Jigsaw Puzzles

COMS 2270 Object-Oriented Programming (Fall 2025)

Assignment 3 (300 points)

- Due Date: Wednesday, November 19th, 11:59 pm (midnight)
- 5% bonus for submitting 1 day early (by 11:59 pm November 18th)
- 10% penalty for submitting late (by 11:59 pm November 20th)
- No submissions accepted after November 20th, 11:59 pm

> **This assignment is to be done *on your own*. See the Academic Integrity policy in the syllabus, for details.**
>
> **You will not be given credit unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away. !!**

If you need help, try office hours, the schedule is on Canvas. Lots of help is also available through the Piazza discussions. Please start the assignment as soon as possible to get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

This is a ”regular” assignment so we are going to read your code. Your score will be based partly on the autograder’s functional tests and partly on the Teaching Assistants’ assessment of the quality of your code. See the ”More about grading” section.

Contents

1 Tips from the experts: How to waste a lot of time on this assignment

1. Start the assignment the night it's due. That way, if you have questions, the TAs will be too busy to help you and you can spend the time tearing your hair out over some trivial detail.

2. Wait until the day before the assignment is due to start using Gradescope. Or better yet don't test your code at all, it's such fun to remain in suspense until it's graded!

3. Don't bother reading the rest of this document, or even the specification, especially not the "Getting started" section. Documentation is for losers. Waste time writing lots of code before you figure out what it's supposed to do.

2 Overview

In this assignment we will implement a computer-based version of a puzzle loosely similar to a jigsaw puzzle. It will not be able to handle irregular interlocking shapes, but it will still be difficult enough to solve even though all the pieces are just squares.

A nice picture will be divided into squares called tiles which are all of the same size, chosen so that the height and the width of the picture are both a whole number of squares. For example a 500x300 picture could be divided into 100x100 square tiles so that the picture is three tiles tall and five tiles wide.

Then the tiles will scrambled up in a random way (without changing the overall shape of the picture). The puzzle solver will then try to unscramble the picture by looking at the visual clues and swapping tiles with adjacent tiles.

To make this even more interesting, we will exploit the symmetries of the square – the computer will optionally turn the individual squares by rotating the tiles randomly and/or flipping the tiles upside

down (horizontally or vertically), or transposing them. This adds to the difficulty of the puzzle and increases the challenge factor from a users' perspective.

From a coder's perspective, what you're going to do is to learn how to manipulate two dimensional arrays in very many ways. You will also have some more practice with ArrayLists and working with files.

It is one thing for the solver to recognize that the picture has been unscrambled correctly, but a computer needs a little more help doing that. AI and computer vision applications can probably detect when the picture is still scrambled (it will have perfectly straight vertical and horizontal edges with "sharp" changes of color) , but since we don't know how to use those tools, we will simply *keep track* of the scrambled state of the picture and keep it up to date while the user makes moves. Each tile of the picture will have a little representative called a **Tracker** that remembers where the tile's original scrambled position is (where it is coming from originally), and where its correct final position (in a solved puzzle) is. It will also remember a summary of how the tile has been turned, flipped or transposed, so that it can recognize when it has been rotated the right way with the correct face up, as required to "unscramble" the picture.

2.1 Symmetries of the Square

If you consider a square, there are different operations you can apply to the square so that it still looks like a square. They are presented in the chart in Figure 1 on page 4.

Here is a fun activity. You can make a very simple diagram that can be used to manually experiment with all the ways to orient a square. Label the top left corner of a blank square with one symbol, such as a red dot, and the top right corner of the square with another symbol, such as a blue dot. Also label the back of the square with a red dot and a blue dot so that the red dots on both sides are at the same corner, and the blue dots on the two sides are at the same corner. Now make a second square just like the first one. You can use both of these to physically experiment with the transformations illustrated in Figure 1 to convince yourself the figure is correct.

We will use a naming system for the different ways you can orient a square. They will be called either Cn or CnF, where n is the number of clockwise rotations that have been applied. Since four rotations return back to the original state, n will be 0, 1, 2 or 3. If after applying the required number of rotations it is also flipped horizontally, then the name ends in F. See Figure 1 on page 4 to help understand the meanings of the names and operations.

A square that has not been rotated and has the correct face up will have the red dot at the top left corner and the blue dot at the top right corner, and will be called C0. This is shown in the top left of the chart in Figure 1.

One operation that can be applied is to rotate the square **clockwise**, turning it by ninety degrees, so that the red dot moves to the top right, and the blue dot moves to the bottom right. In the chart, this is labelled C1 (shown below C0). Note that the green arrow going from C0 to C1 represents a clockwise rotation. Likewise you can rotate it clockwise again by 90 degrees, to obtain C2, which is drawn below C1. Check that the colored dots are moving the way you would expect for a clockwise rotation. After rotating it a third time you have C3, which is in the bottom left corner of the chart. If you rotate C3 clockwise again, you end up back at C0, so this is indicated in the chart by the green arrow going from bottom left C3 to the top left position C0. In the diagram green arrows always mean clockwise rotations.

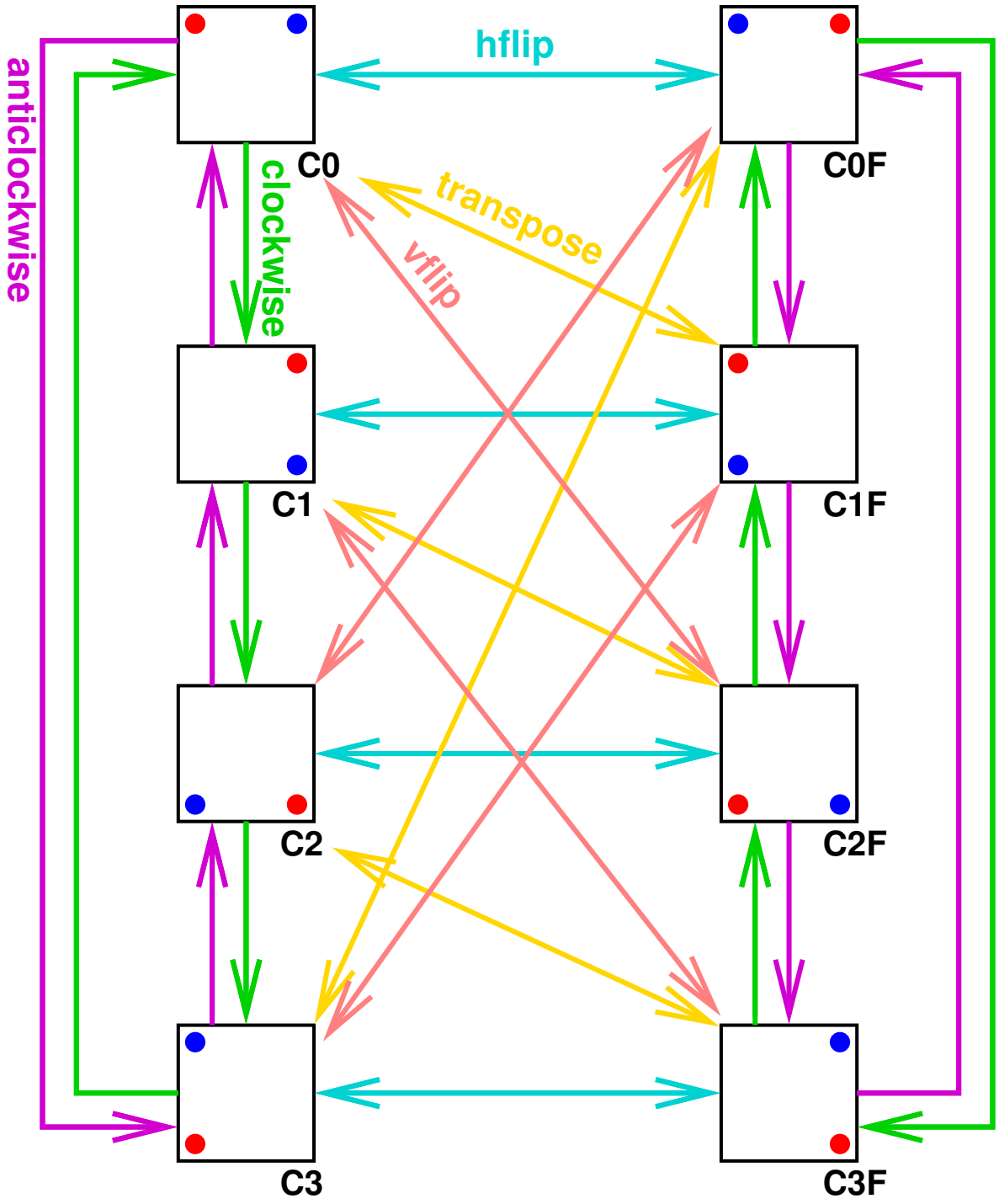

Figure 1: Symmetries of the square. The colored transition arrows are labelled once each with the name of the operation corresponding to the color for the transitions leaving C0.

Another basic operation is to turn the square upside down (or more accurately, "top-face-down") by turning it the way you turn the page of a book (left becomes right and right becomes left). This reflects the square across a vertical line down the middle. Starting from C0 in the top left, this yields C0F which is at the top right side of the chart. This operation is called a horizontal flip, abbreviated **hflip**. Notice that if you hflip starting from C0F, you get back to C0. In general, a second hflip will always cancel an hflip (two hflips are equivalent to doing nothing). We represent hflips throughout the diagram with a

cyan-colored double-ended arrow (it is double ended because both directions are hflips).

Now study the chart carefully to convince yourself that the red dots, blue dots, green arrows and cyan arrows are all correct. The clockwise rotation and the hflip are enough to go everywhere you need to go in the chart. But in practice three more operations are nice-to-have, even though they can be expressed in terms of the first two.

An **anticlockwise** rotation does the exact opposite of what a clockwise rotation does. Since C0 goes clockwise to C1, C1 goes anticlockwise back to C0. The anticlockwise rotations are indicated in magenta arrows. A anticlockwise rotation that immediately follows a clockwise rotation cancels it.

This is a good time to convince yourself that it is correct that the green clockwise arrows cycle downwards on the left side, but upwards on the right side, and the magenta arrows for anticlockwise rotations cycle upwards on the left side but downwards on the right side.

A vertical flip (nicknamed a **vflip**) means you reflect the square across a horizontal line through the middle, so that the top edge becomes the bottom edge and vice versa (in the process it is turned upside down). Note that just like hflips, a vflip cancels a vflip, so they have double-ended arrows. From C0, if you apply one vflip, you end up at C2F. From C2F, another vflip takes you back to C0. The arrows for vflips are pink.

Finally there is the **transpose**, in which the square is reflected along a line from the top left corner to the bottom right corner, so that the top right and bottom left corners exchange positions. For example from C0, if you transpose, the blue dot moves from the top right corner to the bottom left corner but the red dot does not move. This is shown with the gold arrow. It involves turning the square upside down but in a particular way. All transposes are indicated as gold double-ended arrows (because a transpose after a first transpose cancels the first one). The arrows for transposes are colored gold.

The clockwise rotations, anticlockwise ones, hflip, vflip and transpose operations are called stationary operations because they don't change the position of a tile among the other tiles. They only affect how the tile is oriented.

Why do we care about all this?

We need to describe (or summarize) the exact state of a square no matter how many hflips, vflips, clockwise turns and anticlockwise turns have been applied to it. For example if seven consecutive hflips have been applied, we only need to remember that their final effect just like 1 hflip, because six of them cancel each other out in pairs. The summary is to remember the resulting way to orient a square.

Fortunately there are only eight different ways the to orient a square, as indicated in the chart. Another way to explain this is to say that there are only four possible places to put the red dot, and for each possible red dot position, the blue dot can be to its clockwise corner or to its anticlockwise corner, making a total of 8. Even though we can apply lots of different operations to a square, we will always end up in one of those eight arrangements in the chart, and it is enough to remember which of the 8 configurations a square is in.

You can check that there are many sequences of operations which leave the square unchanged:

- simply leave the square alone, or
- hflip it twice, or
- vflip it twice, or
- transpose it twice, or
- rotate it clockwise four times, or
- rotate it anticlockwise four times.

All of these sequences of operations leave the square back in its starting configuration C0.

There are other interesting sequences of operations which are equivalent. For example, you can check that a transpose is the same as one clockwise rotation followed by an hflip. A vflip can be obtained as two clockwise turns, followed by (or preceded by) an hflip, or else two anticlockwise turns, followed by (or preceded by) an hflip. You can check this from the chart.

Note that there is exactly one way to go from C0 to any other square, if you make a rule that first you have to do *up to three* clockwise rotations followed by an optional hflip. This method can derive any of the eight orientations by choosing the correct number of clockwise rotations and deciding correctly whether to hflip. This explains the naming system we used.

While operations are being applied to the tiles (remember a tile is one of the square parts of a picture), we will always (somehow) remember the corresponding orientation, so that when the tile is finally in C0 we will know it. If all the tiles are in a C0 state, and are also in their correct final positions, then the puzzle is solved. Also if all the squares are in C0F, and each row is in opposite order to the correct order, the puzzle can be considered solved (in this case, you are looking at the mirror image of the original picture, but otherwise it is correct).

To see why, consider the following picture in Figure 2:

Figure 2: A wintry picture (Photo Credit: Kinkate of Pixabay)

It can be split divided into tiles as shown in Figure 3:

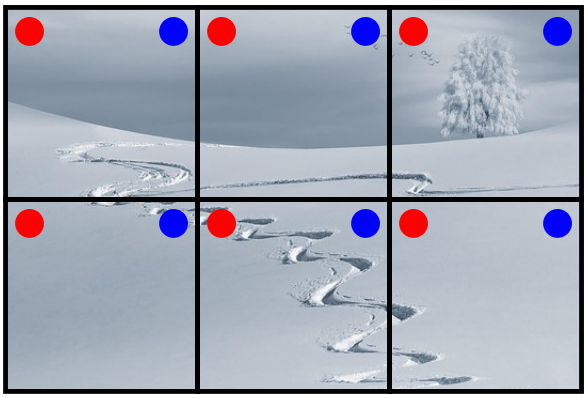

Figure 3: Normal solution with all tiles in C0.

And the “mirror image” solution is shown in Figure 4:

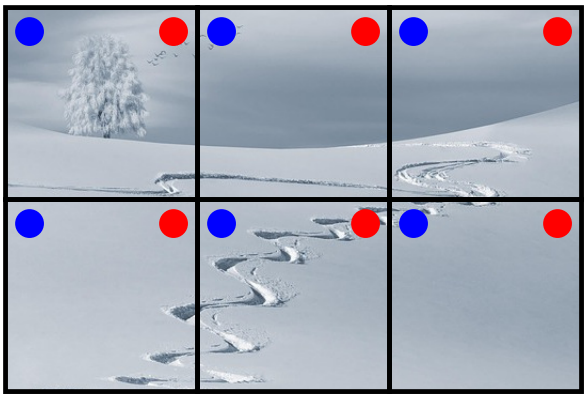

Figure 4: Mirror image solution with all tiles in C0F and each row of tiles arranged “backwards”.

We accept the mirror image solution because a user working with visual cues alone is equally likely to create either version if they have not seen the original picture, and we don’t want to ask them to convert the mirror image solution into the unmirrored one, just so they can finish their puzzle. That would not be fun.

2.2 The Puzzle Cursor and Position Changes

During the puzzle, there is a cursor that indicates which tile the user is currently working on. If the user requests an hflip, vflip, transpose or either type of rotation, the operation is always applied to the tile where the cursor is.

In order to work on a different tile, the cursor can be moved around in the picture without moving any of the tiles as follows. If the cursor is not in the top row, then it is allowed to move up. If it is in the top row, then a request to move up does nothing. Analogous rules hold for when a tile is on the rightmost column (it cannot move right), when it is in the leftmost column (it cannot move left), and when it is on the bottom row (it cannot move down). In all these cases it will simply do nothing if requested to do the impossible.

The tiles of the picture themselves can be swapped with a neighbor according to very similar rules. Only the tile with the cursor is allowed to swap (together with one of its adjacent tiles with which it swaps positions). If the cursor is not in the top row, then its tile is allowed to swap positions with the tile above itself. The cursor must move with the tile during the swap.

Similarly if the cursor is not in the bottom row, its tile is allowed to swap positions with the element below it. A tile is also allowed to swap with the element to its right if it is not in the rightmost column, and to swap with the element to its left if it is not in the leftmost column. If a tile is in the top row (for example) and a swap with the element above it is requested, nothing happens. A similar rule exists for all other directions.

When the puzzle has been correctly solved, the cursor vanishes to reveal the final picture.

2.3 Implementation Details

For our purposes, a picture is simply a two dimensional array of very tiny squares called pixels. This will be stored in a class called **Picture**. Each pixel’s color is represented by an integer, so in the Picture

(0, 0)	(0, 100)	(0, 200)
(100, 0)	(100, 100)	(100, 200)

Table 1: Corner pixel coordinates for a tiled 300x200 Picture with tileSize 100.

class, we will simply work with two dimensional arrays of integers and a method will be provided to you for displaying that array as a picture.

Exactly which integers correspond to which colors is beyond the scope of this assignment. You will only need to know the value of the color red, which we will use for drawing the cursor, so that value will be provided to you in the skeleton code. The rest of the colors will just be copied from position to position, so you don't need to know the meaning of their values.

The width of the picture is the number of pixels from the left edge to the right edge. Likewise the height of the picture will be the distance from the top edge to the bottom edge. The row number of a pixel (that is, the distance from the top edge) will be the first index **`i`** in the expression **`a[i][j]`**. Likewise the **`j`** coordinate of a pixel is its row number (which is its distance from the left edge).

In a 300x200 pixel picture for example, we will use the convention that the width is stated first, and the height second, so this would be 300 pixels wide by 200 pixels tall. When we break it into square tiles, each tile must have the same size, and its size must evenly divide both 300 and 200. So we could use tiles with 100 pixels on each side, and the picture would then be three tiles wide by two tiles tall. Or we could use tiles with a size of 50 pixels, for a puzzle that is six tiles wide by four tiles tall.

In the former case (with 100x100 tiles), the top left corners of the tiles will be at the following **`(i,j)`** locations:

Notice the pattern with multiples of 100. These are simply the indices of cells of a 3x2 array, all multiplied by 100. The tile in the top left, for example has pixels 0 to 99 across the top and pixels 0 to 99 down the left edge, so its bottom right corner is (99, 99).

We will need to write code for reading out a tile from a Picture and also have a way to write a tile back into a Picture. That is how we will implement the movement (swapping) of tiles.

Naturally we will also have code in the Picture class for flipping, rotating and transposing the tiles while the user makes "stationary" moves (moves which change a tile without changing its position). All of these are implemented in the Picture class, which manages the 2D integer array that will be translated into pictures. When a tile has been extracted from a Picture, that will also be a Picture object, just a smaller square one.

A Tracker object will store coordinates for where the corresponding tile began in the puzzle in the scrambled initial state, where its target (correct) location is, in the unscrambled state. Each of these positions will require two indices. It will also store the current orientation of the tile to summarize the "stationary" operations that have been applied to it. Note that a Tracker does not contain the current location of the tile in the picture. Eg. if a tile is moving from position A in the scrambled state towards position C in the solved state but is currently at B, the position B is not stored inside the Tracker object. Only A and C are.

B will instead be maintained in a two dimensional array of Tracker objects in a class called a Board. If the tracker is at position B=(i,j) in the 2D array of Trackers, then the tile which should eventually be at C is currently in position (i,j) among the tiles.

Continuing with the example sizes above, there will be a 3x2 2D array of Tracker objects in the Board. The Tracker in row i and column j of the Board describes the tile in tile-row i and tile-column j of the Picture. Referring to Table 1 for example, the pixel rows are from 0 to 199, but the tile rows are just 0 and 1.

The Board will also store the current cursor position and control how the cursor moves.

Another class called the **`Puzzle`** will provide utilities for scrambling the tiles of a picture initially, and displaying arrays as actual pictures. The Puzzle class is beyond the scope of the assignment and will be provided to you.

At all times, the Puzzle keeps a clean unscrambled copy of the picture being solved. The scrambled state of the picture that the user sees is always generated on the fly from the information in a Board variable, including the cursor position. Basically for each Tracker object at row i and column j in the Board, we figure out where it is supposed to eventually be in the solution, and draw the Picture tile from the solved position at the position (i, j) of the scrambled Picture.

A crude console-based interface is provided in **`KbdUI.java`** so that when you're done coding, you can solve real puzzles on a sample picture (small ones though because the interface is a little clumsy). The picture can be any large size that fits on your screen, but the number of tiles cannot be too high. We recommend no more than 16 tiles, or it would be tedious to solve. A good place to start is with 6 tiles. All of its available pictures are 960x640, with a tileSize of 320, but if you'd like you can change the tileSize to 160, to obtain a 24-tile problem.

3 Special Assignment Requirements

Hints: To solve this assignment you will need to declare instance and local variables, that part of the program design is up to you.

Yes, it is possible to solve this assignment very neatly with just what we have learned so far in class, and the challenge of doing so will make you a better programmer! **Keep in mind that it's always better to turn in something that is working than nothing at all**.

4 More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about 2/3) on Gradescope's functional tests and partly on the TA's assessment of the quality of your code. This means you can get partial credit even if you have errors, but it also means that **even if you pass all the Gradescope tests you can still lose points**. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Are you using a conditional statement when you could just be using **`Math.min`**? Some specific criteria that are important for this assignment are:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
  - You will lose points for having unnecessary instance variables
  - All instance variables should be private.

- Accessor methods should not modify instance variables.
- You should have as little code repetition as possible. Write your own methods if needed! Just don't forget to write javadocs for any private methods you choose to define.

See the "Style and documentation" section below for additional guidelines.

5 Style and documentation

Roughly 10 to 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- For this assignment, most of the public members already have javadocs as a way of specifying their behavior. You are only expected to write javadocs for your own instance variables, and private methods, and the file as a whole. Including the javadocs we have already provided, the following conditions must be met:
- The class and **every method, constructor and instance variable, whether public or private, must have a meaningful Javadoc style comment**. The javadoc for the class itself can be very brief, but must include the **`@author`** tag. The javadoc for methods must include `@param` and **`@return`** tags as appropriate. The javadoc for instance variables can be a very short description of the purpose of the variable.
- Try to briefly state what each method does in your own words. However, there is no rule against copying the descriptions from the online documentation. *However:* **do not literally copy and paste from this pdf!** *This leads to all kinds of weird bugs due to the potential for sophisticated document formats like Word and pdf to contain invisible characters.*
- Run the javadoc tool and see what your documentation looks like! (You do not have to turn in the generated html, but at least it provides some satisfaction :)
  - All variable names must be meaningful (i.e., named for the value they store).
  - Your code should not be producing console output. You may add println statements when debugging, but you need to remove them before submitting the code.
  - Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include an internal comment explaining how it works.)
- Internal comments always precede the code they describe and are indented to the same level. We encourage the use of internal comments to make your code as clear as possible, but do not overdo it.
- Use a consistent style for indentation and formatting.
- Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to

  Window→Preferences→Java→Code Style→Formatter

  and click the New button to create your own "profile" for formatting.

6 Gradescope

A significant portion (but not all) of the points for this assignment are based on the gradescope autograder tests. **It is not advised to wait until the last day to run the gradescope on your code**. You have an unlimited number of submissions, and unless we receive complaints from gradescope, we will not restrict the rate at which you can submit.

Do not try to "cheat" the tests by "hardcoding" your code to pass specific tests. You are expected to implement a solution that meets the specifications described in this document, the tests are only provided to gain confidence that your solution is correct. We reserve the right to add new "hidden" tests during grading and the graders will look for "hardcoding" during the manual part of the grading.

The autograder for this Assignment will be posted on gradescope and announced. Please see the last section of this document for more information on how to submit to Gradescope. Note that this time you will be submitting three class files, so the instructions have changed slightly.

7 Specification

The specification for this assignment includes this pdf (particularly the overview), the detailed javadocs provided in the skeleton code, and any "official" clarifications announced on Canvas. The overview provides the important concepts, and the javadoc specification expresses those ideas in terms of a class design. If you define any private members, please provide brief javadocs for those to help TAs understand your work. Please do not change the provided APIs and javadocs in the skeleton code.

Here is a list of files provided and brief comments about each one.

- **Picture, Tracker & Board** These contains skeleton code and javadocs for the public members. You need to implement the missing public members.
- **PictureIO** This contains code you do not need to modify. They have to do with using java library to display images on the screen and read images from a file.
- **Puzzle** This contains code you don't need to modify. A Puzzle ties together a board and a picture.
- **UnitTests** Contains some unit tests. A few more tests may be added in the gradescope autograder.
- **SimpleTest** contains just one little interactively-run test to demonstrate how you can "test" your clockwise code on a real picture. It is provided just to provide a sense of satisfaction, since the UnitTests do not display any real pictures.
- **KbdUI** The main method in this file may be run if you want to actually try solving a puzzle for fun. I have recorded a couple of videos of "gameplay" that will be available on Canvas so you can see the end result before you finish coding.

8 How to implement stationary operations in Picture

Here is a general way to figure out how to implement each of the stationary operations (clockwise, anticlockwise, hflip, vflip and transpose).

At the heart of each of these operations (in the middle of two nested for loops) is an assignment statement that says that when you read a pixel from position `picture[i][j]` you should write it

into position **`newpicture[newI][newJ]`**. Each of **`newI`** and **`newJ`** must be defined in terms of **`i`** and **`j`** and that is the essence of solving the problem.

We will use clockwise rotation as an example. To see the coordinate changes that are needed, perform the following little exercise:

- Take a sheet of paper or an index card. It should be rectangular (not square).
- Turn the paper so that it is in landscape mode (meaning it is wider than tall), and (horizontally) write on it the words "Hello World" to remind yourself what the original orientation was.
- Draw an arrow from the top left corner, downwards along the left edge, about halfway, and label it "row = i".
- Draw an arrow from the top left corner, rightwards along the top edge, about one-third of the way, and label it "col = j".
- Draw a dot on the paper which is "i" rows down and "j" columns across to the right. It should look something like Figure 5.

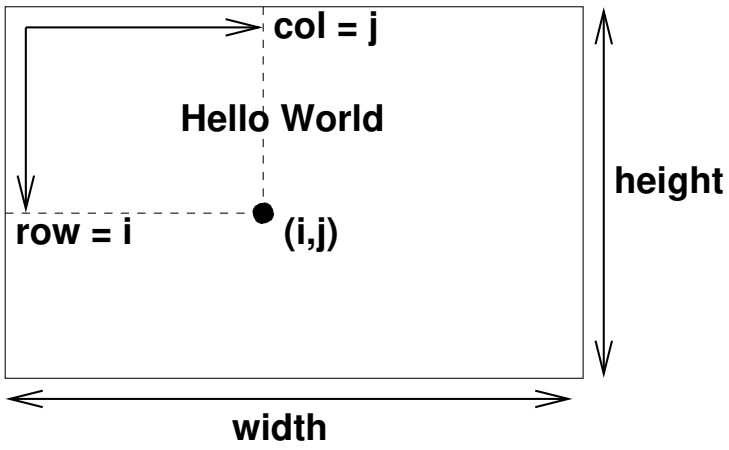

Figure 5: Old layout of coordinates and dimensions.

- Now physically rotate your sheet of paper clockwise by 90 degrees. The long side and short side of the paper should "exchange places" and it should be in portrait mode, that is, it should look taller than wide. The written words "Hello World" should be downwards now. It should look like Figure 6 on page 13.

- While staring at the paper in this new orientation (Figure 6) note how the new value of i (that is, **`newI`**) of the dot depends on the old value of **`i`** and **`j`**. Recall that **`newI`** is the distance down the new page. This will give you an equation for calculating **`newI`** from the old **`i`** and **`j`**.
- Also note how **`newJ`** (that is, the distance to the right across the new page) depends on the old values of **`i`** and **`j`**.

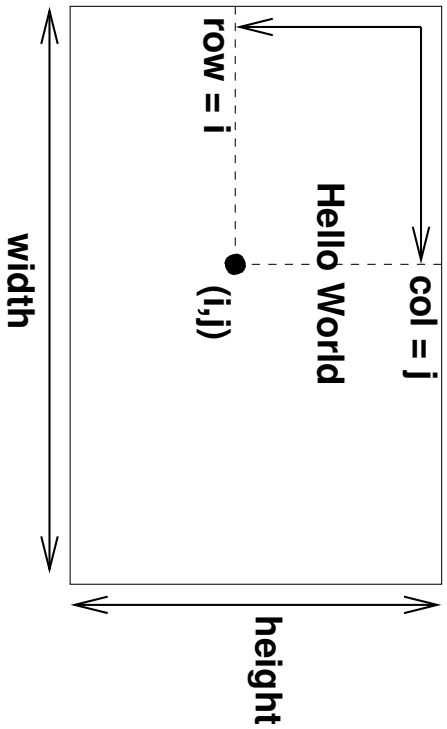

Figure 6: Old coordinates and dimensions after rotating clockwise. Useful to determine new coordinates and dimensions.

- These equations show you the position the old cell at (i, j) will move to, when the clockwise-rotation transform is done. The nice thing is that this is not only true for the dot, but for every pixel position with any row i and any column j.
- The other question you should consider is — what is the width of the new picture in terms of the old width of the picture and the old height of the picture? Likewise note the height of the new picture in terms of the old width of the picture and the old height of the picture. Both of these should affect how you allocate a 2D array to contain the new picture.
- Use your conclusions to write the clockwise transformation code.
- Test it by running a small example with a real Picture to make sure it does what you expect.
- If you want to apply this technique to an operation that turns the paper upside down, then use transparent paper or plastic with a permanent marker instead of white paper, so that even after turning it upside down, you can still read what you wrote. If you don't have transparent paper or plastic, use white paper, but hold it up agaist a light so you can read the other side through the paper.

When the code is written correctly it is much much shorter than the intuitive description above of why it works.

9 Suggestions for getting started

9.1 Initializing the Project

There are two ways you can proceed:

9.1.1 Importing the Project

This is the preferred method. For this you will need the file Homework3SkeletonProject.zip from Canvas. Once you have downloaded that, you may watch this in the uploaded video on canvas called projectImport.mp4.

- Select File Menu > Import
- Popup comes up
- Choose General > Projects from Folder or Archive > Next
- At the top, click the Archive button, then choose the *Project.zip file.
- Then choose one of the paths where you would like to keep the project.
- Choose the one which says ”Eclipse Project” at the right side.
- Click the Finish button
- Then open the project and look at the source files.

9.1.2 Starting from scratch

If the above method does not work for you do this instead. You will need the file Homework3Skeleton.zip from Canvas. Once you have downloaded that, create an empty project and call it any name you like. Then create the package hw3 in it, and install junit in it and also create nearly-empty class files called the following names:

- Board.java
- KbdUI.java
- Picture.java
- PictureIO.java
- Puzzle.java
- SimpleTest.java
- Tracker.java
- UnitTests.java

Then unzip the file Homework3Skeleton.zip on your computer, and copy the unzipped files over the nearly empty class files you just created.

9.2 Working on the Project

Smart developers don't try to write all the code and then try to find dozens of errors all at once; they work *incrementally* and test every new feature as it's written.

Here is an example of some incremental steps you could take in writing this class. The approach here is not the only way to solve the problem, and you may choose your own path. If you do some of the questions here may not make sense. Your only requirement is to implement the specification in the javadocs, and to make sure all the provided tests run correctly (though you don't have to get them running in the order presented here).

You can work on the Picture class or the Tracker class first. You should finish the Tracker class before you work on the Board class. Here we will begin with the Picture class.

Probably the first thing to do for each class is to just get your code to compile and run the tests (even if they all fail) by writing stubs with dummy return values.

Next, generate the javadocs. In Eclipse, click on **Project**, then **Generate Javadoc**, and at least select the **`Picture`**, **`Tracker`** and **`Board`** classes. Look at the html pages and read each class's page while working on it. Or read them in the skeleton code.

Then you can work on the classes one by one.

9.3 Picture

(a) Please read the javadocs very carefully, because they contain additional specification/details not presented in this pdf file.

(b) First implement the constructors, easy getters and the one pixel setter. The specifications are provided in the javadoc.

(c) Next, read Section 8, and then try implementing the clockwise rotation described there. Write a test program that just creates a picture and displays it. Using a rectangular picture ensures that you have got the dimension changes correct. Then rotate the picture clockwise and display the rotated version. The first stationary operation may be a little hard to get right, but once you have the framework the rest will be easier. You could try something like this:

```
1  package hw3;
2
3  public class SimpleTest {
4      public static void main(String[] args) {
5          Picture p = new Picture("photos/bridge-9599215_1920.png", 320);
6          PictureIO pio = new PictureIO();
7          pio.firstShow(p.getPixels());
8          Picture q = p.clockwise();
9          PictureIO qio = new PictureIO();
10         qio.firstShow(q.getPixels());
11     }
12 }
```

(d) Now you should implement the other static operations (anticlockwise, hflip, vflip and transpose), just by turning your paper different ways and staring at it and asking the right questions about it in the "turned" position. How wide is it? How tall is it? What are the new coordinates of the dot,

in terms of the original coordinates and the original dimensions? You can copy your clockwise code and edit small parts of it to obtain each of the other transformations.

(e) Implement **`getTile`** and **`setTile`**. Both of these require that you translate the tile coordinates, which are typically small numbers, into pixel coordinates, which are typically much bigger numbers, by multiplying by the tileSize. To test it, try to read a tile from one corner, and then set it into a different corner. Now the picture should have two of the same corner in different places.

(f) Lastly you want to work on **`drawCursor`**. Start by setting the four corner pixels of the cursor's tile using the **`RED`** color. Then get creative. What do you want your cursor to look like? Use some for loops to draw some patterns that make the cursored tile have some unique red marks on it to distinguish it from the other tiles. We used red because red usually stands out against most photos.

(g) After completing **`Picture`**, tests 1 to 36 should pass.

9.4 Tracker

Now we can work on the **`Tracker`** class.

(a) Please read the javadocs very carefully, because they contain additional specification/details not presented in this pdf file.

(b) First you can implement the constructor with many parameters, and the getters.

(c) Note that the Tracker class does not contain pixels. So here, the operations clockwise, etc, do not physically move pixels, but instead change our representation of the state of the tile.

The interesting part of the tracker class is to figure out how to internally represent the names C0, C0F, C1, C1F etc, in such a way that the colored arrows on the chart in Figure 1 are easy to implement. Your chosen representation should also make **`getRotations`** and **`getIsFlipped`** easy to implement. Now using some simple logic, implement all the colored arrows in the chart. Think of this question: if the chart only consisted of the left half, how would you represent the rotations so that **`clockwise`** and **`anticlockwise`** are easy to implement? Then how can you change your representation to include the right side of the chart and allow for **`getIsFlipped`**?

When you have chosen a representation, implement the clockwise operation, then the anticlockwise. Next work on the hlip, then the vflip, then the transpose. The latter three require you to have a representation of the fact that the paper is upside down.

(d) Last you can work on the constructor which takes a single string parameter, and the implementation of toString.

(e) After completing **`Tracker`**, tests 37 to 56 should pass.

9.5 Board

Now we can work on the **`Board`** class.

(a) Please read the javadocs very carefully, because they contain additional specification/details not presented in this pdf file.

(b) First, do the constructor, getters and setters.

(c) Next work on the four move operations for moving the cursor without moving the underlying tile.

(d) Lastly work on the four swap operations which move the cursor together with the underlying tile to an adjacent tile, effectively swapping the two tiles.

(e) Now consider the stationary operations. These should simply pass-through the intention of the user, to the tile that is currently selected by the cursor.

(f) Implement the undo operation. This requires that for every operation you have stored on the list of user moves, you know how to undo it in one move. This should be true whether it was just a cursor movement, a swap operation or a stationary operation. Please make sure that if a move is impossible, we cannot "undo" it.

(g) Lastly, work on the conversion of the internal state to and from Strings, by implementing the **`load`** and the **`save`** operations. We imagine that in a good UI, the user should be able to pause their work and then go away for a little bit and then come back to continue.

(h) After completing **`Tracker`** and **`Board`**, tests 57 to 70 should pass.

Try playing the Puzzle with your own code by running the **`KbdUI`** as a main application. Does everything play together nicely?

You should also test your code on gradescope, if you haven't done so till now.

10 Requirements

The requirements for the class are embodied in the public API, so it is not mandatory that you use the plan above. But they could be very helpful! If you end up with a lot of unnecessarily duplicated code the grader may take off some points for style.

The autograder will be available on Gradescope. There are many test cases so there may be an overwhelming number of error messages. Always start reading the errors at the top and make incremental corrections in the code to fix them.

10.1 Restrictions on Java Features For this Assignment

See above the above section about using good programming practices. You may not use any outside libraries (i.e., external libraries such as Apache Commons; otherwise, we won't be able to compile your code when grading.

11 If you have questions

For questions, please see the Piazza Q&A pages and click on the folder **`hw3`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`hw3`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, **make the post "private"** so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any announcements from the instructors on Canvas that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be posted in the Announcements section of Piazza (and emailed directly to you). (We promise that no official clarifications will be posted within 48 hours of the due date.)

12 What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, or else you won't get any credit for this assignment.

Assignments must be submitted on Gradescope, there is nothing to submit on Canvas. The assignment will be called "Assignment 3" on Gradescope. See the first page of this document for the early-bird deadline, the normal deadline and the late-penalty deadline, along with their score modifications.

Gradescope will run a number of functional tests and report how many (and which) tests your code passed. There are many test cases so there may be an overwhelming number of error messages. ***Always start reading the errors at the top and make incremental corrections in the code to fix them. Please submit just the three files called***

- **`Picture.java`**
- **`Tracker.java`**
- **`Board.java`**

and at the top of each one, there should be a line that puts it in the hw3 package, that is, "**`package hw3;`**". Do not create or submit any zipfiles with Gradescope even though gradescope may offer you the chance to submit a zipfile. Zipfiles created by speccheckers (just in case that is what you have) are not in the format that Gradescope expects. So just submit the three files as is. Please note that any 5% bonus for early submission or 10% penalty for late submission will not be calculated by Gradescope itself. Instead, it will be graded manually by a Teaching Assistant.