COMPUTER SCIENCE 1027B – Foundations of Computer science II

Assignment 1

Due Date: February 5, 11:55 pm

1. Purpose

To gain experience with

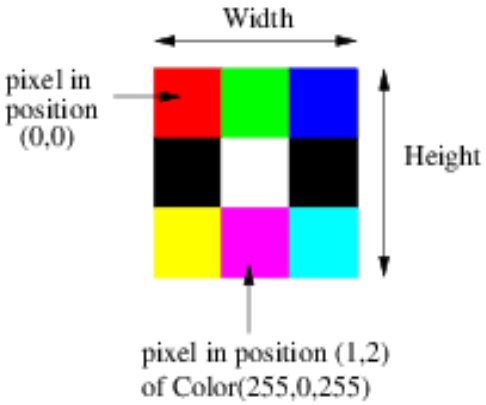
- Java and Java libraries
- Arrays

2. Introduction

The first project has to do with Java array operations. More specifically, you are asked to implement four basic image processing operations on a 2-dimensional array that represents a 2-dimensional digital image.

A 2-dimensional digital image (e.g. a *.jpg*, or a *.png* image) can be represented as a collection of "cells" or pixels (see Figure 1). Each pixel has a color. A color can be described as a combination of 3 primary colors: red, green, and blue. The color of a pixel then can be represented as a triplet of numbers color(R, G, B), where each one of R, G, and B can take integer values between 0 and 255. The value of R represents the intensity of the red color, with 0 meaning the absence of red tone and 255 meaning a full red tone present in the color. Similarly for the values of G and B. For example, color(0,255,0) represents green, color(0,0,0) represents black, and color(255,255,255) represents white.

We can assign to each pixel p of an image a position (x,y), where x indicates the distance, or number of pixels between p and the leftmost edge of the picture and y is the distance from p to the top of the image. Hence, a 2-dimensional image can be represented using a 2-dimensional array, where the [x][y] entry of the array stores the color of the pixel in position (x,y) of the image (see Figure 1).



Matrix representation

0	Color(255,0,0)	Color(0,255,0)	Color(0,0,255)
1	Color(0,0,0)	Color(255,255,255)	Color(0,0,0)
2	Color(255,255,0)	Color(255,0,255)	Color(0,255,255)
	0	1	2

Note. A grayscale image can be represented in the same manner, but all R, G, and B values are equal for each pixel.

The standard libraries of Java has a package called `java.awt` that contains a class called *Color* (https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html) which we will use to represent pixel colors to be stored in the 2-dimensional array. This class provides 3 methods, *getRed(), getGreen(),* and *getBlue()* for getting the values of the red, green, and blue component of each pixel.

Please read the supplied *InverseOperation.java* class to learn how to define a 2-dimensional array storing the pixel colors and how to use the 3 above methods to get the individual R, G, B color components of each pixel.

3. Classes Provided

Class *ImageLoader.java* contains methods to read an image stored in a file with extension *.jpg,* or *.png* and to store it in a 2-dimensional array of objects of the class *Color* described above. To test your program, you can download any image in *jpg* or *png* format from the internet. We also provide a collection of *jpg* files for your reference and testing. The class *ImageLoader* contains two methods, *getColorArray*, and *getGrayscaleArray* that store the pixels of an image in a 2-dimensional array of type *Color*[][]. This is the array that you will use to manipulate the input image, as explained below. Method *getColorArray* stores the pixels of the array in their original colors, while *getGreyscaleArray* stores the pixels in a grey scale format. You will not need to use these methods or any of the methods described in this section in the code that you are required to write; we provide this information for your reference only.

Class *ImageConverter.java* contains auxiliary methods needed by *getColorArray* and *getGreyscaleArray*. It also provides methods for converting the 2-dimensional representation of an image into a format suitable for storage in a *jpg* or *png* file.

Class *ImageWriter.java* contains a method called *writeImage* that takes as arguments an output file and an image represented as a 2-dimensional array and stores this image in the output file. Method *writeImage* uses the auxiliary method *writeFile* also defined in this class.

Class *ImageProcessing.java* contains the main method for your program. Hence, to run your program from a terminal or command window, once you have compiled all your java classes, you will type the command

java ImageProcessing

See below for explanations of how to run your program from Eclipse. The program will read a configuration file containing an image file and a set of image operations (described below); the program will perform the specified operations on the image and the resulting image will then be saved in an output file. The format of the configuration file is specified later.

Class *ImageProcessing* has methods to read the image to be processed and the set of operations that are to be performed on it. This class also has methods to determine the output directory where the processed image will be stored.

Method *processImage* will invoke the code that you will write (explained below) to perform the required operations on the input image. To keep the program simple and to maintain uniformity in the way in which the different image operations are invoked, we provide you with a Java interface class called *ImageOperation* that defines a method called *doOperation*. The *doOperation* method takes as an argument a 2-dimensional array of type *Color*[][] storing an image, as explained above, performs a specific operation, and returns a new 2-dimensional array of type *Color*[][] containing the processed image. The signature of this method is

public Color[][] doOperation(Color[][] imageArray).

This means that all your Java classes will implement the *ImageOperation* Interface; that is, they will all implement the *doOperation* method. To clarify this look at the sample inverse image operation described in the next section.

3.1 Inverse Image Operation

The first image operation that we will consider is the inverse operation that inverts the colors of an image, so black pixels become white, while pixels become black, and other pixels change color to their inverse. If a pixel p has color(R,G,B) the inverse color is defined as color(255-R,255-G,255-B). To perform this operation the 2-dimensional array storing the colors of the pixels is scanned and for every entry the color of the corresponding pixel is inverted. The java code to perform this operation is in the provided *InverseOperation.java* class, which for convenience we show below.

```
import java.awt.Color;
public class InverseOperation implements ImageOperation {
	public Color[][] doOperation(Color[][] imageArray) {
		int numOfRows = imageArray.length;
		int numOfColumns = imageArray[0].length;

		// 2-dimensional array to store the processed image
		Color[][] result = new Color[numOfRows][numOfColumns];

		// Scan the array and invert each color
		for (int i = 0; i < numOfRows; i++)
			for (int j = 0; j < numOfColumns; j++) {
				int red = 255 - imageArray[i][j].getRed();
				int green = 255 - imageArray[i][j].getGreen();
				int blue = 255 - imageArray[i][j].getBlue();
				result[i][j] = new Color(red, green, blue);
			}
		return result;
	}

}
```

Note the first line

```
import java.awt.Color;
```

That allows the program to use the methods from class Color that, as mentioned above, is part of the Java standard libraries. Note also the third an fourth lines that show how to obtain the width and height of the image, information needed to determine the size of the 2-dimensional array `result` that will be used to store the modified image:

```
Color[][] result = new Color[numOfRows][numOfColumns];
```

An example input and output images for this operation is provided below.

4. Image Processing Operations to Implement

For this assignment, you will implement four different operations: contour detection, threshold filtering, adjusting the brightness of pixels based on their distance to the upper left corner, and increasing the size of the image. As mentioned above, each one of your java classes must implement the *ImageOperation* class. Hence, the headers of your classes must be as follows:

```
public class ContourOperation implements ImageOperation {
public class ThresholdingOperation implements ImageOperation {
public class AdjustmentOperation implements ImageOperation {
public class MagnifyOperation implements ImageOperation {
```

Operation 1: Contour. Contour Detection

To perform contour detection, we will need to examine the "color distance" between a pixel and its neighbouring pixels. Note that depending on the position of a pixel in an image, it might have 3, 5, or 8 neighbouring pixels. For example, the red pixel in Figure 1 has 3 neighbours: green, white, and a black pixel. The green pixel has 5 neighbours: red, blue, white, and 2 black. The white pixel has 8 neighbours.

The "color distance" between two pixels is defined as the square root of the sum of the squares of the differences between the R, G, and B components of the pixels. For pixel p, let p.red, p.green, and p.blue denote the red, green, and blue components of its color. Then, for pixels p1 and p2,

$$\text{Color distance(p1,p2)} = \sqrt[2]{(p1.red - p2.red)^2 + (p1.green - p2.green)^2 + (p1.blue - p2.blue)^2}$$

(Note that this is the same definition for Euclidean distance between 2 points in the 3-dimensional space.)

To perform contour detection for an image you will consider each entry of the 2-dimensional array storing the colors of the pixels of the image, and if for some pixel p the color distance between p and any of its neighbours is more than 65 you will change the color of the pixel to black (i.e. Color(0,0,0)); otherwise you will change the color of the pixel to white (i.e. Color(255,255,255)).

Note that the above threshold value 65 is arbitrary, but this is the value that we will use to test your implementation of this operation. Look at class *InverseOperation.java* to get an idea of what this class should look like. You must implement this operation in a Java class called *ContourOperation*.

An example input and output images for this operation is given below.

Hint. Use Java class Math to compute the square root of a value. Documentation for all java classes in the standard library is available online. In Google type "java math class". This will take you to the Oracle documentation page that explains all the methods from the java class Math.

When considering a pixel p and computing the color distance to its neighbours, you need to verify that the neighbours exist. For example, let p be a pixel at position (x,y). Its neighbours are at positions (x-1,y-1), (x-1,y), (x-1,y+1), (x,y-1), (x,y+1), (x+1,y-1), (x+1,y), and (x+1,y+1). Before trying to compute the color distance between p and, say, neighbour (x-1,y-1) you must check that x > 0 and y > 0 as otherwise your Java program will crash with an *ArrayIndexOutOfBounds* exception because no array has an entry with indices [-1][-1].

Operation 2: Thresholding. Threshold Filtering

A threshold filter displays each pixel of an image in only one of two colors, black or white. To perform this operation, you will consider each pixel of the image and if the "*brightness score"* of a pixel is greater than 100, change the color of the pixel to white, otherwise change its color to black.

The "*brightness score"* of a pixel p is defined as:

Brightness score(p) = 0.21**p.red* + 0.71**p.green* + 0.07**p.blue*

The new image will have pixels in only one of the two colors black or white. You must implement this operation in a Java class called *ThresholdingOperation*.

An example input and output images for this operation is provided below.

Operation 3: Adjustment. Adjusting the Brightness Based on Distance to Upper-Left Corner of Image

This operation changes the brightness of each pixel p depending its Euclidean distance from the upper-left corner of the image. If pixel p is in position (x,y), then its distance to the upper left corner of the image is D = $\sqrt[2]{x^2 + y^2}$. The maximum distance from any pixel to the upper left corner of the image is M = $\sqrt[2]{width^2 + height^2}$, where *width* is the width of the image and *height* is its height. Then, the color of pixel p must be modified as follows:

adjustBrightness = D / M
p.red = p.red adjustBrightness*
*p.green = p.green * adjustBrightness*
p.blue = p.blue adjustBrightness*

An example input and output images for this operation is provided below.

You must implement this operation in a Java class called *AdjustmentOperation*.

Operation 4: Magnify. Increasing the Size of an Image

This operation changes the size of the image to twice its width and twice its height. To perform this operation, you will create a 2-dimensional array that has twice as many rows and twice as many columns as the array storing the image. Then you will copy each pixel of the original image into 4 positions of the enlarged array (you need to think which positions of the array you need to use). You must implement this operation in a Java class called *MagnifyOperation*.

An example input and output images for this operation is provided below.

5. Configuration File

To run the program, we need to specify a configuration file. The configuration has the following format

<image-type> <image-file-name> <op1> <op2> …

where

- <image-type> can be *Color* or *Grayscale* (with first letter in uppercase, as shown). If <image-type> is *Color* the input image will be stored as a color image, otherwise it will be stored as a grey scale image.
- <image-file-name> is the name of the image file that will be processed.

- <op> can be any of the following: *Contour*, *Inverse*, *Thresholding*, *Adjustment*, *Magnify* (with first letter in uppercase). These are the image operations to be performed on the input image. The operations are applied to the image in the order specified, i.e. first <op1> is applied, then on the resulting image <op2> is applied, then on the resulting image the next operation is applied, and so on. Observe that the order in which the operations are listed will change the final resulting image.

Example 1

```
Color C:/users/mary/test1.jpg 3 Thresholding Adjustment Contour
```

Means that the file C:/users/mary/test1.jpg file will be treated and stored as a color image, and three operations will be performed on it in the given sequence: first threshold filtering, then brightness adjustment, and finally contour detection .

Example 2

```
Grayscale C:/users/mary/test2.jpg Inverse Magnify Magnify
```

Means that the file C:/users/mary/test2.jpg file will be treated and stored as a grayscale image, and then 3 operations will be applied: inverse, and twice magnify.

6. Non-Functional Specifications

- **Assignments are to be done individually and must be your own work**. **Software will be used to detect cheating**.
- You must properly document your code by adding comments where appropriate. Add comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add comments to methods to explain what they do and to instance variables to explain their meaning and/or purpose. Also add comments to explain the meaning of potentially confusing parts of your code.
  When deciding where to add comments, you need to use your own judgment. If the meaning of a method, instance variable, or fragment of code is obvious, you do not need to add a comment. If you think that someone else reading a fragment of your code might struggle to understand how the code works, then write a comment. However, try to avoid meaningless comments like these:

```
i = 1;       // initialize the value of i to 1
i = i + 1;   // increase the value of i
if  (i == j) // compare i and j
```

- Use Java coding conventions and good programming techniques, for example:
  - Use meaningful variable and method names. A name should help understand what a variable is used for or what a method does. Avoid the use of variable names without any meaning, like *xxy*, or names, like *flower*, that do not relate to the intended purpose of the variable or method.
  - Use consistent conventions for naming variables, methods, and classes. For example, you might decide that names of classes should start with a capital letter, while names of variables and methods should start with a lower-case letter. Names that consist of two or more words like *symbol* and *table* can be combined, for example, using ``camelCasing'' (i.e. the words are concatenated, but the second word starts with a capital letter: *symbolTable*) or they can be combined using underscores: *symbol_table*. However, you need to be consistent.

- Use consistent notation for naming constants. For example, you can use capital letters to denote constants and constant names composed of several words can be joined by underscores: *TABLE_SIZE*.
- Use constants where appropriate.

- Readability.
  - Use indentation, tabs, and white spaces in a consistent manner to improve the readability of your code. The body of a for loop statement, for example, should have a larger indentation than the statement itself:

```
for (int i = 0; i < TABLE_SIZE; ++i)
        table[i] = 0;
```

  - Positioning of brackets, '{' and '}' to delimit blocks of code should be consistent. For example, if you put an opening bracket at the end of the header of a method:

```
private int method() {
        int position;
```

    then you should not put the bracket in a separate line for another method:

```
private String anotherMethod()
{
        return personName;
```

7. Submitting your Work

You **MUST SUBMIT ALL YOUR JAVA** files through OWL. **DO NOT** put the code inline in the text-box provided by the submission page of OWL. **DO NOT** put a ``package'' line at the top of your java files. **DO NOT** submit a compressed file (.zip, .tar, .gzip, ...); **SUBMIT ONLY** .java files.

Do not submit your .class files. If you do this and do not submit your .java files, your assignment cannot be marked!

8. Marking

What You Will Be Marked On:

- Functional specifications:
  - Does the program behave according to specifications?
  - Does it run with the sample input image files provided and produce the correct output?
  - Are your classes implemented properly?
  - Are you using appropriate data structures?
- Non-functional specifications: as described above.
- Assignment has a total of 20 marks.

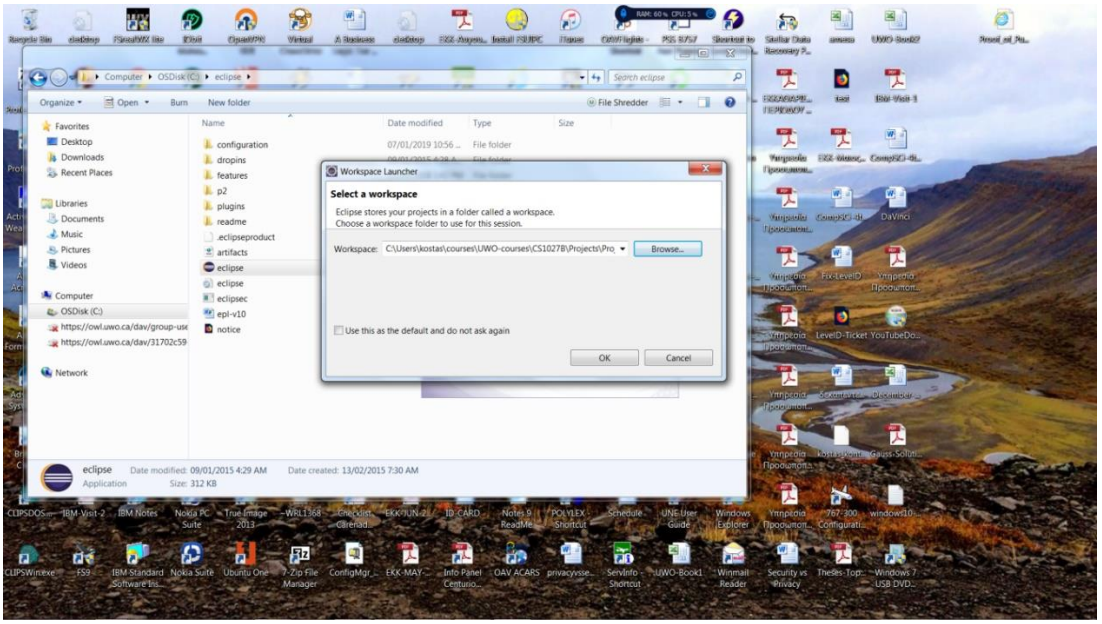
8.1 Marking Rubric

- Program Design and Implementation. All methods in student's java classes are correctly designed and implemented as required: 6 marks.
- Testing. Program produces the correct output for all image operation tests: 10 marks.
- Programming Style: 4 marks
  - Meaningful names for variables and constants.
  - Code is well designed (simple to follow, no redundant code, no repeated code, no overly complicated code, …)
  - Readability:

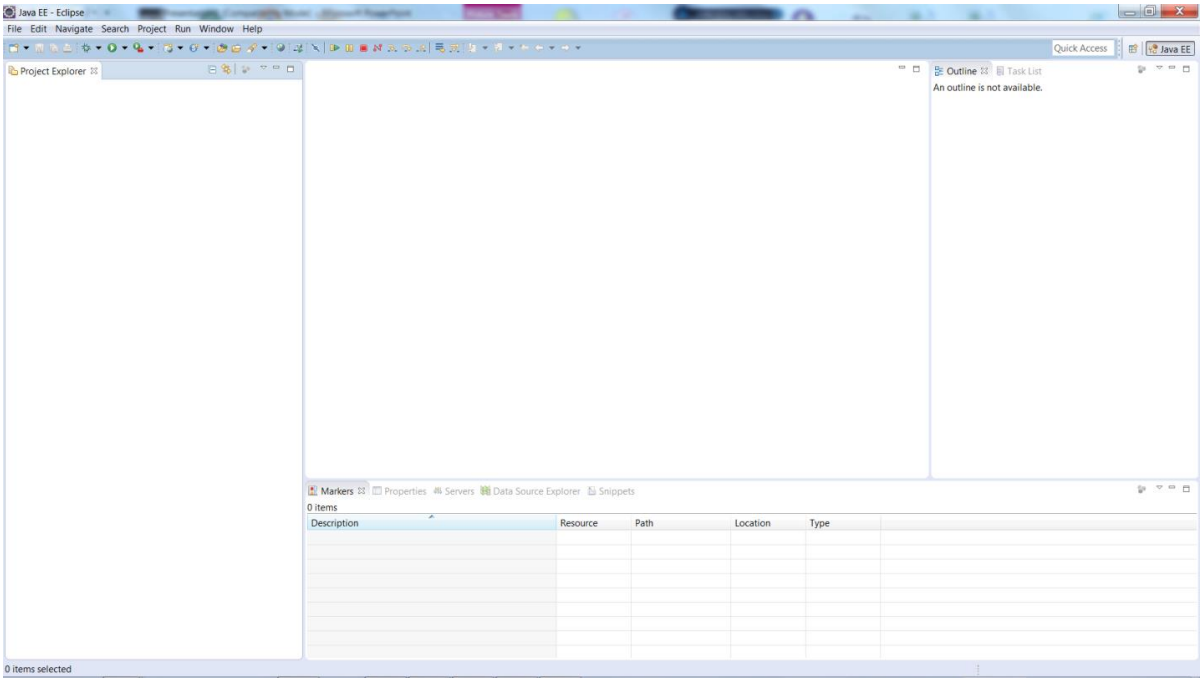
- Good indentation.
- Appropriate code comments.

9. Loading and Running the Program from Eclipse

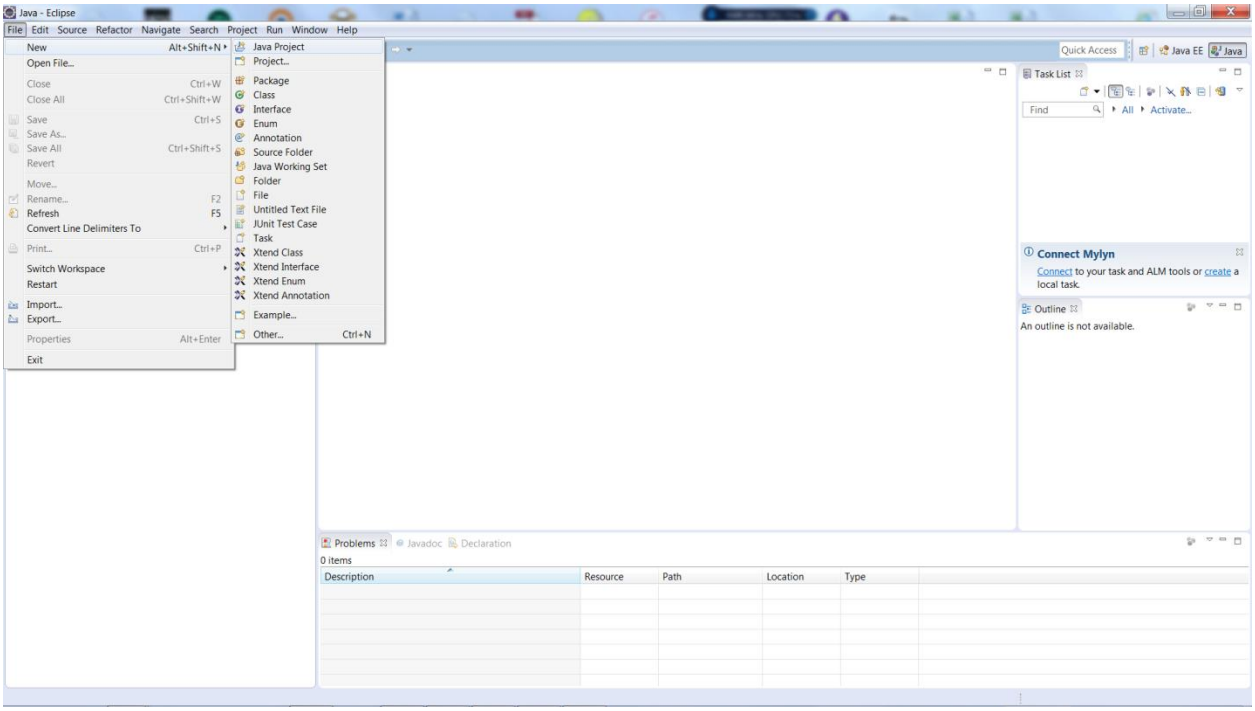
Step 1. Start Eclipse and create a new workspace or use an existing one.

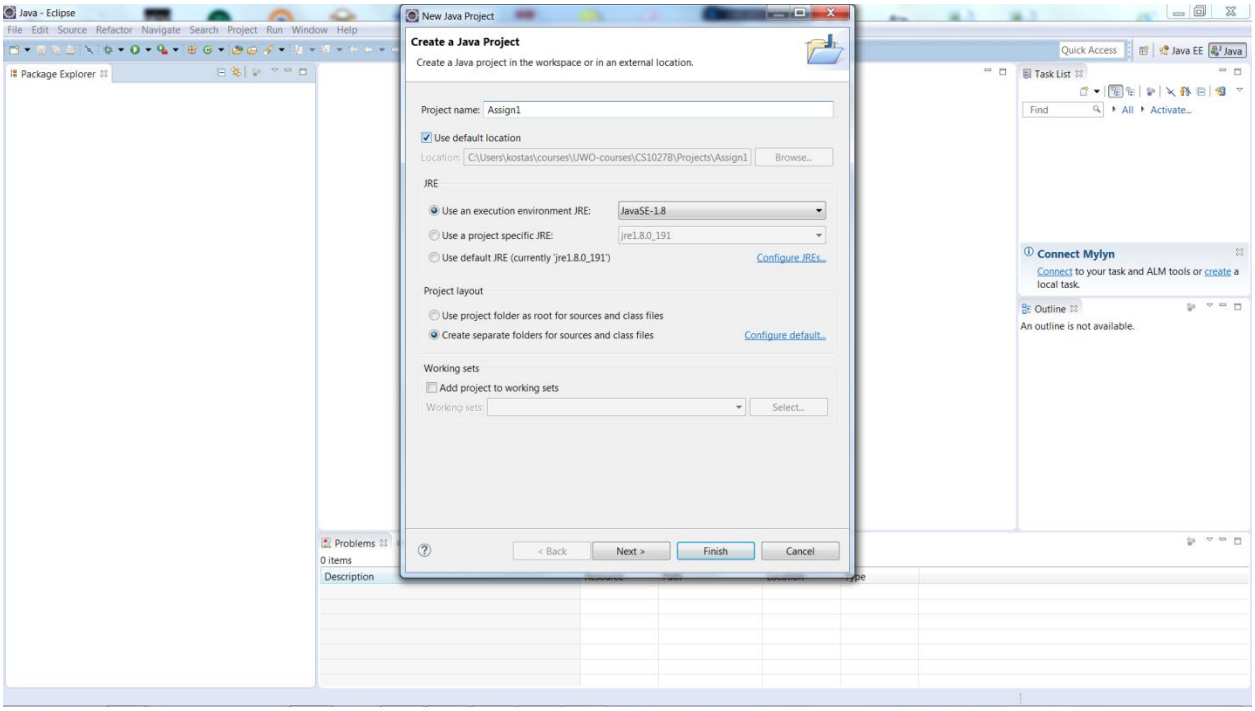


A new workspace will be created.

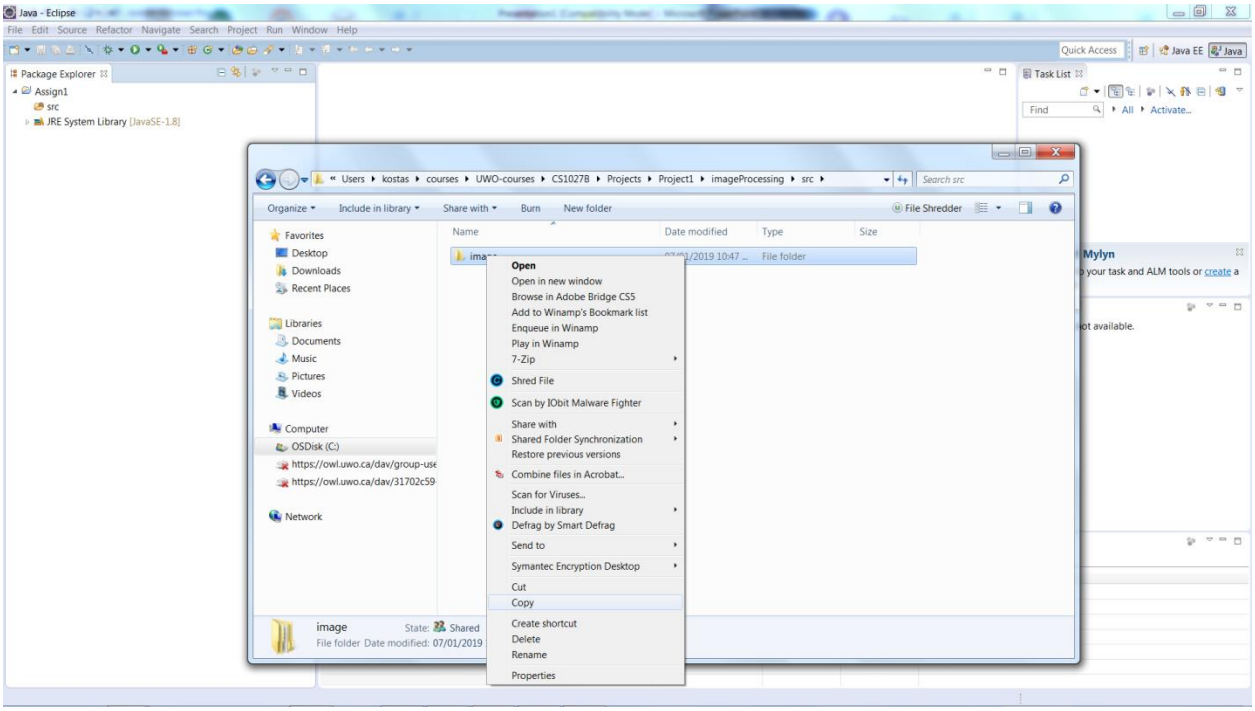


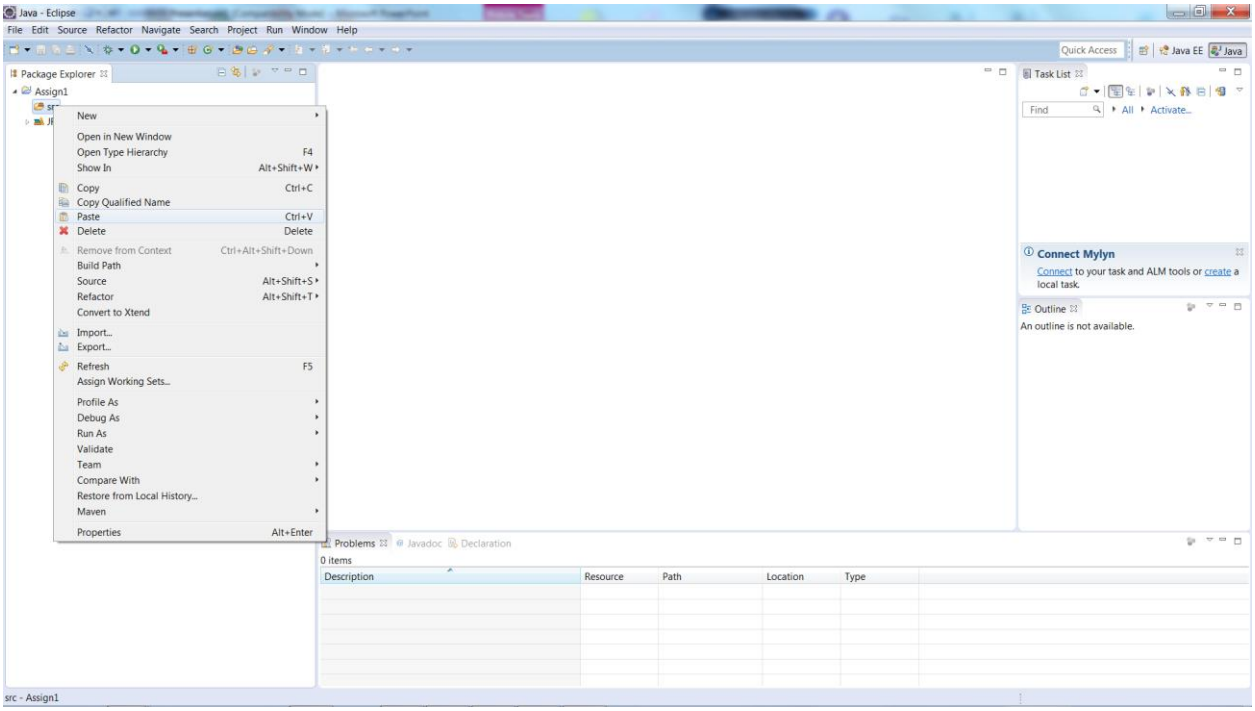
Step 2. Create new Java project (here is called Assign1).



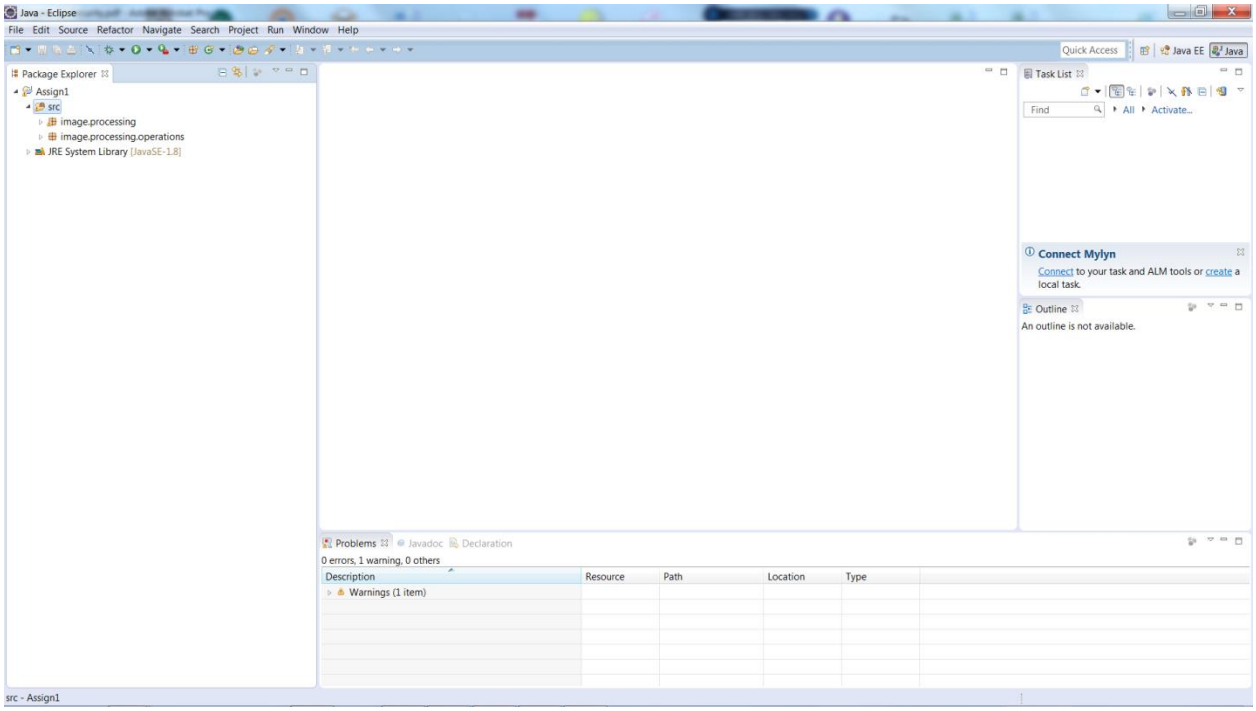


Step 3. Import the sample code you are given. Go to your Windows Explorer and copy/past the source files to Eclipse.

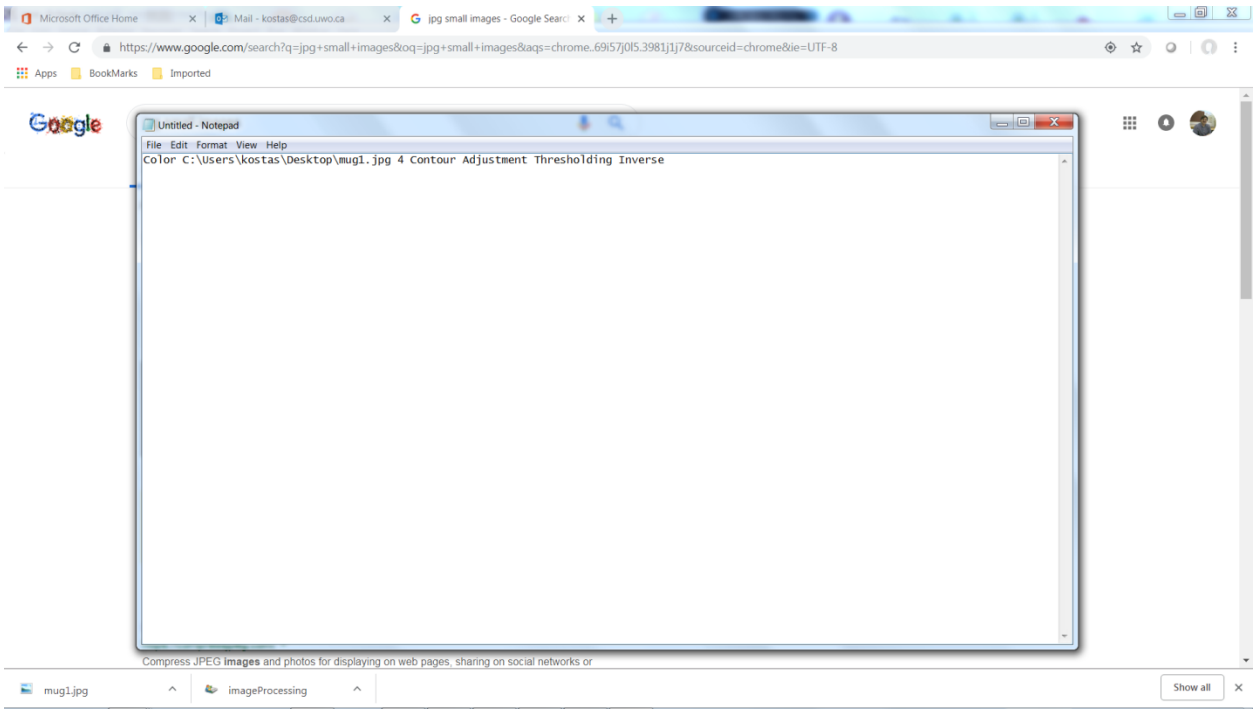




Your code is imported to Eclipse

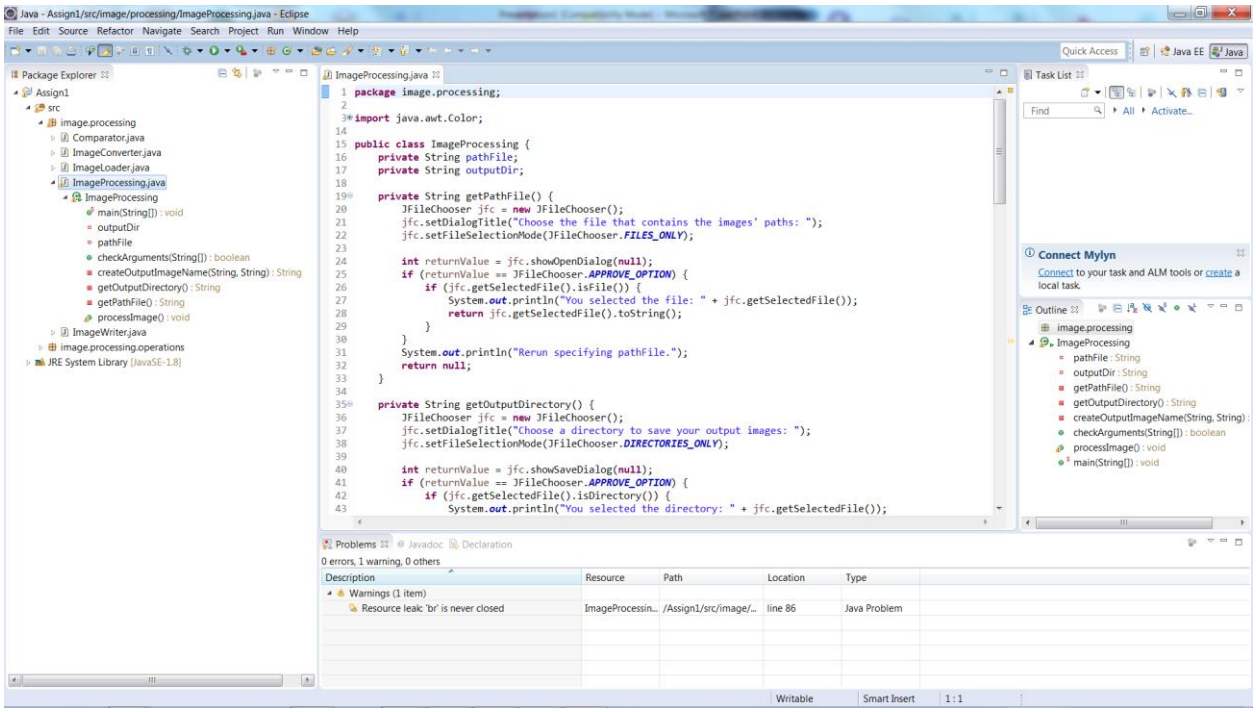


Step 4. Create a .txt configuration file to run the program.

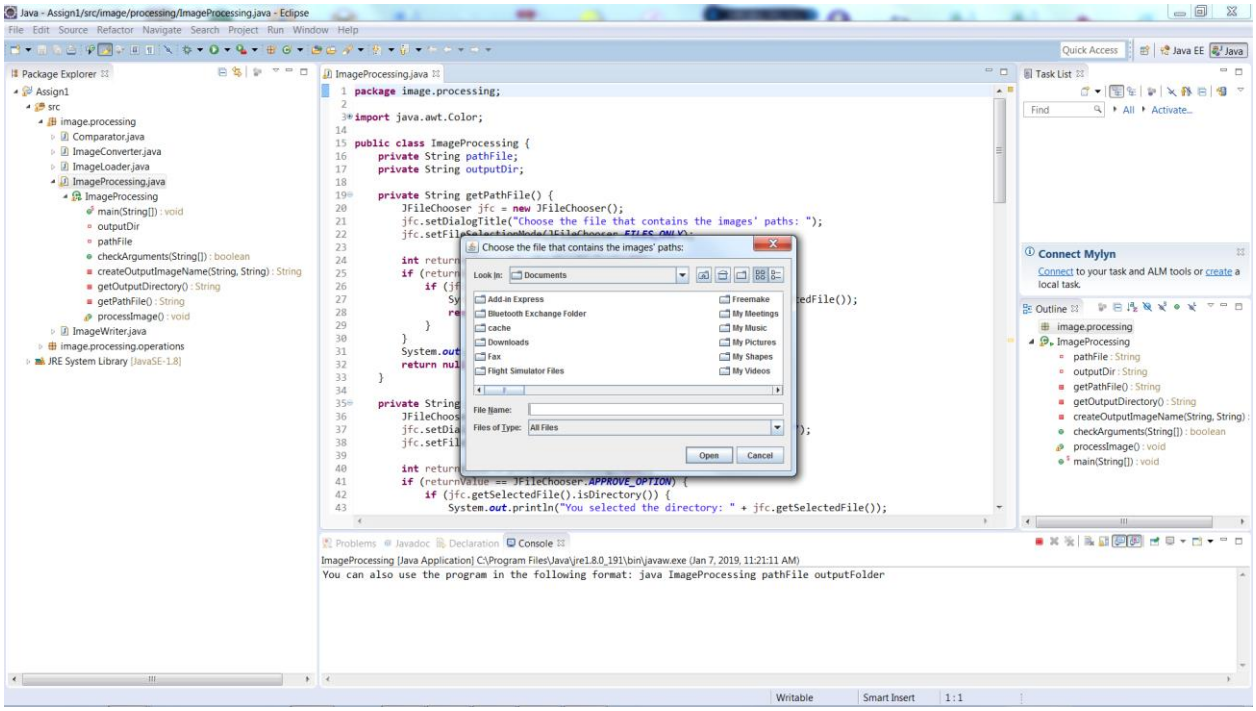


Step 5. Run the program. Select the ImageProcessing.java file (this has the main method) and press the green arrow on the menu to run the application. Since at first you have only the inverse operation in the code, your configuration file might look like this (replace the file name with your path):

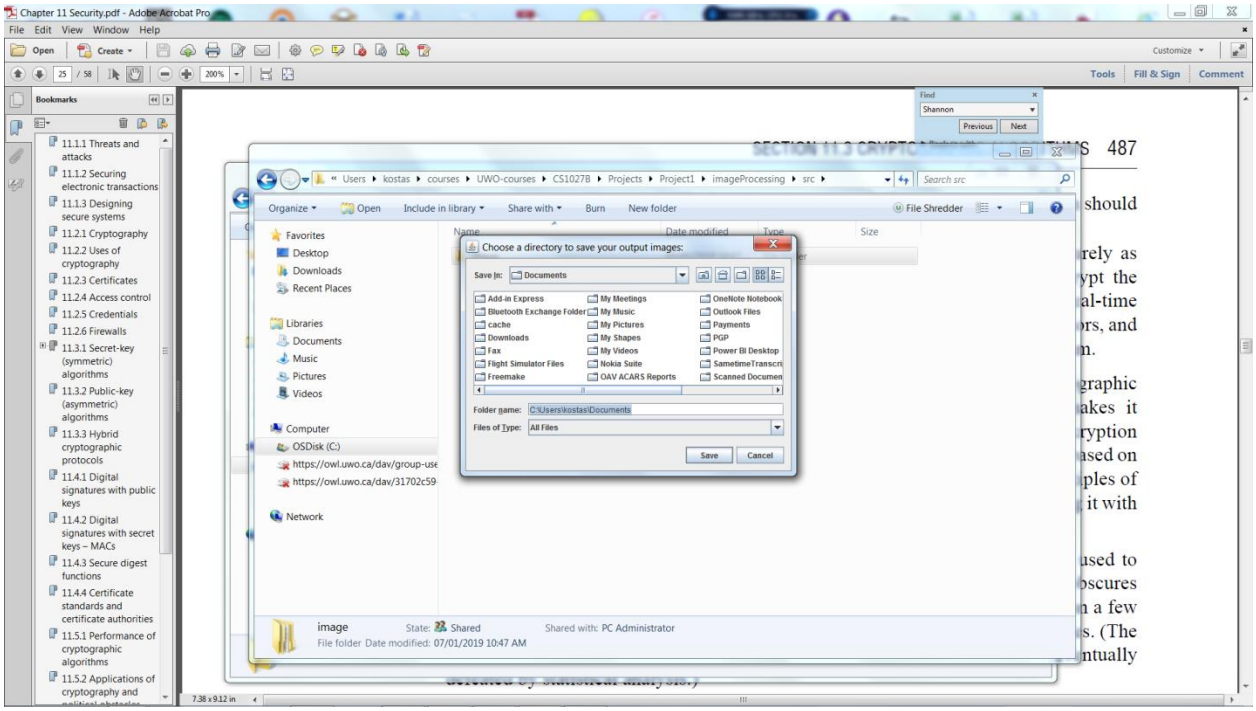
Color C:\Users\kostas\Desktop\mug1.jpg 1 Inverse



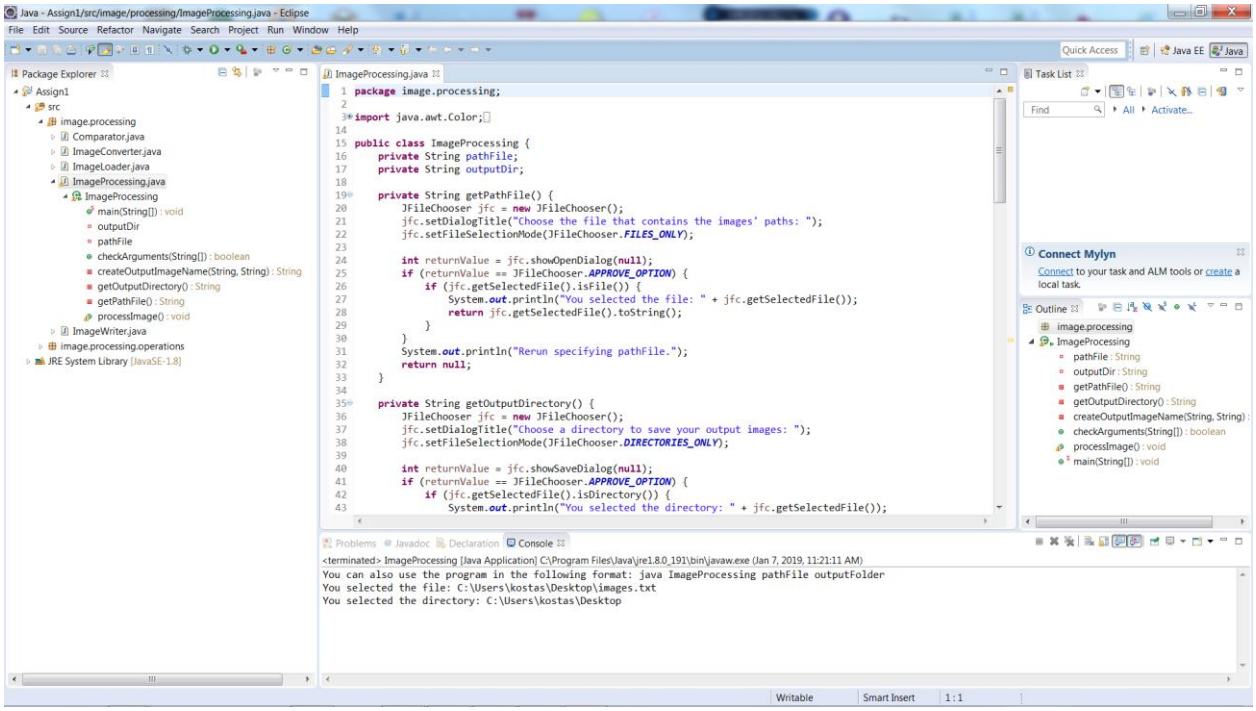
Step 6. Select the input configuration file.



Step 7. Select the output directory.



Step 8. Program completes.


Java - Assign1/src/image/processing/ImageProcessing.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Assign1
src
image.processing
Comparator.java
ImageConverter.java
ImageLoader.java
ImageProcessing.java
ImageProcessing
main(String[]) : void
outputDir
pathFile
checkArguments(String[]) : boolean
createOutputImageName(String, String) : String
getOutputDirectory() : String
getPathFile() : String
processImage() : void
ImageWriter.java
image.processing.operations
JRE System Library [JavaSE-1.8]
ImageProcessing.java
1 package image.processing;
3 import java.awt.Color;
15 public class ImageProcessing {
16 private String pathFile;
17 private String outputDir;
19 private String getPathFile() {
20 JFileChooser jfc = new JFileChooser();
21 jfc.setDialogTitle("Choose the file that contains the images' paths: ");
22 jfc.setFileSelectionMode(JFileChooser.FILES_ONLY);
24 int returnValue = jfc.showOpenDialog(null);
25 if (returnValue == JFileChooser.APPROVE_OPTION) {
26 if (jfc.getSelectedFile().isFile()) {
27 System.out.println("You selected the file: " + jfc.getSelectedFile());
28 return jfc.getSelectedFile().toString();
29 }
30 }
31 System.out.println("Rerun specifying pathFile.");
32 return null;
33 }
35 private String getOutputDirectory() {
36 JFileChooser jfc = new JFileChooser();
37 jfc.setDialogTitle("Choose a directory to save your output images: ");
38 jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
40 int returnValue = jfc.showSaveDialog(null);
41 if (returnValue == JFileChooser.APPROVE_OPTION) {
42 if (jfc.getSelectedFile().isDirectory()) {
43 System.out.println("You selected the directory: " + jfc.getSelectedFile());
Task List
Find
All
Activate...
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Outline
image.processing
ImageProcessing
pathFile : String
outputDir : String
getPathFile() : String
getOutputDirectory() : String
createOutputImageName(String, String) :
checkArguments(String[]) : boolean
processImage() : void
main(String[]) : void
Problems
Javadoc
Declaration
Console
<terminated> ImageProcessing [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (Jan 7, 2019, 11:21:11 AM)
You can also use the program in the following format: java ImageProcessing pathFile outputFolder
You selected the file: C:\Users\kostas\Desktop\images.txt
You selected the directory: C:\Users\kostas\Desktop
Writable
Smart Insert
1 : 1