Java Picture Processing

161 (C1), Integrated Programming Laboratory

19th – 26th January 2015

Aims

- To practice writing simple programs and designing classes in Java.
- To introduce the use of *packages* in Java.
- To introduce the unit testing framework *JUnit*.

Problem

- You will be given a skeleton project with several helper classes, a partially completed 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 to transform a specified image, saving the resulting image to a file.
- You will also need to complete a partially 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 helper classes to use during this lab: `picture.Color`, `picture.Picture` and `picture.Utils`.

`picture.Color`

The class `picture.Color` provides the following methods for inspecting and setting the colour components of a pixel:

```
public int getRed()
public int getGreen()
public int getBlue()
public void setRed(int red)
public void setGreen(int green)
public void setBlue(int blue)
```

`picture.Picture`

The class `picture.Picture` defines the following interface 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.

`picture.Utils`

The class `picture.Utils` provides a set of *static* methods to create, load and save `picture.Picture` objects. For testing purposes, there are some images provided within the `images` directory of the skeleton project, but you can always find or create new images yourself.

- `public static Picture createPicture(int width, int height)` creates a new instance of a `Picture` object of the specified width and height, using the RGB colour model.
- `public static Picture loadPicture(String locationString)` creates a `Picture` object from the the picture at the specified location. The location can either be a filesystem location (e.g. `"images/red64x64.png"`), or a URL (e.g. `"http://www.doc.ic.ac.uk/~mvalerae/linux.jpg"`)
- `public static boolean savePicture(Picture picture, String destination)` saves the given `Picture` to the filesystem location in destination. If anything goes wrong while saving, this method returns `false`, otherwise it returns `true`.
- `public static String toArray(Picture picture)` creates a `String` representation of the colours within the given `Picture` 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 $(255 - c)$.

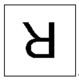
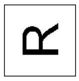
Example:  inverts to 

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 {red=*avg*, green=*avg*, blue=*avg*}. Note that you should use integer division (by 3 in this case) freely without worrying about rounding errors.

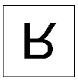
Rotate

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 , , 

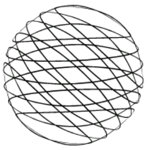
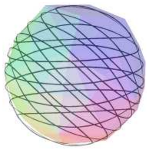
Flip

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 be specified as a command-line parameter to your program.

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.

Example:  and  blend to 

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:

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

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.

Example: 

blurs to 

Submit Monday January 26th 2015

What to Do

Clone your exercise repository (remembering to replace *login* with your login) with:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab1415_spring/javapictureprocessing_login.git
```

If you list all the files in the newly checked out repository, you should see the following:

```
% ls -a picture_processing/
.  ..  .git  .gitignore  images  src  testsuite
```

These are:

- `.`, `..` – these refer to the directory `ls` was run on (`picture_processing`), and its parent directory.
- `src` – the directory in which the Java source files for the exercise reside. The `src` directory contains two further directories (`picture` and `utils`), which correspond to two Java *packages*. For example, the *class* `picture.Picture` can be found in `src/picture/Picture.java`, where `picture` is the name of the package.
- `testsuite` – this directory contains Java source files corresponding to a JUnit 4 test suite for the exercise. These will be discussed further under **Testing**, below.
- `images` – this directory contains some images which you may wish to use for testing.

- `.git`, `.gitignore` – these contain the git repository information, and a list of filename patterns that should be ignored by git, respectively.

This term, we strongly recommend you use an Integrated Development Environment (IDE) to develop and debug your Java labs. Under the lab notes for this course you will find a short document introducing Eclipse, one such IDE.

`picture.Main`

You should implement the `picture.Main` class to implement 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 `picture.Main`, and add any additional packages or classes you wish under the `src` directory. `picture.Main` will need to use the static methods exported by `picture.Utils` to actually load and save `picture.Picture` objects.

A suggested design is to create a new class, `picture.Process`. This will accept the input `picture.Picture` as a constructor argument, and have instance methods that perform the transforms on the contained picture. For example, the instance method `public void invert()` will invert the image, and `public void flipHorizontal()` will flip it horizontally. `picture.Main` will focus on parsing the input arguments, and `picture.Process` will focus on actually performing the transformation.

Running the Program

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

Eclipse will automatically compile your code for you, and place the resulting class files in the directory `bin` (which it will also create for you). If you are in the `picture_processing` directory, you can invoke your program from the command line with the following command:

```
% java -ea -cp bin picture.Main
TODO: Implement main
```

Here, the `-cp bin` flag tells Java to look for `.class` files in the `bin` 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 bin picture.Main rotate 90 images/green64x64doc.png /tmp/test.png
```

You can also manually compile `Main` from the command line and put the results in `bin` with the following command:

```
% javac -g -d bin -sourcepath src src/picture/Main.java
```

However you shouldn't need to do this if you are working with an IDE.

Testing

This term we will be using the *JUnit* library as a standard way to test the Java exercises. In the `testsuite` source directory, a partial test suite suitable for testing your program has been set up.

If you look at `testsuite.TestSuite` you will see 6 methods that have been *annotated* by `@Test`. The `@Test` is a Java annotation, which you will learn more about later in your Java course. Here, they tell JUnit that the following method represents a test. The `assertEquals` method is also part of JUnit, which checks that the two arguments it is passed are equal, otherwise it makes the test fail.

We have provided you with a static helper method, `runMain`, 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 (call it `/tmp/blah/`) in which to store the output image your program should create.

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

```
assertEquals(Utils.loadPicture("images/white64x64.png"),
  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`.

On the command-line, you can compile the test suite with:

```
javac -g -d bin -cp /usr/share/java/junit4.jar -sourcepath src:testsuite
  testsuite/testsuite/TestSuite.java
```

which instructs `javac` to place the compiled `.class` files in the `bin` directory, to use the JUnit 4 jar file, and to look for source files in both the `src` and `testsuite` directories.

And then run it with:

```
java -cp /usr/share/java/junit4.jar:bin org.junit.runner.JUnitCore
  testsuite.TestSuite
```

Alternatively, you can compile and run the test-suite very easily, with a more flexible visualisation of the errors, using an IDE (see the lab notes for specific Eclipse instructions).

You will notice that this test suite is not complete (for example, it tests flipping vertically (`flipVGreen`), but not horizontally. You should add extra test cases to the `testsuite.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.

Submission

As with the previous exercises, you will need to use the commands `git add`, `git commit` and `git push` to send your work to the gitlab server. Then, log into the LabTS server `https://teaching.doc.ic.ac.uk/labts`, and click through to your `javarecursion` exercise. Request an auto-test of your submission and ensure it is correct before downloading and submitting the relevant `cate_token.txt` to CATe.

Remember to ensure that you submit the correct `cate_token.txt` file to CATe. Web browsers tend to rename duplicate files to e.g. `cate_token.txt (1)`.

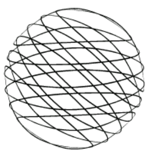
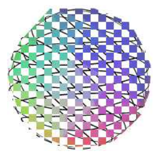
Suggested Extensions

Mosaic

The mosaic transformation takes a *list* of pictures and combines them together to create a mosaic. The mosaic transform takes a 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):

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

Example:  ,  , mosaics to 

Assessment

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Some implementation has been attempted, however it may be
       incorrect, incomplete, or severely lacking in good code style.

B - A: A complete implementation has been produced, with a good coding
       style.  However a few edge cases, or harder methods (e.g. blend,
       blur) may be incorrect.

A+:    There are no obvious deficiencies in the solution or
```

the student's coding style. Some testing attempted.

A*: As for an A+, with work beyond the basic spec attempted (e.g a suggested extension, or a transformation of the student's own design).