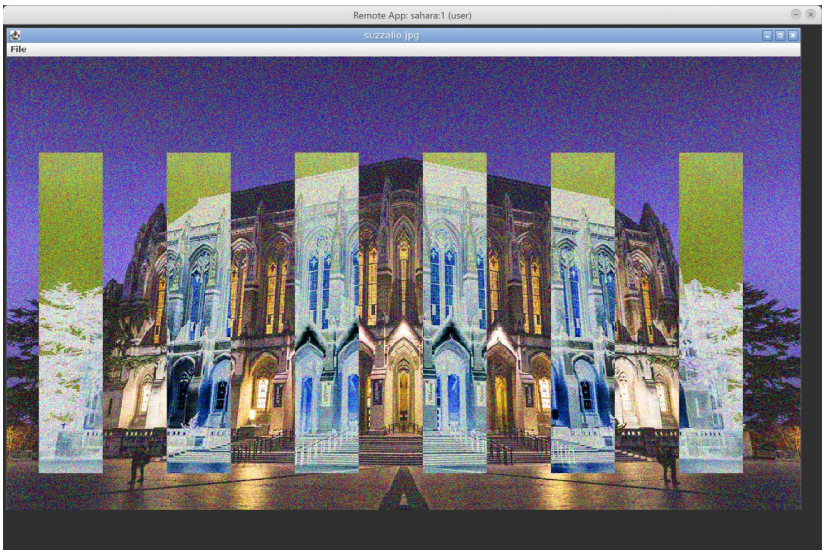
C1

1️⃣ General Image Manipulation Specification

For the first part of this assignment, you will be implementing a number of image manipulation methods which you will then use to produce an interesting output. For each Task listed below, you will create a single new method to manipulate an image. After working on all 4 tasks, you will produce and turn in an image that you create using a combination of the filters and edits from the previous tasks. Your program will also produce images after each task showing the effect of the filters which have been applied so far.

Below is an example of what your final image might look like, created by applying a combination of Task 1 - 4's methods on a photo of Suzzallo hall:



Your program will produce an image like the above, but changed by your own creative decisions. We have provided a scaffold main which loads the initial image from a file, saves changes after each filter, and saves a finalized version at the end. Your task is to write the filter methods and call them from the appropriate places in `main`.

Summary: For this part of the assignment, you will complete the following tasks (details below)

- Task 1: Implement your own filter to change the color of an image
- Task 2: Choose at least one of the choice of filters to implement for this task
- Task 3: Modify the filter(s) you implemented for Task 2.
- Task 4: Implement mirror right

Working with your own Images

Although we have loaded a sample image of Suzzallo, and provided a secondary image of Drumheller fountain, you can use and work with any image you desire so long as it follows some basic criteria:

1. Your image is not hateful, offensive, sexually explicit, or created with the intent to cause harm to others.
2. Your image is within the public domain or is an image that you took / have ownership of.
3. You and anyone else in the image are comfortable with TA's and instructors seeing the image you have provided in its edited and unedited states.

You are not required to choose a different image to edit, and you should try not to spend too much of your time deliberating on an image. We recommend getting everything to work with the provided images of UW, and then applying the edits to an image of your choosing afterwards.

You should take care not to work with images that are too large in this assignment. The number of pixels that you are editing will quickly grow out of hand as the size of your image increases. Due to computational limitations with Ed, **if you are trying to work with an image that is too large, your program will never produce an output. You should try to limit your images to be less than 900 x 900 in resolution, or some resolution with a similar number of pixels.**

Feel free to take advantage of the image re-sizing software on your computer or on the web to alter an image to be suitable to work with in this assignment.

Uploading an Image to Ed

In order to work with your own photo you need to upload it to the Ed environment on the following slide. You can do this by clicking the '+' in the top left of the coding area, and then selecting upload. After this, change the name of the image being loaded on line 6 to the name of your image as it is in the Ed file explorer.

2️⃣ Program Behavior

You will be marked on the culminating image, `creative4.jpg`, that is produced by your final turn-in when run is pressed.

Your program should also produce progress images, one saved after each new filer is applied, named `creative1.jpg`, `creative2.jpg`, `creative3.jpg`. These images are saved so you can see how your image is changed at each step of the program.

Since the exact output of this assignment is chosen by you, there are no tests associated with the individual tasks of this assignment.

Task 1

For the first task in this assignment, we will be copying the `increaseRed(Color[][] pixels)` method from the previous slide's tutorial and changing it to increase a color or shade of your choosing.

The method used to produce the sample Suzzallo photo made each color more purple by increasing the RGB values of each color by a ratio similar to that of purple. Since purple is composed of roughly 63% red, 13%

green, and 94% blue, we can make a color more purple by increasing the RGB values of a color by that ratio, e.g. red + 63, green + 13, blue + 94.

This isn't a perfect purple filter, but for our purposes it works well enough.

You should implement a similar filter for task 1. You can choose to increase purple like we did, or you can experiment with your own color combinations and ratios. If you want to keep things simple, we recommend only boosting a single RGB component to create your filter; we have already implemented an increase of red however and so you should use a different color.

We also recommend following the structure of the example `increaseRed()` method from the previous slide's tutorial. Remember that no RGB color component can have a value greater than 255; you will need to use a call to `Math.min()` to limit any potential increases to 255.

This method can have any name of your choosing, but we suggest something along the lines of `"increaseColor()"`, where `"Color"` is replaced by the name of the color or shade that you are increasing with this filter.

After your method is called from main, your program should save the updated image as `creative1.jpg`.

Task 2

For Task 2 you have three options for which filter to implement. **To receive full credit on this assignment you only need to complete one of the following three filters**, although you can implement more if you would like. All three of these tasks are similar in difficulty, but we have listed them with what we perceive to be the easiest first, and the hardest last.

1. Invert the color of your image

- **For each pixel in your image, you should change its color to the negative of itself. This is done by subtracting each RGB component from 255 and creating a new color from the result.** For example, an RGB color with the values R = 40, G = 100, B = 255 would be inverted as such: `int r = 255 - 40;int g = 255 - 100;int b = 255 - 255;Color negativeColor = new Color(r, g, b);` The new `negativeColor` having the RGB values R = 215, G = 155, B = 0. By applying this transformation to each pixel, the negative of the image will be produced. Your method should have the following method signature: `public static void negative(Color[][] pixels) {`

2. Convert the image to grayscale

- **For each pixel in your image, you should take an average of the RGB components and create a new color with each component equal to this average.** The resulting image will be a grayscale version of the previous. For example, an RGB color with the values R = 150, G = 210, B = 14 would be converted to grayscale as such: `int average = (150 + 210 + 14) / 3;Color avgColor = new Color(average, average, average);` This new `avgColor` would have the RGB values R = 124, G = 124, B = 124. Notice that in the equation `(150 + 210 + 14) / 3` we perform `int` division, not `double` division. Since the creation of a new RGB `Color` construct requires the RGB components to be `ints`, we allow truncating `int` division to take place. This saves us from casting our values to an `int` later on and will not meaningfully effect the output of the method. Your method should have the following method signature: `public static void grayscale(Color[][] pixels) {`

If you choose to implement the `grayScale` functionality, the image manipulation performed by the other tasks may not be visible. **Please do include method calls for each of your tasks in `main` anyway!** The TAs will

look at the incremental images, eg: `creative1.jpg` , to see the effect of each filter even if that effect is later overwritten.

3. Make the image grainy

- This method is similar to the `increaseRed()` example from the previous slide, except **for each pixel we will randomly choose whether to increase red, green, or blue by 100**. You should create and use a `Random` construct within your method to create a pseudorandom number for the selection of red, green, or blue to increase. Remember that no RGB color component can have a value greater than 255; you will need to use a call to `Math.min()` to limit any potential increases to 255. Your method should have the following method signature: `public static void grainy(Color[][] pixels) {`

After you call the method for one of the three filters above from main, your program will save the image in a file called `creative2.jpg`. Note that `creative2.jpg` will show the effects of Task 1 and Task 2 since these filters are applied cumulatively to the image.

Task 3

For this task, you will modify one of your previous filters to specify which portion of the image will be modified. You will now allow the caller to specify a rectangle within the image that the filter will be applied to. This will be done through supplying the top left and bottom right coordinates of the rectangle to modify as parameters to the new method.

You should create a copy of one of the functions you have completed from above, and add four new `int` parameters. For example, the `negative()` function from Task 2's method signature would be modified as such:

public static void negativeRectangle(Color[][] pixels, int row1, int col1, int row2, int col2) {

You can choose any appropriate name for this new method. Make sure that it is clear through the name and comments which of your previous methods you have adapted for Task 3.

In order for your rectangle filter to be perceived after you complete Task 4 (below), you should make sure to pass coordinates to your Task 3 method that represent a rectangle on the **right half** of the picture.

After your program calls the method for this task, your program will save the image in a file called `creative3.jpg`. Note that `creative3.jpg` will show the effects of Task 1, Task 2, and Task 3 since these filters are applied cumulatively to the image.

Task 4

In the previous tasks, we only manipulated pixels in-place. This means that we retrieve a pixel from coordinates (x, y), modify it, then save it in the same place. For this task, we will be retrieving pixels from a different location to where we are storing them, specifically we will be mirroring the right half of the image onto the left.

For each pixel on the right half of the screen, you should put the same RGB values on the mirrored x-position on the left side of the screen.

Your method should have the following method signature:

public static void mirrorRight(Color[][] pixels) {

This Task is harder than the ones before it, and so we are providing some additional tips here:

- You will have to change the bounds on one of your for loops for this Task.
- The (x, y) coordinates of the pixel you change are not the same as the pixel you load. You should draw a picture to help visualize the Task. Thinking through some concrete cases can be very helpful.

  For example if you have a Picture that is 100 pixels wide and 50 pixels tall:

  - What coordinates do you want your for loop to go through?
    - What are the minimum and maximum values of x and y?
  - If you have a point at x = 75, y = 10, where would you want to mirror that to?
  - Can you recreate this transformation using variables that you have access to inside your for loops?

Because there are 4 tasks for you to complete, your finished solution should contain **4 methods that are called from** `main` (one for each task). You may have more methods if you would like to improve the functional decomposition of your program. You may call some methods multiple times (eg: to create multiple rectangles in Task 3).

After calling `mirrorRight` your program should save the finished version of your image to `creative4.jpg`. This file will show the cumulative effect of all the filters you've created.

3️⃣ Assignment Requirements

There is no formal autograder for this assignment. Instead, we will be visually inspecting the images your program outputs to ensure the filters produce the desired outcome. As long as your results look visually similar to the operations described, your solution will have correct behavior.

For this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- When possible, you should avoid creating unnecessary objects in your program. In many cases you need to make a new object (e.g., making a new `Color`), but avoid creating objects that aren't needed. In particular, you should only be calling the `getPixels` method once to create the 2D Array of colors and shouldn't construct any additional 2D arrays to solve these problems.
- You should comment your code following Commenting Guide. You should write comments with basic info, a class comment, and a comment for every method other than main.

Compile.sh

You may notice a file named `compile.sh` in the workspace for C0. This file is used by our TAs to execute your program during grading and ensure the produced image files are appropriately saved. You should not edit or remove `compile.sh`.

Face Averaging Specification

1️⃣ Working with Multiple Images

While looking at one image is interesting, when looking across many images we can often find patterns emerging that can tell us about the places and people shown. One way to get a visual sense of this is by looking at the **average of multiple images**. For example:

- Prof. Eddy Lopez at Bucknell uses the averaging of images to look at a period of war in his home country.
- Face averaging techniques can be used to look at trends across time.
- They can also be used to look at trends across countries.

Face Averaging

One application of averaging images is the averaging of faces. By creating a data set of faces with fixed eye positions, we can apply an average of the RGB values on each pixel across the data set to produce a composite average face.

Smaller data sets can also be made from the overall one to create different results, looking at a part can give us more insight than looking at the whole sometimes.

For the final part of this assignment, you'll be working with a face averaging algorithm that we have implemented for you, along with a dataset of 10 faces. Your task will be to create your own data sets to average from the given faces, and then to reflect on the results through a series of questions on the following slide.

You can find the faces that we will be averaging in the file explorer of the following slide, but they are also linked here.

We have already written some code to synthesize a sequence of pictures of faces and produce the average face. Read and familiarize yourself with the face averaging method we have implemented below, we will explain how the code works afterwards:

This method takes an array of `Picture` constructs which contains the group of faces to be averaged. If the array contains fewer than 2 `Pictures`, an `IllegalArgumentException` will be thrown. The method also returns a `Picture`, which will be the average face produced from the array you supply.

Incremental Averaging

You might notice that the logic to compute the average pixel value looks a little more complex than what we've seen before.

Part of this complexity is the fact that we have to take 3 averages, one for each color channel (R, G, B), and store them separately. The bigger change is that this method use a slightly different algorithm for averaging a sequence of numbers called a *incremental average.* This somewhat strange averaging algorithm is quite necessary for this problem since we are using `Color` objects to represent our data.

As mentioned earlier, `Color` objects will throw an exception if given an RGB value greater than 255! This won't work though with a normal averaging algorithm since is very likely for the sume to exceed 255 even though the average doesn't.

To create an average, this method goes through each face in the `Picture` array, and creates an incremental average. Using this process, the average is calculated and updated at each element in the array.

First, an average of the first 2 images is produced, then the third image is introduced to produce the average of 3, then the fourth, etc... In order to do this you must multiply our previous average by the number of elements it took to make that average - you cannot simply add the old average to the new elements and divide by 2.

Here is a simplified example of this style of averaging compared to typical averaging with a basic list of numbers, `[1, 2, 3]`:

Typical Average Computation

Input: [1, 2, 3]
  Compute sum: 1 + 2 + 3 = 6
  Divide by num elements: 6 / 3 = 2
Output: 2

Incremental Average Computation

Input: [1, 2, 3]
  Repeatedly compute average of numbers at index 0-i

  Average of numbers with indices 0: 1

  Average of numbers with indices 0, 1: 1.5
    Multiply old average by num elements so far: 1 * 1 = 1
    Add current number: 1 + 2 = 3
    Divide by new num elements so far: 3 / 2 = 1.5

  Average of numbers with indices 0, 1, 2: 2
    Multiply old average by num elements seen so far: 1.5 * 2 = 3
    Add current number: 3 + 3 = 6
    Divide by new num elements so far: 6 / 3 = 2

Output: 2

For another intuition for this algorithm, think about tracking the average rainfall per day. If we have gone 100 days with 2 inch of rain each day (average rainfall is 2) and then on day 101 get 50 inches of rain, what is the average rainfall over this 101 day period? It's not as simple as saying `(50 + 2) / 2 = 26` since we have to account for *100 days* of 2-inch rain.

Instead, we need to compute `(50 + 100 * 2) / 101 = 2.48`. The algorithm above is doing this calculation at every step of the loop to build the next average from the previous.

This style of average allows us to access the `Pictures` one at a time and generalize our algorithm; we would need access to the value of the same pixel in each of the `Pictures` we want to average at once otherwise.

This style of averaging also works very well combined with a `for` loop given the counting variable can be used to reference your current averages sample size.

Using the algorithm

As well as the method to average an array of `Pictures`, we have also supplied some starter code within the `main` method. Read and familiarize yourself with this code, we will explain what it does afterwards:

The first thing that `main` does is create a new Picture construct for each of the images within our data set. We have labelled these Pictures as `face1` through `face10`. These images are then put into a `Picture` array, which allows them to be passed into the `faceAverage()` method and be subsequently averaged.

The last 3 lines of code create a `Picture` construct to catch the return of `faceAverage()`, call and catch said return from `faceAverage()`, and call `.save(fileName)` on the resultant `Picture` to display it to the user.

The Task

There are 2 `TODO:` comments found in the code. The first one shows where you should create your new data sets, and the second shows where you should edit the call of `faceAverage()` to take one of your new data sets. For the second `TODO:`, you must simply edit the name of the parameter within the method call to be the same as the name of the new data set you wish to average.

In order to receive full credit for this part of the assignment, you must create 3 new `Picture` array data sets, each containing 3 or more images. These new data sets can be created from any grouping of the original 10 images that you want.

Make sure to vary the reasons behind your grouping, and contemplate on the decisions you make, as we will be asking you to reflect on these processes afterwards. You should run the face averaging algorithm on each of these new data sets and observe the outputs.

You must 'turn-in' the new main method after your edits. Your TA should be able to see the 3 new `Picture` array data sets that you have created. Your code does not need to display a particular image when it is ran, but all 3 data sets you create should be able to produce an image.