

Java Picture Processing

COMP40009 – Computing Practical 1

4th – 8th March 2024

Aims

- To gain more experience writing Java code by writing simple programs and designing classes in Java.
- To introduce some algorithms for image processing.

So far this term the lab exercises have been entirely in Kotlin, or in a combination of Kotlin and Java. To give you additional experience working with Java this is a Java only lab. The Java code you will write will build on what was shown in the lectures, but you should consult online resources and library books to research additional Java features as needed.

At various points during this exercise you might find it useful to use *static* methods and fields. A static method in Java is like a top-level function in Kotlin: you do not call it on a particular object; instead you simply call the method and pass it the parameters that it requires. However, because all methods in Java must be declared inside a class or interface, a static method has to be defined inside a class or interface. If static method m is defined in a class C then to call m you write $C.m(\dots)$. A static field in Java is a bit like a top-level property in Kotlin: it does not belong to any specific object, but instead declares a piece of data that can be used by the whole program. Again, a static field must be declared inside a class in Java (static fields cannot belong to interfaces with the exception of *final* static fields). If static field f is defined in class C , you can access f by writing $C.f$.

Aside: A static method of a class has private access to the fields and methods of instances of the class. The analogous concept in Kotlin is a *companion object*.

Problem

- You will be given a skeleton project with several helper classes, a sample test suite, and several test images.
- Your task is to implement several picture transformations, and provide a command line program that allows a user to transform a specified image, saving the resulting image to a file.
- You will also need to work with the given test suite by adding extra test cases for the parts of your program that it does not currently test.

Colours and Pictures

- An image can be represented in memory as a bounded two-dimensional array of pixel values. A *colour-model* is used to translate a pixel-value to colour components. In this lab,

pixel-values will be interpreted using the RGB colour-model, so that each point within an image is mapped on to a red, green and blue component. These components are encapsulated in the provided class `picture.Color` which provides get and set methods for each primary colour. Each component has 256 possible intensities, ranging from 0 to 255. The final colour of each pixel depends on the intensities of the primary colour components. The coordinates (x,y) always mean “along and down”, counting from $(0,0)$ at the *top left*.

- You will be given three classes to use during this lab: `picture.Color`, `picture.Picture` and the empty `picture.PictureProcessor`.

`picture.Color`

The class `picture.Color` provides the following methods for inspecting (*getting*) the colour components of a pixel:

```
public int getRed()
public int getGreen()
public int getBlue()
```

`picture.Picture`

The class `picture.Picture` defines the following methods for manipulating and querying images:



- `public int getWidth()` returns the width of the picture
- `public int getHeight()` returns the height of the picture
- `public Color getPixel(int x, int y)` returns the colour of the pixel-value located at (x,y) .
- `public void setPixel(int x, int y, Color rgb)` updates the pixel-value at the location specified.
- `public boolean contains(int x, int y)` returns `true` iff the specified point lies within the boundaries of the picture.
- `public Picture(int width, int height)` creates a new instance of a `Picture` object of the specified width and height, using the RGB colour model.
- `public Picture(String filepath)` creates a new instance of a `Picture` object from the the picture at the specified location. The location can either be a relative filesystem location (e.g. `"images/red64x64.png"`), or an absolute URL (e.g. `"https://www.doc.ic.ac.uk/~kgk/hello.png"`)
- `public void saveAs(String filepath)` saves the current `Picture` object to the filesystem location in destination. If anything goes wrong while saving, this method throws a `RuntimeException`.
- `public String toString()` overrides the default `toString()` method in order to created a `String` representation of the colours within the current `Picture` object, which may be helpful for debugging purposes.

Picture Transformations

You will need to implement the following picture transformations:

Invert

The invert transformation inverts the colour components of each pixel in the given picture. A colour component may be inverted by replacing the original intensity value of each primary colour, c , with the intensity $(255 - c)$.





Example:  inverts to 

Grayscale

The grayscale transformation creates a monochrome version of the input picture. Gray values, under the RGB colour model, are defined when the values for red, green and blue are equal. A ‘gray’ value can be computed by first finding the average, avg , of the three colour components, then creating a new colour with components $\{\text{red}=avg, \text{green}=avg, \text{blue}=avg\}$. Note that you should use integer division (by 3 in this case) freely without worrying about rounding errors.




Rotate

The rotate transformation creates a picture that is rotated by **90**, **180** or **270** degrees clockwise about the picture’s centre. The angle of rotation will be specified as a command-line parameter to your program.

Example:  rotates 90, 180, 270 to , , 

Flip

The flip transformations rotate a picture about an axis. ‘Flip horizontal’ reflects the image about the y-axis, while ‘flip vertical’ mirrors the image about the x-axis. The direction of reflection, horizontal (**H**) or vertical (**V**), will once again be specified as a command-line parameter.

Example:  flips H, V to , 

Blend

The blend transformation takes a *list* of pictures and combines them together so they appear to be layered on top of each other. The resulting picture will have dimensions corresponding to the *smallest* individual width and individual height within the given set of pictures. A blended pixel is computed by finding the average colour component of each pixel across the list of pictures at any point (as before, use integer division to calculate the average). The list of pictures will be passed as arguments to your program.



Blur

The blur transformation creates a blurred version of the input picture. A blurred-pixel-value is computed by setting its pixel-value to the average value of its surrounding ‘neighbourhood’ of pixels. For example, the average of the neighbourhood:

$$\text{new value for } e = \text{average} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \frac{\sum\{a,b,c,d,e,f,g,h,i\}}{9}$$

Boundary pixels, where a 3x3 neighbourhood is not defined, should not be changed. As with grayscale, you should use integer division when computing the average.



What to Do

Clone your exercise skeleton repository (remembering to replace *username* with your username) using:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab2324_spring/javapictureprocessing_username.git
```

`PictureProcessor.main`

You should implement the `PictureProcessor.main` method to produce the transformations given above. Your program will be invoked with command line arguments specifying which operation to perform, and the input and output image locations. These arguments will be passed as the `args` array to the `main` method.

The format of these commands are:

```
invert <input> <output>
grayscale <input> <output>
rotate [90|180|270] <input> <output>
flip [H|V] <input> <output>
blend <input_1> <input_2> <input_...> <input_n> <output>
blur <input> <output>
```

So, for example, if the `args` array contained:
`{"rotate", "90", "images/green64x64doc.png", "/tmp/test.png"}`, then your program should rotate `images/green64x64doc.png` by 90 degrees, and then save the result in `/tmp/test.png`.

You are free to alter `PictureProcessor.main`, and add any additional packages or classes you wish under the `src` directory. `PictureProcessor.main` will need to use the methods defined in `picture.Picture` to actually load and save `Picture` objects.

A suggested design is to add new methods in the `picture.Picture` class to perform the transforms. For example, the instance method `public void invert()` will invert the image, and `public void flipHorizontal()` will flip it horizontally. `PictureProcessor.main` will focus on parsing the input arguments, and `picture.Picture` will focus on actually performing the transformation.

Running the Program

If you are using an IDE such as IDEA, you can run your program within it. However you might find it easier to run your program from the command line for interactive testing.

IDEA will use Gradle to automatically compile your code for you, and place the resulting class files in a directory called **build/classes/java/main** (which it will also create for you). If you are in the `javapictureprocessing` directory, you can invoke your compiled program from the command line with the following command:

```
% java -ea -cp build/classes/java/main picture.PictureProcessor
TODO: Implement main
```

Here, the `-cp build/classes/java/main` flag tells Java to look for `.class` files in the `build/classes/java/main` directory (`cp` is short for *class path*).

Remember, any extra arguments will be passed to your `args` argument in `main`. So to reproduce the example from earlier, one would invoke:

```
% java -ea -cp build/classes/java/main picture.PictureProcessor rotate 90
images/green64x64doc.png /tmp/test.png
```

Testing

A test suite to help you evaluate your solution is provided via the `PictureProcessorTest` class.

We have provided you with a static helper method, `runMain` in `TestSuiteHelper.java`, which will execute your `main` method with the arguments provided, and then append an extra argument for the output file. However the first argument to `runMain` must be `tmpFolder`, which is a special variable used to allow `runMain` to create a temporary folder (let's call it `/tmp/blah/`), in which to store the output image your program should create.

For example, the first test, `invertBlack()`, calls:

```
Assert.assertEquals(
    new Picture("images/white64x64.png"),
    TestSuiteHelper.runMain(tmpFolder, "invert", "images/black64x64.png"));
```

This says that the image produced in `/tmp/blah/out.png` when `main` is invoked with the arguments `invert images/black64x64.png /tmp/blah/out.png` should be the same as `images/white64x64.png`.

You will notice that this test suite is not complete (for example, it tests flipping `flipVGreen` vertically, but not horizontally. You should add extra test cases to the `TestSuite` class to test other combinations. The `images` directory contains images you can use (the filenames of the images should hint at how they were produced), or you can produce your own.

Suggested Extensions

Mosaic

The mosaic transformation takes a *list* of pictures and combines them together to create a mosaic. The mosaic transform takes an integer parameter, **tile-size**, which specifies the size of a single square mosaic tile. The output picture will have dimensions corresponding to the *smallest* individual width and individual height within the set of specified pictures, *trimmed to be a multiple of the tile-size*.

The tiles in the picture are arranged so that for every tile, the neighbouring tiles to the east and south come from the next picture in the list, (wrapping round as appropriate). The top-left tile comes from the first picture. e.g. Consider making a mosaic of pictures *a*, *b* and *c* (of different sizes, where *a* is 3 tiles wide by 3 tiles high, *b* is four tiles wide by 3 tiles high, and *c* is four tiles wide by four tiles high):

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_4 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix} + \begin{pmatrix} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \\ b_9 & b_{10} & b_{11} & b_{12} \end{pmatrix} + \begin{pmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \\ c_{10} & c_{11} & c_{12} \end{pmatrix} = \begin{pmatrix} a_1 & b_2 & c_3 \\ b_5 & c_5 & a_6 \\ c_7 & a_8 & b_{11} \end{pmatrix}$$



Rewriting your code in Kotlin

To gain a deeper understanding of the differences between Kotlin and Java, you could try rewriting your code using Kotlin. You might want to try out the IntelliJ support for automated Java → Kotlin conversion, and then inspect the resulting Kotlin code to see whether you like its style or whether you think it could be improved.

Submission

Once again, the autotester will run your submitted **PictureProcessorTest.java** test suite against your submitted **PictureProcessor.java** file. It is up to you to include test cases that will convince your PPT Tutor that your solutions are correct. A solution with no test cases to show that it works, may as well be a solution that does not work at all! Please make sure that you maintain the provided directory structure, in order for the tests to work correctly.

Use `git add`, `git commit` and `git push` to send your work to the GitLab server. Then, log into LabTS, click through to your **Java Picture Processing** exercise, request an autotest of your submission and ensure it is correct before submitting to Scientia.

Assessment

- F - E: Very little to no attempt made.
Submissions that fail to compile cannot score above an E.
- D - C: Implementations of most functions attempted;
solutions may not be correct, or may not have a good style.
- B: Implementations of all functions attempted, and solutions
are mostly correct. Code style is generally good.
- A: There are no obvious deficiencies in the solution or
the student's coding style. In addition, there is
evidence of productive testing.
- A*: As for an A -- plus the student has done additional work
beyond the basic spec, e.g. by considering (and clearly
commenting) interesting variations or extensions to the
given functions; e.g. based on their own research.