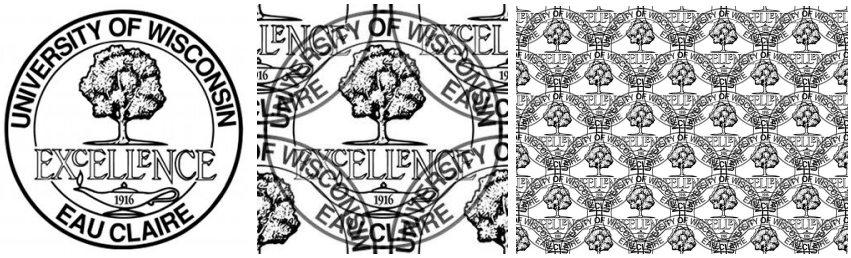
CS 145 Homework 4
Computer as Pilot

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 methods to generate seamless images that tile across virtual terrains or web page backgrounds without tripping up the human eye on discontinuities. See Figure 5 for an example of this tiling.

The algorithm used to make an image seamless works by overlaying on each corner a semi-transparent version of the opposing corner. When this seamless image is laid out side by side with itself, these blended corners will match up with the blended corners of the adjacent images. Consider the example shown in Figure 1.



(a) Source (b) Seamless (c) Seamless tiled 4x3

Figure 1: Tiling of the UWEC seal

2 Requirements

Complete the three 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. We suggest you use it to test your methods directly—not just through the SpecChecker.

To test your methods you'll first need some images saved on your computer. Then, in your `main` method, pop open a graphical open dialog, and process the selected image with something like the following:

```
public static void main(String[] args) throws IOException {
  JFileChooser dialog = new JFileChooser();
  int status = dialog.showOpenDialog(null);
  if (status == JFileChooser.APPROVE_OPTION) {
    File file = dialog.getSelectedFile();
    BufferedImage src = ImageIO.read(file);
    BufferedImage dst = ImageUtilities.swapCorners(src);
```

```
    ImageIO.write(dst, FilenameUtilities.getExtension(file),
                  FilenameUtilities.appendToName(file, "swapped"));
  }
}
```

Note that some of the methods called in this snippet are ones that you are to write. This code won't run until you implement them. The `BufferedImage` class is available in the Java standard library.

If you'd like to generate a blank image programmatically, you can create one through the normal construction idiom:

```
BufferedImage img = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
```

The third parameter specifies the pixel format, which for this assignment could be `TYPE_INT_RGB` (for pixels with red, green, and blue components) or `TYPE_BYTE_GRAY` (for pixels with just a grayscale intensity).

2.2 FilenameUtilities

Write a class `FilenameUtilities`, which contains several `static` methods for manipulating file names. It has the following:

1. Method `getExtension`, which accepts one parameter: a `File`. It returns its extension as a `String`. An extension is the portion of the file's name after the last period—unless the last period is the first character of the `String`, in which case the file has no extension. If the file has no extension, return `null`. For example:

   - `getExtension(new File("honeypot/private/ssn.csv"))` → `"csv"`
   - `getExtension(new File("2015.10/README"))` → `null`
   - `getExtension(new File(".DS_Store"))` → `null`
   - `getExtension(new File("iexplore.exe"))` → `"exe"`

   Consider only the file's name—as provided by a `File`'s `getName` method—when searching for its extension. Leading directory names should be ignored. Do not hardcode any directory-separating slashes in your solution, as different operating systems uses different slashes to separate directories.

2. Method `appendToName`, which accepts two parameters in this order:

   (a) a file whose name is used to derive a new name, of type `File`

   (b) the text to append to the file's name, of type `String`

   It returns a new `File` that's nearly identical to the parameter, but with an underscore (`_`) and the text appended after the name but right before any extension. Consider the following example calls:

   - `appendToName(new File("a.docx"), "copy")` → `a_copy.docx`
   - `appendToName(new File("/home/jugo"), "2")` → `/home/jugo_2`

You'll need to construct a new `File` that has the same directory as the parameter `File`. Use the `File(File parentDirectory, String name)` constructor and the `getParentFile` method to accomplish this.

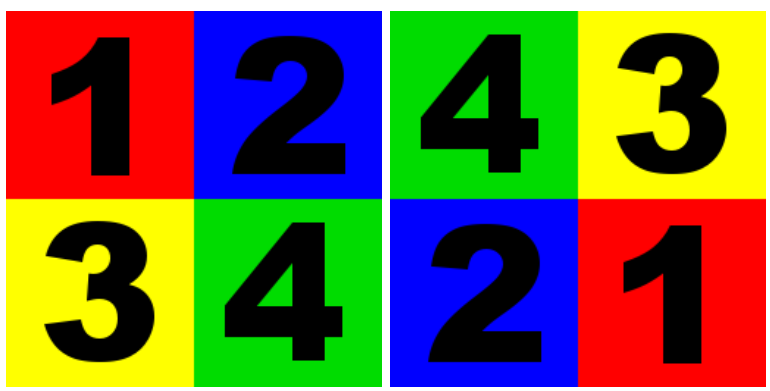
This method will be used in later code to generate the names of files derived from a source file.

2.3 ImageUtilities

Write a class `ImageUtilities`, which contains a number of `static` methods for manipulating `BufferedImage`s. None of these methods modifies any parameter image it is given. It has the following:

1. Method `swapCorners`, which accepts one parameter: a image of type `BufferedImage`. It returns a new `BufferedImage` like the original but with its corners swapped.

   One could write a series of loops that processed each quadrant separately, but there's a more elegant way. First, find the image's half-height and half-width as `int`s, truncating any fractions. Then, march through the pixels of the new image, figuring out the source pixel (co, ro) in the old image that corresponds to each destination pixel (cn, rn) in the new image. The source pixel is the pixel one half-width to the right and one half-height up from the destination pixel. Use `%` to wrap back around in case this offset goes beyond the edge of the image. An example swap is shown in Figure 2. Test this method before moving on.



(a) Our test image... (b) ... with its corners swapped.

Figure 2: Swapping quadrants with their opposites

2. Method `getCircleMask`, which accepts three parameters in this order:

   (a) an width of type `int`

   (b) an height of type `int`

   (c) a power of type `double`

   It returns a grayscale `BufferedImage` containing a radial gradient. The center of the image is white, which fades to black as it approaches the image's edges. Compute grayscale intensity for the pixel at (c, r) with the following computation.

First find the pixel's distance from the image center using the Pythagorean theorem—using `double`s for all calculations:

$$\text{midC} = \frac{\text{width}}{2} \tag{1}$$

$$\text{midR} = \frac{\text{height}}{2} \tag{2}$$

$$\text{distance from center} = \sqrt{(c - \text{midC})^2 + (r - \text{midR})^2} \tag{3}$$

Then *normalize* this distance so the center of the image—where the distance is 0—maps to 0, and the closest edge of the image maps to 1. We do this by dividing by the smallest reach from the center to an edge, yielding the image shown in Figure 3a. That smallest reach we call the image's *radius*:

$$\text{radius} = \min(\text{midC}, \text{midR}) \tag{4}$$

$$\text{normalized distance} = \frac{\text{distance from center}}{\text{radius}} \tag{5}$$

If the pixel lies on this radius or beyond, we assign it black. For pixels within the radius, their normalized distance grows linearly from 0 at the center to 1 at the edge. We can change the rate of increase by raising the normalized distance to the given power, yielding the image shown in Figure 3b:

$$\text{tweaked distance} = \text{normalized distance}^{\text{power}} \tag{6}$$

This tweaked distance is still 0 at the center and 1 at the nearest edge. We want the opposite, with the center white and the nearest edge black. So, we take its complement to arrive at our desired grayscale intensity, yielding the image shown in Figure 3:

$$\text{gray} = 1 - \text{tweaked distance} \tag{7}$$

Compute the grayscale intensity as a `float`. To write this intensity into our image, check out `BufferedImage`'s `setRGB` method. Notice that it expects an `int` representation of the `Color`. How do we turn our `float` into an `int`? Feed it three times to the `Color` constructor that accepts three `float`s, replicating the intensity for red, green, and blue components. Once you have a `Color`, use its `getRGB` method to turn it into an `int` representation. Test this method before moving on.

3. Method `mix`, which accepts three parameters in this order:

   (a) a value a of type `int`

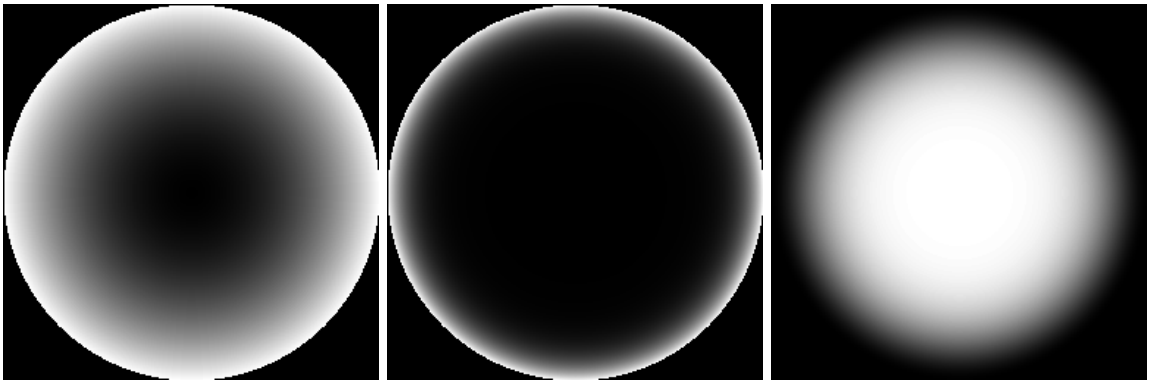
   (b) a value b of type `int`

   (c) a weight in $[0, 1]$ of type `double`

   It returns as an `int` the weighted blend of a and b. For example, if the weight is 0.7, return 70% of a mixed with 30% of b. Follow this formula so that we produce the same results:

$$\text{weight} * a + (1 - \text{weight}) * b \tag{8}$$

   Truncate any fraction to arrive at your final `int`; don't ever round up, or you may exceed the maximum color intensity of 255. Test this method before moving on.



(a) Normalized distance (b) Raising to a power (c) Complementing

Figure 3: Stages of constructing a circle mask

4. Method `mix`, which accepts three parameters in this order:

   (a) a color a of type `Color`

   (b) a color b of type `Color`

   (c) a weight in $[0, 1]$ of type `double`

   It returns a new `Color` that is a weighted mix of a and b. For example, consider this code:

```
Color a = new Color(100, 200, 0);
Color b = new Color(110, 190, 20);
Color ab = mix(a, b, 0.8);
```

   Here, `ab` is the color (102, 198, 4).

   To produce the new `Color`, use your `mix(int, int, double)` to individually mix the `Color`s' red, green, and blue components. Test this method before moving on.

5. Method `addMasked`, which accepts four `BufferedImage` parameters in this order:

   (a) image a of type `BufferedImage`

   (b) a grayscale mask to apply to image a, of type `BufferedImage`

   (c) image b of type `BufferedImage`

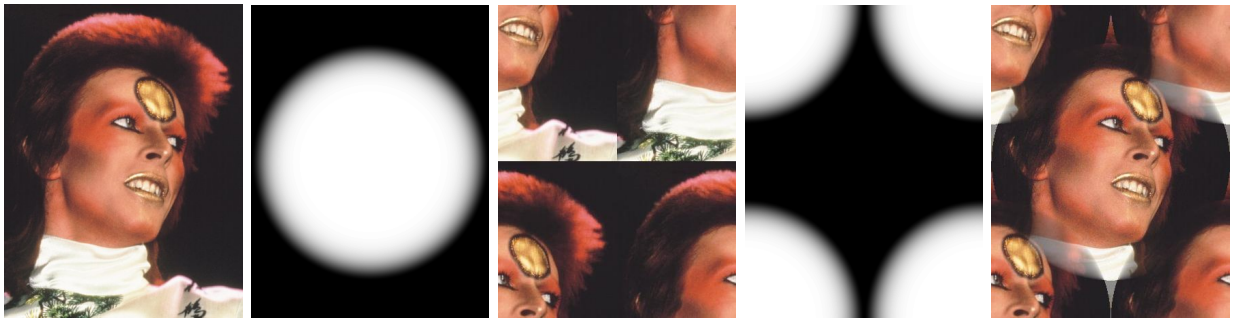
   (d) a grayscale mask to apply to image b, of type `BufferedImage`

   Assume that all images are the same size.

   A mask can be thought of as a coefficient image—an image that is multiplied by another. It is used to scale the colors of its partner image. Where a mask is white, the partner image will appear in full intensity. Where a mask is black, the partner image will diminish to black.

   This method creates a new blended image of the same dimensions as the parameter images. Roughly, we blend the images using the following mathematical formulation:

$$\frac{\text{maskA}}{\text{maskA} + \text{maskB}} \cdot a + \left(\frac{\text{maskB}}{\text{maskA} + \text{maskB}}\right) \cdot b \tag{9}$$



(a) Source (b) Mask (c) Source swapped (d) Mask swapped (e) Seamless

Figure 4: The four images used to make a seamless image

Note that this is not Java code, so don't try to type it in. We can't add or multiply entire images like this.

Instead, follow this general algorithm:

```
make new blended image
for each row r
  for each column c
    find weight of a's pixel
    find weight of b's pixel
    calculate weight proportion
    compute blended pixel by mixing a's and b's pixels
```

Give the blended image the same type as image a. Compute the weights and the color of pixel (c, r) in the blended image using the following computation:

$$\text{weightA} = \text{red component of maskA}[c, r] \tag{10}$$

$$\text{weightB} = \text{red component of maskB}[c, r] \tag{11}$$

$$\text{weightSum} = \text{weightA} + \text{weightB} \tag{12}$$

$$\text{weight} = \frac{\text{weightA}}{\text{weightSum}} \tag{13}$$

$$\text{blended}[c, r] = \text{weight} * a[c, r] + (1 - \text{weight}) * b[c, r] \tag{14}$$

The masks are grayscale images, but if we use `getRGB` on them, we get back an `int` that has the red, green, and blue components all packed together in separate bytes. Each component happens to be identical since we are in grayscale, but we only want one of them to represent the weight. Feed the packed RGB `int` to the `Color` constructor, and then use `getRed` on the color object to isolate the red component.

If the weights from mask a and b sum to 0, avoid the illegal division but instead consider them to be equally weighted. Test this method before moving on.

6. Method `makeSeamless`, which accepts two parameters in this order:

   (a) an image to make seamless, of type `BufferedImage`

(b) a power of type `double`

It returns a seamless image built using the following algorithm:

(a) Swap the corners of the source image a to generate image b

(b) Generate a circle mask $maskA$ the same size as a and of the given power

(c) Swap the corners of the circle mask to produce $maskB$

(d) Mix images a and b according to their masks

An example source image, its three helper images, and the final seamless image are shown in Figure 4. Test this method before moving on.

7. Method `tile` which accepts three parameters in this order:

   (a) an image to tile, of type `BufferedImage`

   (b) the number of horizontal repetitions, of type `int`

   (c) the number of vertical repetitions, of type `int`

   It returns a new `BufferedImage` containing the source image tiled across it the given number of times. The new image has the same pixel type as the source (see `BufferedImage.getType`). An example of tiling an image of gravel is shown in Figure 5. Test this method before moving on. Visit `http://www.twodee.org/teaching/cs145/2016c/homework/hw4/tiler.html`, upload your image, and scroll through it endlessly.

3 Extra

For an extra credit participation point, share an image not already shared and its tiling 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 145 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.

(a) Source

(b) Seamless

(c) Seamless tiled 4x3

(d) Source tiled 4x3

Figure 5: Tiling of gravel