CS 145/148 Homework 4
Proportable

1 Overview

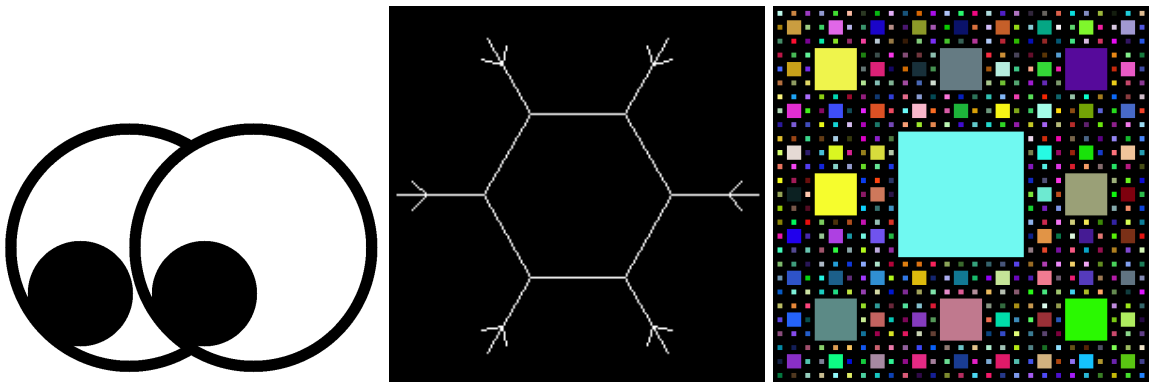
Your objective in this homework is to acquaint yourself with conditional statements and loops, which enable you to write code that diverges and repeats. You will do this in the context of writing an application that draws vector graphics. See Figure 1 for several images that were generated using this application.

Many images are encoded as a 2D matrix or *raster* of pixel colors. The amount of available color information is fixed by the image's width and height—its resolution. But what happens when you scale this fixed resolution image up to a very large area, like a poster? Typically, new colors in the large image are generated from the old ones in the small image by averaging nearby pixels. Because of this averaging, the larger image is often blurry.

A better solution exists. Instead of storing a fixed grid of pixels, the elements of the image are stored in a scalable, parametric fashion. For instance, if a circle is to appear at the center of the image and have a radius of $\frac{1}{4}$ of the image's width, an image file might contain the command:

```
circle 0.5 0.5 0.25
```

The image content is defined using proportional coordinates, which means the image can be scaled to *any* size without a loss of quality. Images that are encoded in this way are called *scalable vector graphics* (SVG).



(a) Eyes (b) Snowflake (c) Squares

Figure 1: Example scalable vector images. Your code in this homework will read in textual descriptions of an image and output raster versions.

2 Requirements

Complete the classes described below. Place all classes in package `hw4`. Make all methods `static`.

2.1 Main

Write a class `Main` with a `main` method, which you are encouraged to use to test your code. Nothing in particular is required of it, but it must exist. Write code in it that tests your methods directly—not just through the SpecChecker. For example, you can test `Proportable.clamp` with a quick call like this:

```
System.out.println("103 clamped to [0, 100]? " + Proportable.clamp(103, 0, 100));
```

Many of the methods you write will generate or edit images. Testing these only through the SpecChecker is a frustrating endeavor. Instead, write these images to a file and manually inspect them to make sure they are functioning correctly. For example, we can test `Proportable.plotRectangle` with code like this, which writes the image to a file named `test.png` in your `Desktop` directory:

```
BufferedImage image = new BufferedImage(200, 100, BufferedImage.TYPE_INT_RGB);
Proportable.plotRectangle(image, 0, 0, 1, 1, Color.YELLOW);
try {
  File desktop = new File(System.getProperty("user.home"), "Desktop");
  ImageIO.write(image, "png", new File(desktop, "test.png"));
} catch (IOException e) {
  throw new RuntimeException(e);
}
```

2.2 Proportable

Write a class `Proportable`, which contains several `static` methods for drawing raster images from scalable, proportion-based descriptions of the image's content. It has the following:

1. Method `blankImage`, which returns a new `BufferedImage` of the given dimensions and with all pixels set to the given color. It accepts three parameters in the following order:

   (a) The image's width, of type `int`

   (b) The image's aspect ratio (the ratio of the image's width to its height), of type `double`

   (c) The color with which to fill the image, of type `Color`

   Solve for the image's height using the given width and aspect ratio. Truncate any remainder (which means cast to an integer, losing any fractional component). For the image's pixel type, use the predefined constant `BufferedImage.TYPE_INT_RGB`.

2. Method `toHorizontal`, which accepts two parameters parameters in the following order:

   (a) An image of type `BufferedImage`

   (b) A proportional x-coordinate of type `double`

   It returns as an `int` the nearest column index in the image that corresponds to the proportional coordinate. The proportion is applied to the image's width, with proportion 0 mapping to the leftmost column of pixels, and proportion 1 mapping to the rightmost column of pixels, whose index is one less than the width. For example, if an image has width 20, the following values should be produced for various proportions:

- `toHorizontal(image20, 0.0)` → 0
- `toHorizontal(image20, 1.0)` → 19
- `toHorizontal(image20, 0.75)` → 14
- `toHorizontal(image20, 0.77)` → 15

3. Method `toVertical`, which accepts two parameters parameters in the following order:

   (a) An image of type `BufferedImage`

   (b) A proportional y-coordinate of type `double`

   It returns as an `int` the pixel coordinate in the image that corresponds to the proportional coordinate. The proportion is inversely applied to the image's height, with proportion 0 mapping to the bottommost row of pixels (whose index is one less than the image's height), and proportion 1 mapping to the top row of pixels, whose index is 0. This inverse mapping has the effect of situating the image's origin in proportional space to the bottom left of the image. For example, if an image has width 30, the following values should be produced for various proportions:

   - `toVertical(image30, 0.0)` → 29
   - `toVertical(image30, 1.0)` → 0
   - `toVertical(image30, 0.75)` → 7
   - `toVertical(image30, 0.77)` → 7

4. Method `clamp`, which accepts three `int` parameters in the following order:

   (a) A number

   (b) An inclusive lower-bound

   (c) An inclusive upper-bound

   It returns as an `int` the number clamped into the given range. That is, the number returned is no lower than the lower-bound and no greater than the upper-bound. If the number is already within the range, it is returned as is. Do not implement this with an `if` statement.

5. Method `plotRectangle`, which plots a rectangle onto an image. It accepts six parameters in the following order:

   (a) The image on which to plot, of type `BufferedImage`

   (b) The x-proportion of the rectangle's bottom-left corner, of type `double`

   (c) The y-proportion of the rectangle's bottom-left corner, of type `double`

   (d) The x-proportion of the rectangle's top-right corner, of type `double`

   (e) The y-proportion of the rectangle's top-right corner, of type `double`

   (f) The color of the rectangle, of type `Color`

Ignore any pixels in the rectangle that do not fall within the image bounds.

Normally you have considerable freedom in how you implement methods, and only the method's interface is mandated. However, so that we may produce comparable results for this method, you will need to implement it in a particular way. First, convert the proportions to indices. Second, clamp the indices to the image's dimensions. Third, visit all pixels in these inclusive integer bounds, setting each to the given color.

6. Method `plotCircle`, which plots a circle onto an image. It accepts six parameters in the following order:

   (a) The image on which to plot, of type `BufferedImage`

   (b) The x-proportion of the circle's origin, of type `double`

   (c) The y-proportion of the circle's origin, of type `double`

   (d) The proportional radius of the circle, of type `double`

   (e) The color of the circle, of type `Color`

   Ignore any pixels in the circle that do not fall within the image bounds.

   As with `plotRectangle`, you will need to implement this method in a particular way. First, convert the circle's proportions to indices. Apply the proportional radius to image's width—not its height. Second, visit all the pixels within the rectangle circumscribing the circle. (The code will be similar to `plotRectangle`.) But only set the color of the pixels whose distance to the center pixel is less than or equal to the integer radius.

7. Method `chessboardDistance`, which computes the so-called *chessboard distance* between two points. It accepts four `int` parameters in the following order:

   (a) Point A's x-coordinate

   (b) Point A's y-coordinate

   (c) Point B's x-coordinate

   (d) Point B's y-coordinate

   Chessboard distance is defined as the shortest number of moves it takes a king in the game of chess to move from its current square to some target square. (On a single move, kings can move to any of its eight neighboring squares. I encourage you to draw some pictures to get a visceral feel for the shortest path between two squares.) More formally, chessboard distance is the maximum of the x-coordinate difference and the y-coordinate difference of the two points. The sign of the difference is ignored. For example, `chessboardDistance(1, 3, 4, 5)` $\rightarrow$ 3—because the differences are 3 and 2, and 3 is larger. `chessboardDistance(9, 30, 1, 23)` $\rightarrow$ 8—because the differences are 8 and 7, and 8 is larger.

8. Method `lerp`, which *linearly interpolates* between two values. It accepts three `double` parameters in the following order:

   (a) a value a

   (b) a value b

(c) a proportion in $[0, 1]$

It returns as a `double` the value between a and b corresponding to the given proportion. A proportion of 0 maps to a. A proportion of 1 maps to b. A proportion of 0.7 maps to a value that is 70% of the way from a to b. For example, `lerp(50, 100, 0.9)` $\rightarrow$ 95.

9. Method `plotLine`, which accepts six parameters in the following order:

   (a) the image on which to plot, of type `BufferedImage`

   (b) Point A's x-proportion, of type `double`

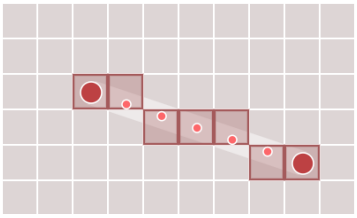
   (c) Point A's y-proportion, of type `double`

   (d) Point B's x-proportion, of type `double`

   (e) Point B's y-proportion, of type `double`

   (f) the color of the line, of type `Color`

   There are several popular line-drawing algorithms. Some are simpler to implement; others are lightning-fast to execute. We will aim for simpler to implement. Consider the following illustration of two points:



   The difference in their x-coordinates is 6, and the difference in y-coordinates is 3. Therefore, the chessboard distance between them is 6, and that is exactly how many pixels must be filled in to plot a line between the two—not including the starting pixel, which must also be filled in. We evenly distribute the pixels between these two points. Pixel 0 is $\frac{0}{6}$ of the way from point A to point B. Pixel 1 is $\frac{1}{6}$ of the way. Pixel 2 is $\frac{2}{6}$. And so on, until we reach pixel 6, which is $\frac{6}{6}$ of the way. We can determine the exact x-coordinate for these pixels by lerping between the two points' x-coordinates using the pixel's proportion. Same for the y-coordinate. Lerping will yield a fractional pixel coordinate, which we round to produce an integer.

   This algorithm can be described in pseudocode:

```
n = compute chessboard distance between two points
for i through n
  p = compute proportion of traversal
  x = lerp and round x-coordinate
  y = lerp and round y-coordinate
  plot pixel (x, y)
```

   Before executing this traversal algorithm, convert the proportional coordinates to real pixel coordinates. If the traversal visits any pixels that are not within the image bounds, ignore them.

10. Method `plot`, which reads a file of drawing commands, plots the described shapes, and returns the resulting `BufferedImage`. It accepts two parameters in the following order:

    (a) the file of drawing commands, of type `File`

    (b) the width of the image to be drawn onto, of type `int`

    The format of the file is described through this example:

```
2.0                     # aspect ratio -- 1 double
1.0 0.5 0               # background color -- 3 floats
color 1 0 0             # plot next shapes in red
rectangle 0 0 0.4 0.3   # plot rectangle in bottom-left quadrant -- (0, 0) to (0.4, 0.3)
color 0 1 0             # plot next shapes in green
circle 0.5 0.5 0.1      # plot circle at center of image filling tenth of width
line 0 0 1 1            # plot line from (0, 0) to (1, 1)
```

    Line breaks, for the most part, are not significant. The first number in the file is the aspect ratio. The next three determine the background color. Next is a series of commands. Commands `rectangle`, `circle`, and `line` correspond to the methods described above. Each is followed by an appropriate sequence of proportions. The `color` command sets the color for subsequent shapes.

    The proportions are all `double`s, but the color intensities are always `float`s. (The two types of numbers don't look any different in the file, but they matter in your code.)

    Processing this file should not require sophisticated parsing. Follow this pattern:

```
while tokens remain
  read token
  process token accordingly, reading more tokens as necessary
```

    If you encounter a `#`, consider the remainder of the line to be a comment, which should not be processed. Just consume and ignore the rest of the input on that line.

3 Extra

For an extra credit participation point, share an SVG file of your own making and its resulting image on Piazza under folder `hw4_share`.

4 Submission

To submit your work for grading:

1. Put the SpecChecker for this homework in your Build Path. Run the SpecChecker as a Java Application and fix problems until all tests pass.

2. Commit and push your work to your repository. Verify that your solution is on Bitbucket.

A passing SpecChecker does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The SpecChecker checks some of them, but not all.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, CS 1 moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.
- Your code must be submitted correctly and on time. Most excuses devolve into, "I started too late." The fix for this problem is not an extension.