CSCE 155 - Java

Lab 6.0 - Methods, Enumerated Types and Exceptions

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Read Chapters 5–6 and 29–30 of the Computer Science I textbook

Peer Programming Pair-Up

For students in the online section: you may complete the lab on your own if you wish or you may team up with a partner of your choosing, or, you may consult with a lab instructor to get teamed up online (via Zoom).

For students in the face-to-face section: your lab instructor will team you up with a partner.

To encourage collaboration and a team environment, labs are be structured in a *peer programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Basics of enumerated types
- How to use exceptions for error handling
- Have exposure to a formal unit testing framework

2 Background

Enumerated Types

Enumerated types are data types that define a set of named values. Enumerated types are often ordered and internally associated with integers (starting with 0 by default and incremented by one in the order of the list). The purpose of enumerated types is to organize certain types and enforce specific values. Without enumerated types, integer values must be used and the convention of assigning certain integer values to certain types in a logical collection must be remembered or documentation referred to as needed. Enumerated types provide a human-readable "tag" to these types of elements, relieving the programmer of continually having to refer to the convention and avoiding errors.

In Java, enumerated types are a special type of class in which you define a list of values. An example:

```
1 public enum DayOfWeek {
2         SUNDAY,
3         MONDAY,
4         TUESDAY,
5         WEDNESDAY,
6         THURSDAY,
7         FRIDAY,
8         SATURDAY
9 }
```

Elsewhere in the code, you can use the enumerated type as follows.

```
//create a variable:
DayOfWeek today = DayOfWeek.MONDAY;

//make a comparison:
if(today == SATURDAY) {
  ...
}
```

Exceptions & Error Handling

Errors in the execution of a program are unavoidable: users may enter invalid input, or the expected resources (files or database connections) may be unavailable, etc. In Java, errors are communicated and handled through the use of *exceptions*. When an error condition occurs or is detected, a program can `throw` an exception with a human-readable error message.

Exceptions can give errors *semantic* meaning. For example, a `NullPointerException` and an `IllegalArgumentException` are two distinct *types* of exceptions that can be distinguished (and thus treated differently) by the language itself. When the program executes a piece of code that could potentially result in an exception, it can be handled by using a `try-catch` block: we `try` to execute the snippet of code and, if an `Exception` occurs (is thrown) then we can `catch` it and handle it. A block of code may potentially throw multiple different types of exceptions. You can write code to handle each exception in a different manner or simply echo (print) to the user a different message based on the error. A basic example:

```
Integer a, b;
//read in a, b
try {
  if(b == 0) {
     throw new IllegalArgumentException("Division by " +
        "zero is not valid.\n");
  } else {
     c = a / b;
  }

  BufferedWriter out = new BufferedWriter(
        new FileWriter("/etc/passwd"));
  out.write("result = "+c);

} catch(IllegalArgumentException e1) {
```

```
16          System.err.println("Division by zero is undefined!");
17          System.exit(1);
18      } catch(SecurityException e2) {
19          System.err.println("You are not root, you " +
20              "can't write to the password file! ");
21          System.exit(1);
22      }
```

3 Activities

Clone the GitHub project for this lab using the following URL: https://github.com/cbourke/CSCE155-Java-Lab06

3.1 (Re)designing Your Functions

In the previous lab you designed several functions to convert RGB values to gray-scale (using one of three techniques) and to sepia. The details of how to do this available in the previous lab and are repeated for your convenience below.

In this lab you will update these functions to make them a bit easier to use and to utilize error handling. For the gray scale functions, there was a function for each of the three techniques. We can simplify this design and have only one function that takes another parameter: an enumerated type that identifies which of the three techniques (average, lightness or luminosity). In addition, you will add error handling to both functions.

1. Create an enumerated type named `GrayScaleMode` and specify the three modes.
2. Implement the `toGrayScale()` method to convert the given `RGB` using the technique specified by the given `mode`
3. Update both functions to check if the given `RGB`'s color values are all within the expected range of $[0, 255]$. If not, throw an `IllegalArgumentException` with an appropriate error message.

3.2 Running Unit Tests

As with the previous lab, you can test your functions using the full image driver program on the provided images or some of your own. However, this is essentially an *ad-hoc test* which is not very rigorous nor reliable and is a manual process.

In the last lab you wrote several *informal* unit tests. Writing unit tests automates the testing process and is far more rigorous. However, this involved writing a lot of boilerplate code to run the tests, print out the results and keep track of the number passed/failed.

In practice, it is better to use a formal unit testing framework or library. For Java, the standard unit testing tool is JUnit (https://junit.org/junit5/). We have provided a JUnit testing suite (see `ColorUtilsTests` that contains a number of unit tests for your functions. A few observations about JUnit testing code:

- There is no `main` method, but you can still run the file in Eclipse via the play button. It will launch JUnit and the framework will run all of the tests, producing a report on how many tests were run and how many passed/failed with some details on how they failed.
- Methods are designated as "test" methods using *annotations* (`@Test` for example). This tells the JUnit framework that the method performs some unit test and makes an *assertion* about the result that can be used to determine if the test failed or passed. Annotations do not affect the code, but instead give it aspects or attributes that other code can recognize and thus affect. This is known as *Aspect Oriented Programming* (or *Attribute Oriented*).
- A typical Eclipse project setup for JUnit involves separating the project code and testing code into separate source files as this project has been setup: the project files are in `src/main/java/` and the testing code is in `src/test/java/`. Note that these are *not* packages. Generally, Testing code for a class is placed in the *same* package.

Run the test suite and verify that your code passes *all* the tests. Fix any issues or bugs that become apparent as a result of this testing. Passing 100% of the provided test cases will suffice to complete the lab. However, we *highly encourage* you to read the JUnit test file to understand how the tests are setup and performed and then to add a few of your own tests.

Color Formulas

To convert an RGB value to gray-scale you can use one of several different techniques. Each technique "removes" the color value by setting all three RGB values to the same value but each technique does so in a slightly different way.

The first technique is to simply take the average of all three values:

$$\frac{r+g+b}{3}$$

The second technique, known as the "lightness" technique averages the most prominent

and least prominent colors:

$$\frac{\max\{r, g, b\} + \min\{r, g, b\}}{2}$$

The luminosity technique uses a weighted average to account for a human perceptual preference toward green, setting all three values to:

$$0.21r + 0.72g + 0.07b$$

In all three cases, the integer values should be *rounded* rather than truncated.

A sepia filter sets different values to each of the three RGB components using the following formulas. Given a current (r, g, b) value, the sepia tone RGB value, (r', g', b') would be:

$$\begin{aligned} r' &= 0.393r + 0.769g + 0.189b \\ g' &= 0.349r + 0.686g + 0.168b \\ b' &= 0.272r + 0.534g + 0.131b \end{aligned}$$

As with the gray-scale techniques, values should be rounded. If any of the resulting RGB values exceeds 255, they should be reset to the maximum, 255.

4 Handin/Grader Instructions

1. Hand in your completed files:
   - `ColorUtils.java`
   - `GrayScaleMode.java`
   - `ColorUtilsTests.java`
   - (no worksheet is necessary for this lab)

   through the webhandin (https://cse-apps.unl.edu/handin) using your cse login and password.
2. Even if you worked with a partner, you *both* should turn in all files.
3. Verify your program by grading yourself through the webgrader (https://cse.unl.edu/~cse155h/grade/) using the same credentials.

5 Advanced Activities (Optional)

5.1 Custom Exceptions

In this lab you threw an `IllegalArgumentException` if the `RGB` value was invalid. The exception was still generic/general. You can make your own project-specific exception types by creating your own class and *extending* a `RuntimeException`. Read the course textbook for details and create your own `IllegalRgbException` class. Modify your project to use it.

5.2 Ant

Large projects require even more abstraction and tools to manage source code and specify how it gets built. For Java, a popular build tool is Apache Ant (http://ant.apache.org/). Ant is a build utility that builds Java projects as specified in a special XML file (`build.xml`). The build file specifies how pieces get built and the inter-dependencies on components. Familiarize yourself with Ant by reading the following tutorials.

- http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html
- http://www.vogella.com/articles/ApacheAnt/article.html

Provided in project is an example `build.xml` file. Modify it for the code base you created in this lab and use it to compile and run the code from the command line. In particular, from the command line:

1. Place all source files into a folder named `src` in the same directory as the `build.xml` file (it should be like this already)
2. Modify the `build.xml` file appropriately by specifying what your main executable class is. To do this, modify the `main.class` property's value to the fully qualified (full path name) class that you wish to run.
3. Compile your project by executing the following command: `ant compile`
4. Run your project by executing the following command: `ant run`