#HashtagPrimePhotoFilters

Introduction

We've been working on a proof of concept (POC) that allows prime photo users to convert their photos by applying different filters. For example, the user provides a photo and requests it to be converted to a Sepia image. The service creates the new sepia image and returns it. So far we can create a greyscale, sepia, or inverted image. For the POC, we're just working with files locally.

We had it working for any single conversion. If you wanted a sepia puppy pic, you got a sepia puppy pic. A greyscale puppy pic, you got it! We're running into some trouble when you want a sepia, greyscale, AND inverted puppy pic. Then you get a few files that look all screwed up!

Before Starting

We'll be using image files as resources to test our converter. Run the main method in `PrimePhotoConverterManualTester`. When it finishes running open up the `src/output` folder. You should see a few dalmatian files, one for sepia, one for greyscale, and one for inversion. Take a look - we somehow made a dalmatian puppy not cute! If you want to see a cute sepia dalmatian, run the other test in the main method that we have commented out. The result file will be in the `src/output` folder. We have a timestamp in the path, so the results of the most recent test will be at the bottom of the list.

We could really use some help getting this fixed. Let's walk you through the code and see what you can come up with.

Code Walk Through

A request to convert a photo is received in the `ConvertPrimePhotoActivity` class. With this being a POC, we haven't developed our Lambda function yet so we haven't created any formal Request/Result objects for the `handleRequest` method. First we load the image into memory. This could take a while and use up a lot of memory depending on the photo, so we really only want to do this once. We use a static method in the `PrimePhotoUtil` class to load the photo. That class has been around for a pretty long time, and has always worked so we're pretty sure the bug isn't there. The in-memory image is saved in a `PrimePhoto`. If you take a look at that class, it defines the image's dimensions and holds all its `Pixel`s in a `List`. The `type` field is related to Java's image libraries. We keep it around so we know how to re-save the photo. A `Pixel` represents one tiny, little speck in our photo. An image is just a grid of `Pixel`s. Each `Pixel` has an x and y coordinate and a color associated with it. We store the color in a class called `RGB`. `RGB` stands for "Red Green Blue." RGB refers to three hues of light that can be mixed together to create different colors.

So how do we do a conversion? We have different strategies based on the conversion type. `ConverterStrategyMapper` keeps a track of which strategy to use based on the type of conversion. We have three strategies that all implement `PrimePhotoConverter`:

1. `SepiaConverter`
2. `GreyScaleConverter`
3. `InversionConverter`

Each strategy essentially copies the list of `Pixel`s, but with a different `RGB` value for each of them. It then creates a new `PrimePhoto` with the same width, height, and type of the original image, but with a new list of `Pixel`s. `RGB` has some methods to help do this. The `RGB` `toGreyscale()` method can transform the current red, green, and blue values to new values that represent a grey shade.

That all seemed to be working when we were just accepting one type of conversion. Our manager asked us to update the API to accept a list instead. We knew it was going to be pretty slow to do the conversions one after another, so we decided to use multiple threads. Each thread processes a different conversion. We wrote a `Runnable` called `ConcurrentConverter` that just calls the convert method of one of the strategies. We're using some more advanced techniques to manage the threads, so we added in some comments. No need to get bogged down in those details though. We need you to think about why running these conversions concurrently is giving us problems!

We're guessing it has something to do with sharing state across the threads. Concurrency is tough!

Task

Make edits to the POC to get the multiple conversion functionality working. You can validate it is working by running the following tests: `ConvertPrimePhotoActivityTest` and `ConvertPrimePhotoIntegrationTest`. This will run the activity's unit tests and the integration tests. If the integration tests fail, they will provide you with files to inspect.

You shouldn't need to edit any of the threading logic in the Activity class, nor any of the logic in `PrimePhotoUtils`.

You can validate you made all the changes expected to the model files by executing the following tests:

- `PrimePhotoIntrospectionTest`
- `PixelIntrospectionTest`
- `RGBIntrospectionTest`

You can also run the `main` method in `PrimePhotoConverterManualTester`.

Note: You do not need to check any of your files in the `src/output` directory. If there get to be a lot in there while you are testing and you find it distracting, feel free to delete any of the files.

Hints

- A strategy to write thread safe code
- Can I edit method signatures?
- The prime-photo, pixel, and rgb tests pass but my integ tests do not