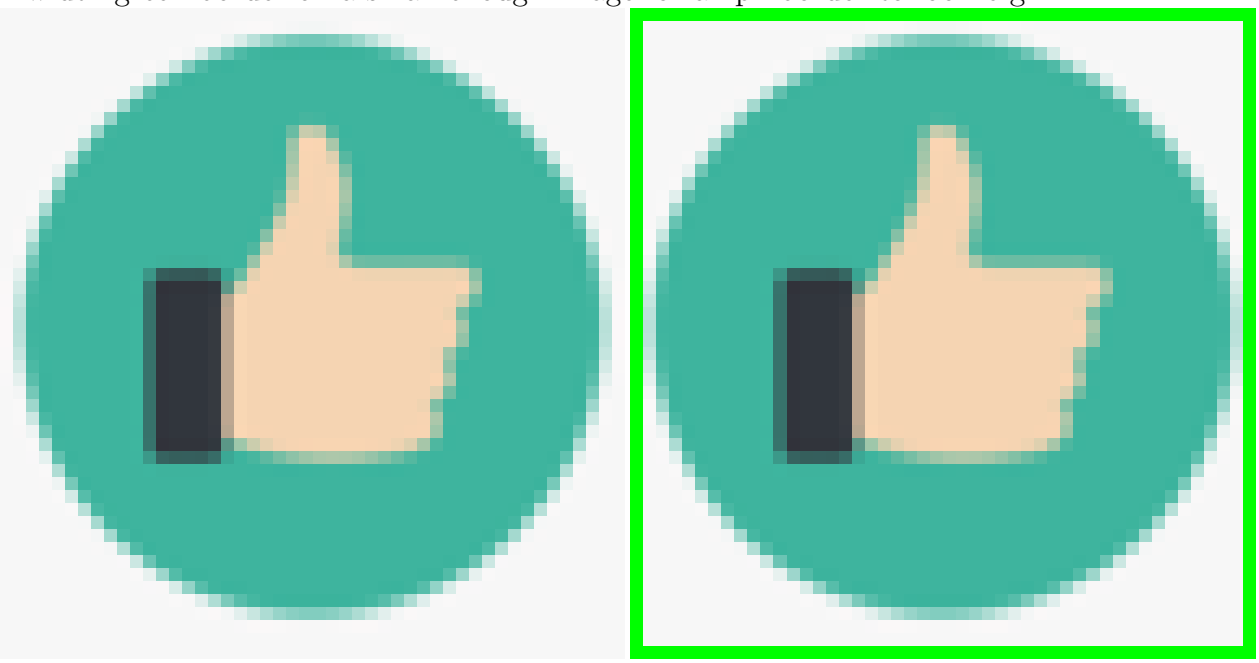
The add Border Transformation draws a variable-width border around the outline of an image. It will not change the size of the image to do this, therefore this border will be drawn OVER part of the image.

Constructor : The add border class should have a two-argument constructor

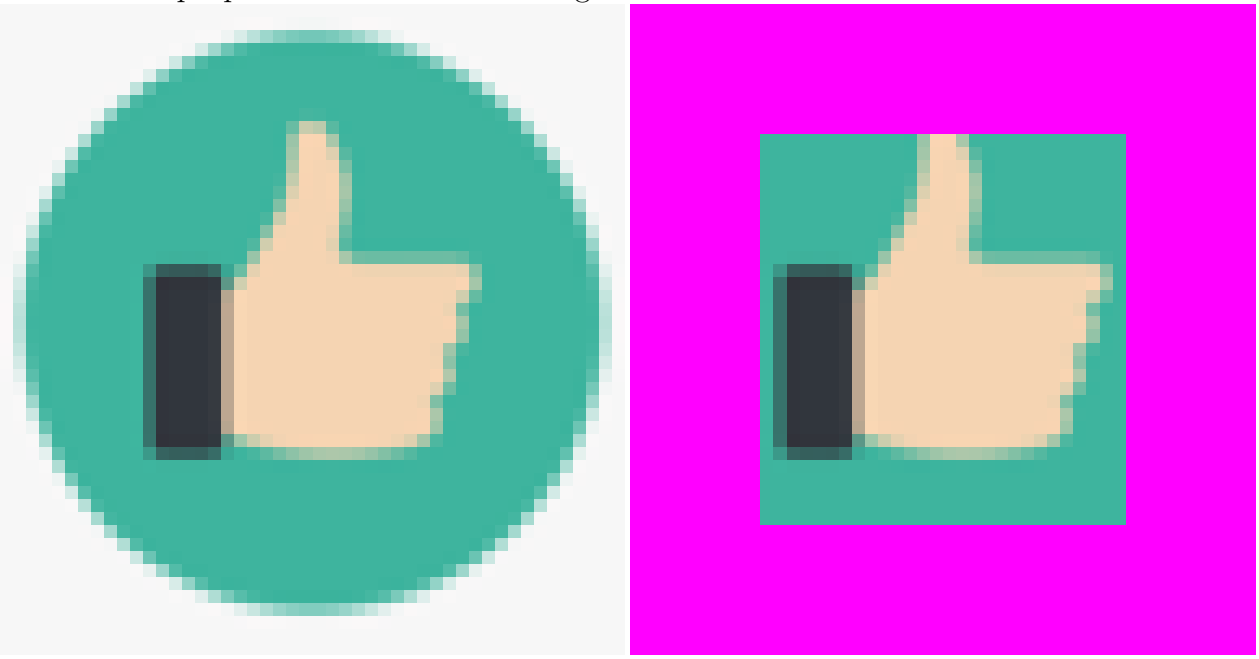
- width – an integer. This changes the width (number of pixels) of the border that is drawn.
- borderColor – an RGBColor. This changes the color of the border.

Transformation : Performing this transformation is simple – if we are transforming an "edge" pixel, set it to the border color, otherwise simply leave the pixel at it's original color. The hard part will be making sure you have an accurate boolean expression for asking if a pixel is near one of the 4 edges. In particular, I recommend checking carefully if you've set the correct amount of pixels on each edge – an off-by-one issue will be quite easy (either making too wide, or too small of a border on one of the four edges.) One useful debugging strategy might be transforming a very small image – which you can then zoom in on and see the actual pixels. This is particularly easy to do in Intellij.

Example
1 width green border on a small-enough image for a 1px border to look big



10 width purple border on a small image



10 width yellow border on a larger image (so the border appears smaller, but is *technically* the same number of pixels)

7.5 ColorPallet

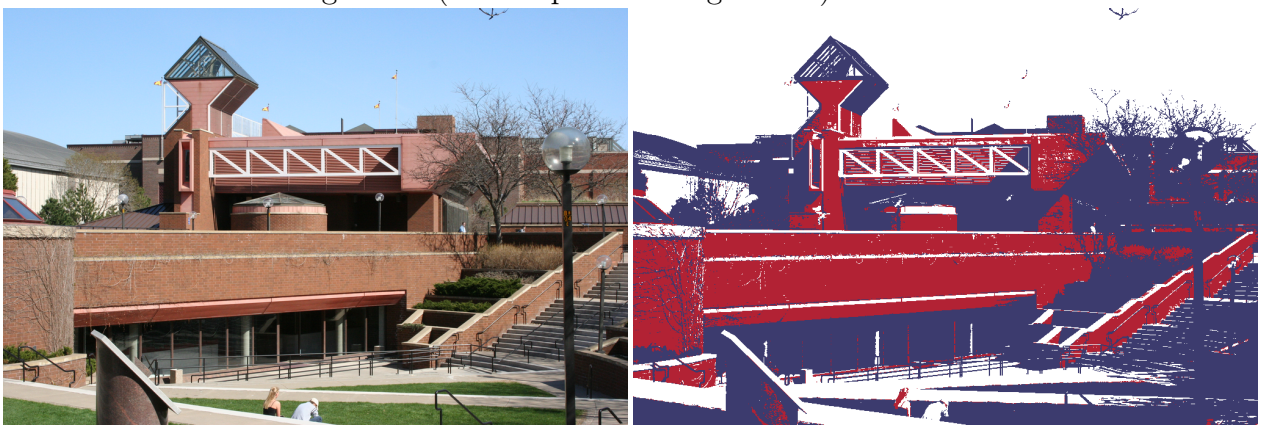
The color pallet transformation will modify images to have a reduced pallet of colors. Depending on how well the color pallet matches the image this can achieve many different effects. With a well-matched color pallet, this can give a cartoon "cell shaded" look. With a more esoteric color pallet, this can lead to images that look like they belong on political posters.

Constructor : The ColorPallet constructor should have one parameter.

- pallet – an array of RGBColors. You can assume that this array itself is not null, and that all spaces in the array store non-null colors. Likewise you can assume that the length of the array is at least 1.

Transformation : While there are, in theory, a lot of ways to match a full-color image to a color pallet, we will take one of the most straightforward, and flexible approaches. For each pixel, compute the distance from the original color to each color in the color pallet. Then, for that pixel, return the closest (lowest distance score, so most visually similar) color in the color pallet. Remember – there's a distance function available on the color class to make this a bit easier.

Example
Shifted to American flag colors (3 color pallet – all get used)



Shifted to a pleasing lilac-focused color pallet I found online (5 color pallet – not all of which appear to get used.)

7.6 Stamp

This transformation object has many possible uses, but most clearly matches the idea of a "watermark" – a small semi-transparent image placed (stamped) upon another image as a way of embedding ownership information. Our version will place a provided image into the top-left corner of the images it transforms.

Constructor : The Stamp constructor should have a single parameter:

- toStamp – and RGBImage to apply to the images this object transforms.

Transformation : The transformation process here is a little more involved than others. Since the stamp will go in the top left corner of the image, you first need to decide if a given pixel in the image is located within that top-left area where the stamp is going. Pixels not located in that area should have the original color returned unmodified.

However pixels located in the top-left (x between 0 and the width of toStamp, y between 0 and the height of toStamp) should have their colors modified. The new color for that location should be a blend of the original color in that location, and the related color for that position in the toStamp image. You can perform this blending by setting the color-values (red,green,blue) equal to the average of the original color value, and the related stamp color value. (As with greyscale – use simple integer division here, and round towards 0.)

Mathematical example, if we have a 10x10 stamp image, and a 100 x 100 image to transform, when transforming pixel (5, 40) we would simply return the original image's color at position (5, 40). However, if we were transforming the pixel at position (4, 6) we would need to get the original image's color at (4, 6) and the stamp's color at (4, 6) and make the new color an "average' of these two original colors (element by element.)

$$new_{4,6}^{red} = (original_{4,6}^{red} + stamp_{4,6}^{red})/2$$
$$new_{4,6}^{green} = (original_{4,6}^{green} + stamp_{4,6}^{green})/2$$
$$new_{4,6}^{blue} = (original_{4,6}^{blue} + stamp_{4,6}^{blue})/2$$

When averaging, be sure to use integer division. This "averaging" approach will give the stamped image a semi-transparent look.

As with addBorder – this transformation has some quite-specific positional requirements. Therefore we again recommend focusing on smaller images when testing visually.

Example
stamping the groovy image with the thumb image (it's hard to see, focus on the top-left)

stamping with the groovy image – which is quite frankly a bit too big for stamping this image.

8 Requirement: TransformationUtils

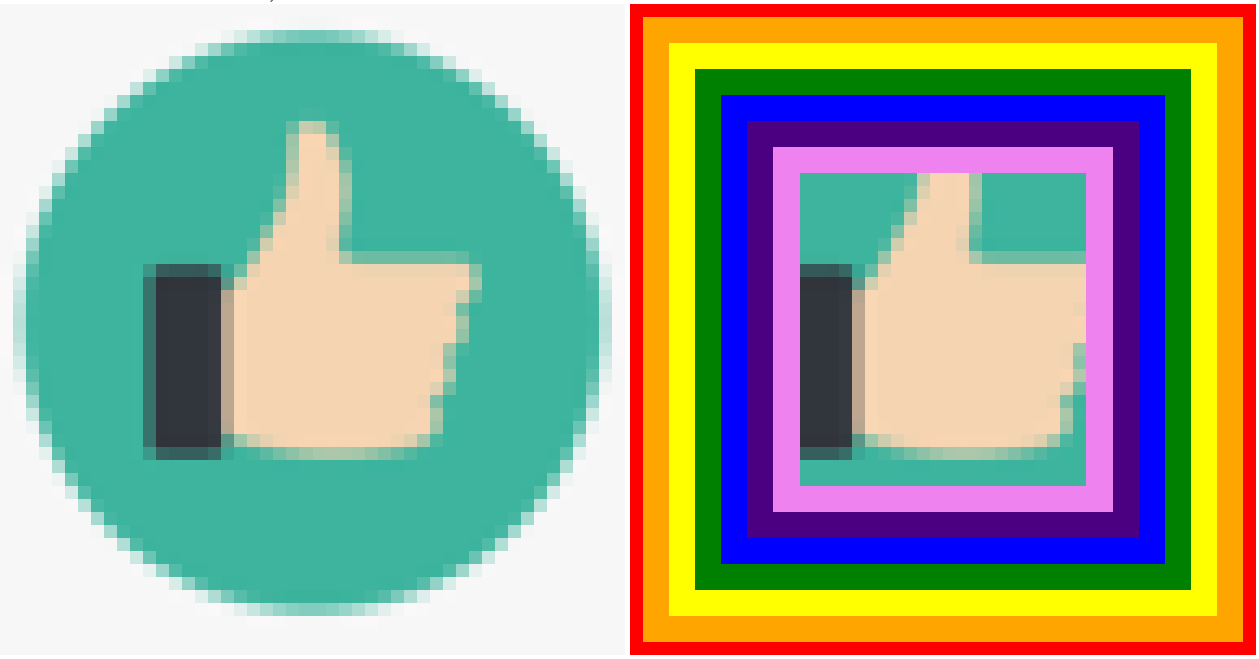
The last class you are required to make is TransformationUtils – this has a single function:

```
public static void transformMany(Transformation[] transformations, String inptfle, String otpFle)
```

This function should take an array of transformation objects, as well as two strings to indicate the input and output file names. Then it should load the image – apply each transformation (one after another) and then save the final output image. This function will be polymorphic in nature – that is to say, you should expect the transformation array to contain not-only instances of the base Transformation class, but possibly any Subclass of Transformation. Your code will, therefore, be expected to work correctly with all 6 Transformation types (1 provided, 5 you wrote) and also **any other Transformation subclass** (even ones you may not be aware of, or which are programmed in the future.) If done well, this impressive sounding requirement does not require equally impressive code – if you find yourself coding A LOT to make this work, you're probably approaching it incorrectly.

Examples

Adding 7 different boarders (starting with the largest, then adding a smaller boarder on top of that and so-forth)



greyscale, followed by a border, then finally a stamp (gotta make sure the stamp is last, or the border would cover it!)

brighten by 100, then map to the color pallet from before, then darken, then stamp.


9 Submission

For this lab you need to turn in six files:

- `Greyscale.java`
- `Stamp.java`
- `AddBorder.java`
- `ColorPallet.java`
- `Brighten.java`
- `TransformationUtils.java`

You can submit other files if you wish, but we do not promise to look at them during grading.

10 Grading

This assignment will have a manually graded component and an auto-graded component. The autograded component is worth 42 points

- 6 points for each of 5 Transformation subclasses (total 30 points)
- 4 points for each of 3 TransformationUtils tests (total 12). Note – these tests use the base transformations – even if your TransformationUtil class is correct you can lose these points if the individual Transformations are not correct.

The remaining 58 points are allocated to manual grading:

- 10 points for code style
- 8 points for each of 6 classes.