Assignment 3

ECSE 250 Fall 2024

Due: 5pm (Montreal Time), Dec 4, 2024

Learning Objectives

By the end of this assignment, you will be able to model hierarchical data using trees. You will also be comfortable implementing recursive methods which take advantage of the fact that the data structure you are working with is recursively defined. You will learn how to convert a tree into a flat, two-dimensional structure, and, finally, you will strengthen your knowledge and your comfort with using inheritance in Java.

General Instructions

This assignment contains 3 parts. You need to download the files provided. Your task is to complete and submit the three following files:

`Block.java`
`PerimeterGoal.java`
`BlobGoal.java`

- **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD YOUR CODE HERE" block.** You may add `private` helper methods to the three classes you have to submit, but you are not allowed to modify any other class.
- Please make sure that all the files you submit are part of a package called `assignment3`.
- You are NOT allowed to use any class other than those that have already been imported for you. **Any failure to comply with these rules will give you an automatic 0.**
- Do NOT start testing your code only after you are done writing the entire assignment. It will be extremely hard to debug your program otherwise. If you need help debugging, feel free to reach out to the teaching staff. When doing so, make sure to mention what is the bug you are trying to fix, what have you tried to do to fix it, and where have you isolated the error to be.

Submission instructions

- Late assignments will be accepted up to 3 days late and will be penalized by 20 percent of the total marks. Note that submitting one minute late is the same as submitting 72 hours

late. We will deduct points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it the wrong file submitted, the wrong file format submitted or any other reason. We will not accept any submission after the 3 days grace period. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.

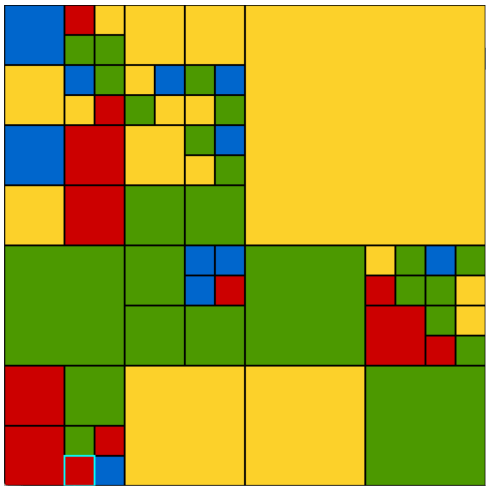
- Don't worry if you realize that you made a mistake after you submitted: you can submit multiple times but **only the latest submission will be evaluated**. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and Ed Lessons may be overloaded during rush hours).
- **Do not submit any other files, especially .class files and the tester files. Any failure to comply with these rules will give you an automatic 0.**
- The tests on ED are the tests we will be using to grade your work. Whenever you submit your files to Ed, **you will see the results of some exposed tests**. If you do not see the results, your assignment is not submitted correctly. **If your assignment is not submitted correctly, you will get an automatic 0. If your submission does not compile on ED, you will get an automatic 0.**
- The assignment shall be graded automatically on ED. **Requests to evaluate the assignment manually shall not be entertained, and it might result in your final marks being lower than the results from the auto-tests.** Please make sure that you follow the instruction closely or your code may fail to pass the automatic tests.
- Next week, a mini-tester will also be posted. The mini-tester contains tests that are equivalent to those exposed on Ed. We encourage you to modify and expand it. ***You are welcome to share your tester code with other students on Ed.*** Try to identify tricky cases. Do **not** hand in your tester code.
- **Failure to comply with any of these rules will be penalized**. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Ed.

Introduction

This assignment consists of implementing a visual game in which players apply operations such as rotations to a recursive structure in order to work towards a goal. The main data structure can be represented with a quad-tree (i.e. a tree in which each internal node has exactly four children). The rules are simple, but the game is still challenging to play. The game board resembles a Mondrian painting, and you can easily pick the color scheme that is most appealing to you. This assignment is adapted from an assignment created by Diane Horton and David Liu from University of Toronto.

The Game

The game is played on a randomly-generated game board made of squares of four different colors, such as the following:



Each player is randomly assigned their own goal to work towards: either to create the largest connected "blob" of a given color or to put as much of a given color on the outer perimeter as possible.

There are three kinds of moves a player can do:

- rotating a block (clockwise or counterclockwise),
- reflecting the block horizontally or vertically (i.e. along the x-axis or the y-axis if you imagine the origin of the axes being place in the center of the block), and
- "smashing" a block (giving it four brand-new, randomly generated, sub-blocks).

After each move, the player sees their score, determined by how well they have achieved their goal. The game continues for a certain number of turns, and the player with the highest score at the end is the winner.

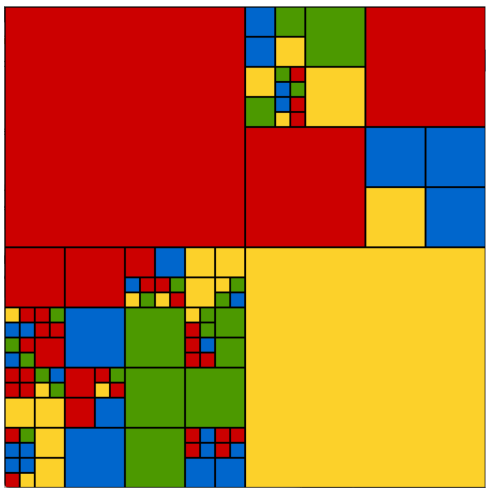
The Game Board

We will call the game board a "block". Blocks can be recursively defined; a **block** is either:

- a square of one color, or
- a square that is subdivided into 4 equal-sized blocks.

The largest block of all, containing the whole structure, is called the **top-level block**. We say that the top-level block is at level 0 (i.e. it would be at the root of the quad-tree used to represent it). If the top-level block is subdivided, we say that its four sub-blocks are at level 1 (i.e. these would correspond to the children of the root in the aforementioned quad-tree). More generally, if a block at level k is subdivided, its four sub-blocks are at level $k + 1$.

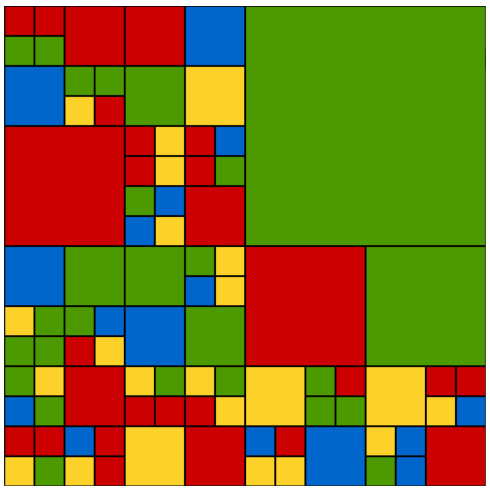
A board will have a maximum allowed depth, which is the number of levels down it can go. A board with maximum allowed depth 0 would not be fun to play on since it couldn't be subdivided beyond the top level, meaning that it would be of one solid color. The following board was generated with maximum depth 5:



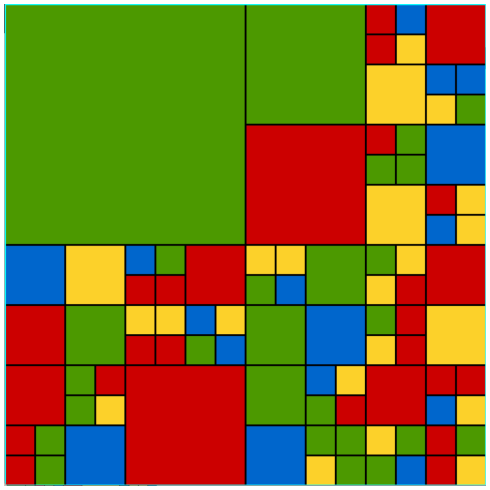
For scoring, *the units of measure are squares the size of the blocks at the maximum allowed depth.* We will call these blocks **unit cells**.

Moves

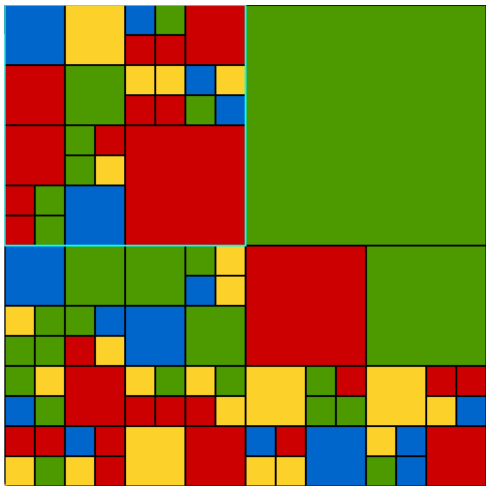
To achieve their goal, the players are allowed the three type of moves described above. Note that, smashing the top-level block is not allowed, since that would be creating a whole new game. And smashing a unit cell is also not allowed, since it's already at the maximum allowed depth. What makes moves interesting is that they can be applied to any block at any level. For example, if the player selects the entire top-level block (highlighted) for this board



and chooses to rotate it counter-clockwise, the resulting board would be the following:

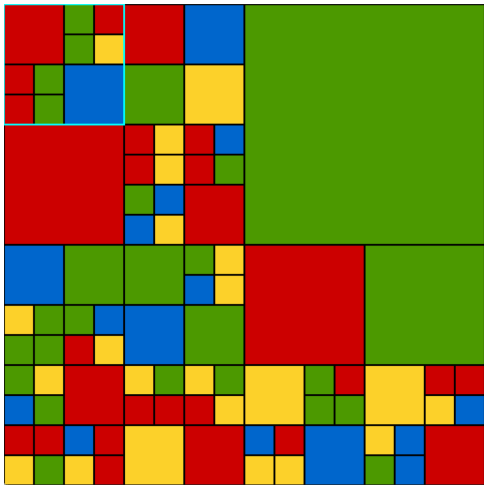


But if instead, on the original board, they choose to rotate (still counter-clockwise) the block at level 1 in the upper left-hand corner, then the resulting board would be the following:



Finally, if instead they choose to rotate the block a further level down, still sticking to the upper-left

corner, they would get this:



Of course, the player could have chosen many other possible blocks on the board and they could have decided to perform a different type of move.

Goals and Scoring

At the beginning of the game, each player is assigned a randomly-generated goal. There are two types of goal:

- *Blob goal.* The player must aim for the largest "blob" of a given color c. A **blob** is a group of orthogonally connected blocks with the same color. That is, two blocks are considered connected if their sides touch; touching corners does not count. The player's score is the number of unit cells in the largest blob of color c.
- *Perimeter goal.* The player must aim to put the most possible units of a given color c on the outer perimeter of the board. The player's score is the total number of unit cells of color c that are on the perimeter. There is a premium on corner cells: they count twice towards the score.

Notice that both goals are relative to a particular color. We will call that the **target color** for the goal.

PART - 0: Familiarize yourself with the starter code

Let's start by getting comfortable with the code provided and the idea behind the block data structure.

As mentioned in the introduction, we will be using a quad-tree to represent the structure of a block. Quad-trees are trees in which each internal node has exactly four children. It would not make sense for us to use a tree in which nodes could have a different number of children, say three. This is because a block is either solid-colored or subdivided; if it is solid-colored, it is represented by a node with no children (i.e. a leaf), and if it is subdivided, it is subdivided into exactly four sublocks.

Open the `Block.java` file and familiarize yourself with the provided code in this class. Note that the class has the following fields:

- Two `int`s, `xCoord` and `yCoord`, representing the coordinates of the upper left corner of this Block. Note that the origin, $(0, 0)$, is the top left corner of the window. Just like in A2, the window has the following coordinates:

(0,0)	(1,0)	(2,0)	→
(0,1)	(1,1)	(2,1)	
(0,2)	(1,2)	(2,2)	
↓			

- An `int size` representing the height and width of this Block. Since all blocks are square, one variable is enough to represent both.
- An `int level` representing the level of this Block within the overall block structure. The top-level block, corresponding to the root of the tree, is at level 0. As already noted, if a block is at level i, its children will be at level $i + 1$.
- An `int maxDepth` representing the deepst level allowed in the overall block structure.
- A `Color color`. If this block is not subdivided (i.e. if it is a leaf), then this field stores its color. Otheriwse, this field should be `null`.
- A `Block[] children` representing the blocks into which this block is subdivided. The children are stored in this order: upper-right child, upper-left child, lower-left child, lower-right child. If this Block has no children, then this field should store a reference to an array of length 0.

Before beginning to write your own code to complete the implementation of this class, let's get more comfortable with the Block structure. Start by drawing on paper the quad-tree representing game board from Figure 1 (assuming the maximum depth is 2):

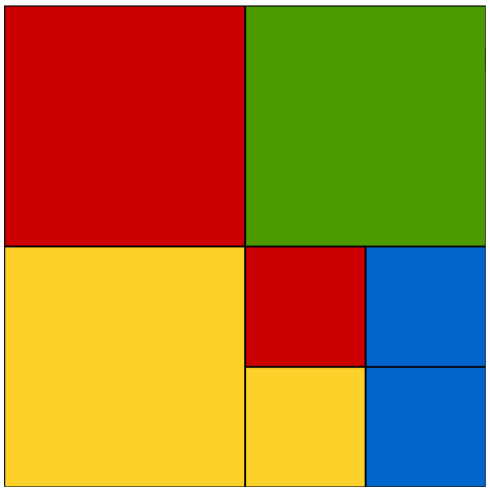


Figure 1: A Block with max depth 2

Each node should contain the values that would be assigned to their corresponding `Block` objects. You can assume that the size of the top-level block is 16. How many nodes have you drawn?

NOTE: If you come to office hours, we will ask to see your drawing before answering questions!

Now that you can draw the block structure on paper, can you translate it into code? Inside the `main` method of the `Block` class, create a `Block` object that corresponds to the same game board above. You can access constants representing the `Colors` needed from the class `GameColors`. Once you have created the board, you can display its text representation using the method `printBlock()` provided to you. If the structure has been created correctly the following should be displayed:

```
pos=(0,0), size=16, level=0
    GREEN, pos=(8,0), size=8, level=1
    RED, pos=(0,0), size=8, level=1
    YELLOW, pos=(0,8), size=8, level=1
    pos=(8,8), size=8, level=1
        BLUE, pos=(12,8), size=4, level=2
        RED, pos=(8,8), size=4, level=2
        YELLOW, pos=(8,12), size=4, level=2
        BLUE, pos=(12,12), size=4, level=2
```

Invariants. While thinking about the block structure, you might have noticed that each nodes must satisfy a lot of invariants for the tree to correctly represent a Block. Here is a list of some of them:

- Nodes have either 4 children or 0.
- If this Block has children then:
  - their `maxDepth` is the same as the one of this Block
  - their `size` is half that of this Block
  - their `level` is one greater than that of this Block
  - the position of their upper left corner can be determined from their size and the position of the upper left corner of this Block (I'll leave it up to you to figure out how).
  - the `color` of this Block is `null`
- `level` is always smaller than or equal to `maxDepth`

TEST and DEBUG. Please note that to test and debug your code you should not rely on the GUI. The GUI can definitely help fine tune your code, but you should test and debug it before using it. The method `printBlock()` provided to you displays a text representation of the block and can help you through this process. Remember that you are allowed and encouraged to add as many `private` helper methods as you wish. This will help you to: keep your code cleaner and better organized, as well as break each task into smaller steps. All of this will in turn help you when testing and debugging.

PART - I: Set up the board (30 points)

With a good understanding of the data structure, you are now ready to start working on the completion of the class `Block`.

[10 points] For the game, we want to be able to generate random boards. This is what the constructor `Block(int lvl, int maxDepth)` is for. The method generates a random Block with level `lvl`, and maximum depth `maxDepth` using the following the strategy: if a Block is not yet at its maximum depth, it can be subdivided. Decide whether or not to do so as follows:

- Use the `Random` object stored in the field `gen` to generate a random number in the interval [0, 1). Please note that you should generate this number *if and only if* it is possible for this Block to be subdivided.
- Subdivide the block if the random number is less than `Math.exp(-0.25 * level)`, where `level` is the level of the Block within the tree.
- If a Block is not going to be subdivided, use a random integer to pick a color for it from the array of colors in `GameColors.BLOCK_COLORS`. Make sure to generate the integer in the appropriate range.

Notice that the randomly-generated Block may not reach its maximum allowed depth. It all depends on what random numbers are generated.

This constructor is responsible for assigning the appropriate values to the fields of all Blocks within the Block it generates except the fields `size`, `xCoord`, and `yCoord` which should be instead initialized by default. Next, you will write a method that can be used to correctly initialize all these fields.

If you use the seed 2 when initializing the `Random` variable `gen` on line 17 of `Block.java`, then when executing the following snippet of code

```
Block blockDepth2 = new Block(0,2);
blockDepth2.printBlock();
```

the text below will be displayed:

```
pos=(0,0), size=0, level=0
    GREEN, pos=(0,0), size=0, level=1
    RED, pos=(0,0), size=0, level=1
    YELLOW, pos=(0,0), size=0, level=1
    pos=(0,0), size=0, level=1
        BLUE, pos=(0,0), size=0, level=2
        RED, pos=(0,0), size=0, level=2
        YELLOW, pos=(0,0), size=0, level=2
        BLUE, pos=(0,0), size=0, level=2
```

[**10 points**] Implement the method `updateSizeAndPosition()`. This method takes three integers as input: one representing the size of this block, and the others representing the coordinates of the upper left corner of this block. The method updates the size and position for this block and all of its sub-blocks, while ensuring consistency between the values of the fields and the relationship of the blocks. Make sure the invariants are all respected! The method should throw an `IllegalArgumentException` is the input for the size is invalid, i.e. if it is negative or it cannot be evenly divided into 2 integers until the max depth is reached. There's no need to perform any input validation for the coordinates.

If you use the seed 2 when initializing the `Random` variable `gen` on line 17 of `Block.java`, then when executing the following snippet of code

```
Block blockDepth2 = new Block(0,2);
blockDepth2.updateSizeAndPosition(16, 0, 0);
blockDepth2.printBlock();
```

the text below will be displayed:

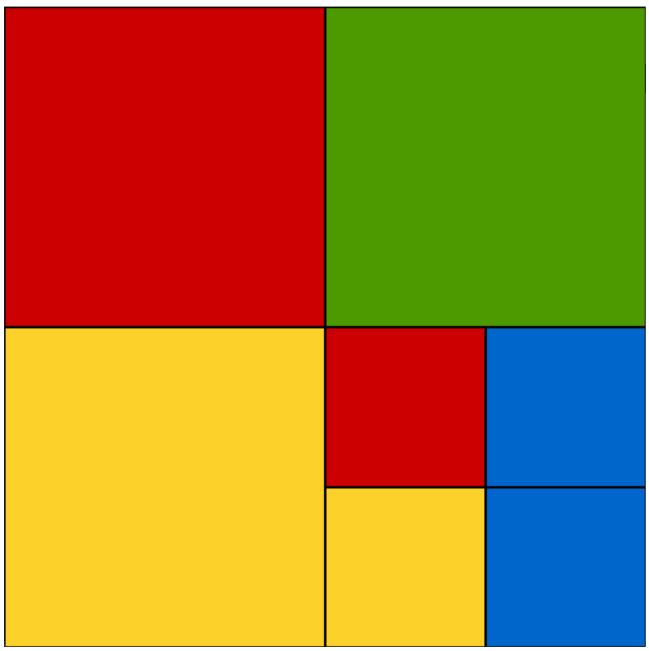
```
pos=(0,0), size=16, level=0
    GREEN, pos=(8,0), size=8, level=1
    RED, pos=(0,0), size=8, level=1
    YELLOW, pos=(0,8), size=8, level=1
    pos=(8,8), size=8, level=1
        BLUE, pos=(12,8), size=4, level=2
        RED, pos=(8,8), size=4, level=2
        YELLOW, pos=(8,12), size=4, level=2
        BLUE, pos=(12,12), size=4, level=2
```

[**10 points**] In order for the game to be able to draw the blocks you can now generate, you need to provide an implementation for the method `getBlocksToDraw()`. The method returns an `ArrayList` of `BlockToDraw`s. Open the file `BlockToDraw.java` to familiarize yourself with this data type. The list returned by `getBlocksToDraw()` should contain, for each undivided Block:

- one `BlockToDraw` in the color of the block
- another `BlockToDraw` in the `FRAME_COLOR` (see `GameColors.java`) and a stroke thickness equal to 3.

Note that a stroke thickness equal to 0 indicates that the block should be filled with its color. The order in which the objects `BlockToDraw` appear in the list does not matter.

After having implemented this method, you can try to run `BlockGame`. If you use the seed 2 when initializing the `Random` variable `gen` on line 17 of `Block.java` and you select 2 as the maximum depth when running the game, you will see the following board displayed:



As you might have noticed, this is the same Block from Figure 1.

PART - II: Make the game playable (30 points)

Time to make the game real by allowing players to be able to select a block from the board and apply a move of their choice.

[12 points] The first thing to do is implement the method `getSelectedBlock()` which allows the game to retrieve the Block selected by the player. In order for the user to play the game, they must be able to select a block on which they would like to make a move. They will do so by clicking on the board at a desired location, and using the up and down arrows to choose the level of the block. The method `getSelectedBlock()` takes those inputs (the (x, y) coordinates of where the user clicked, and an `int` representing the level selected), and finds the corresponding Block within the tree.

If the level specified is lower than the lowest block at the specified location, then the method returns the block at the location with the closest level value. Note that if a Block includes the location (x, y), and that Block is subdivided, then one of its sub-Blocks will contain the location (x, y) too. This is why the level is needed to identify which Block should be returned. Please note that you can view the quad-tree representing the block data structure as a search tree. As such, this method should run in $O(h)$, where h is the height of the tree.

Input validation: if the level provided is smaller than this Block's level or larger than its maximum depth, then the method should throw an `IllegalArgumentException`. If the position, (x, y), is not within this Block, then the method should return `null`.

For example, using the seed `4` when initializing the `Random` variable `gen` on line 17 of `Block.java`, the following snippet of code generates a board with maximum depth 3, top left corner in

position $(0, 0)$, and size equal to 16.

```
Block blockDepth3 = new Block(0,3);
blockDepth3.updateSizeAndPosition(16, 0, 0);
```

We can then select and print the Block at level 1 containing the location $(2, 15)$ as follows:

Code to execute	Text displayed
`Block b1 = blockDepth3.getSelectedBlock(2, 15, 1);` `b1.printBlock();`	`RED, pos=(0,8), size=8, level=1`

Please note, that the same would have been displayed even if the level selected was 2 or 3. On the other hand, we can select and print the Block at level 2, containing the location $(3, 5)$ as follows:

Code to execute	Text displayed

`Block b1 = blockDepth3.getSelectedBlock(3, 5, 2);`
`b1.printBlock();`
	`pos=(0,4), size=4, level=2`
	`  YELLOW, pos=(2,4), size=2, level=3`
	`  GREEN, pos=(0,4), size=2, level=3`
	`  YELLOW, pos=(0,6), size=2, level=3`
	`  BLUE, pos=(2,6), size=2, level=3`

[6 points] Implement the method `reflect()`. The method takes an `int` as input representing whether this Block should be reflected over the x-axis (if the input is 0) or the y-axis (if the input is 1). When thinking about performing a reflection of this Block, you can think of the origin of the axes being placed in the center of this Block.

To successfully perform a reflection, the same operation should be propagated to all the sub-blocks of this Block. Make sure that the method correctly updates all the necessary fields. The method throws an `IllegalArgumentException` if the integer received as input is neither a 0 nor a 1.

For example, using the seed `4` when initializing the `Random` variable `gen` on line 17 of `Block.java`, let's generate a board with maximum depth 3. In Figure 2, you can see the effect of a reflection over the x-axis of either the top-level block or the level 1 block on the top-left corner.

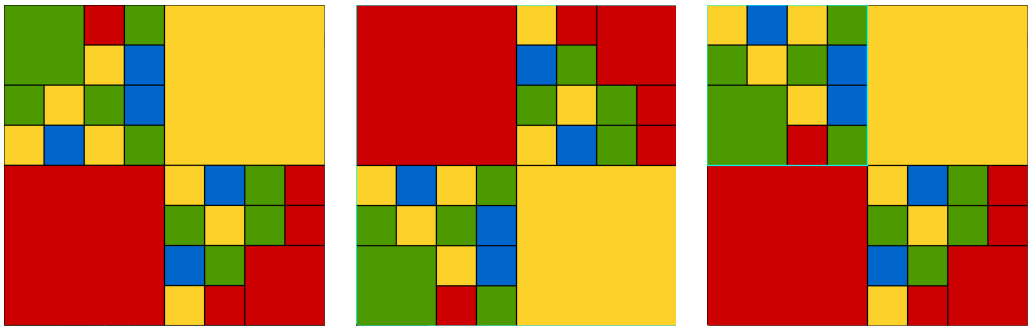


Figure 2: Left: Original board. Center: Board after reflection on top-level block. Right: Board after reflection on the level 1 top-left block.

[6 points] Implement the method `rotate()`. The method takes an `int` as input representing whether this Block should be rotated counter-clockwise (if the input is 0) or clockwise (if the input is 1).

The operation should be propagated to all of its sub-blocks. If this Block has no children, then the method should not do anything. Make sure that the method correctly updates all the necessary fields. You can throw an `IllegalArgumentException` if the integer received as input is neither a 0 nor a 1.

For example, using the seed `4` when initializing the `Random` variable `gen` on line 17 of `Block.java`, let's generate a board with maximum depth 3. In Figure 3, you can see the effect of a clockwise rotation of either the top-level block or the level 1 block on the top-left corner.

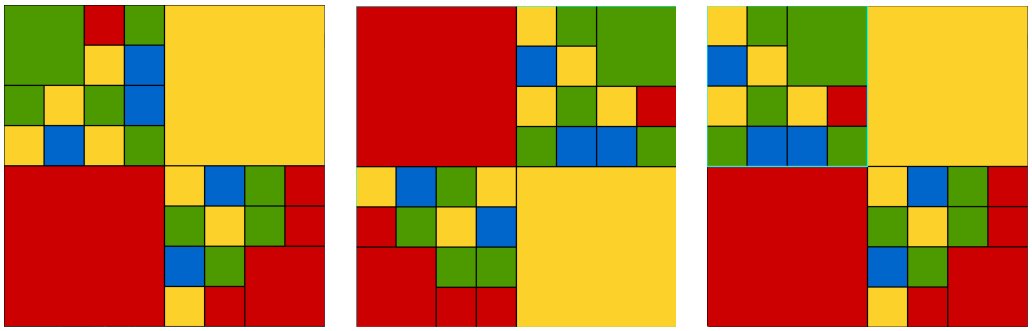


Figure 3: Left: Original board. Center: Board after a clockwise rotation of the top-level block. Right: Board after a clockwise rotation of the level 1 top-left block.

[6 points] Implement the method `smash()`. This method takes no inputs and, if this Block can be smashed, it randomly generates four new sub-blocks for it. If the Block already had children, they will be discarded. Make sure that the method correctly updates all the necessary fields.

Remember that a Block can be smashed if and only if it is not the top-level Block and it is not already at a level equal to the maximum depth. The method returns `True` if this Block was smashed and `False` otherwise.

For example, using the seed `4` when initializing the `Random` variable `gen` on line 17 of `Block.java`, let's generate a board with maximum depth 3. In Figure 4, you can see the effect of smashing the level 1 block on the top-left corner or the effect of smashing the level 2 block on the top-left corner.

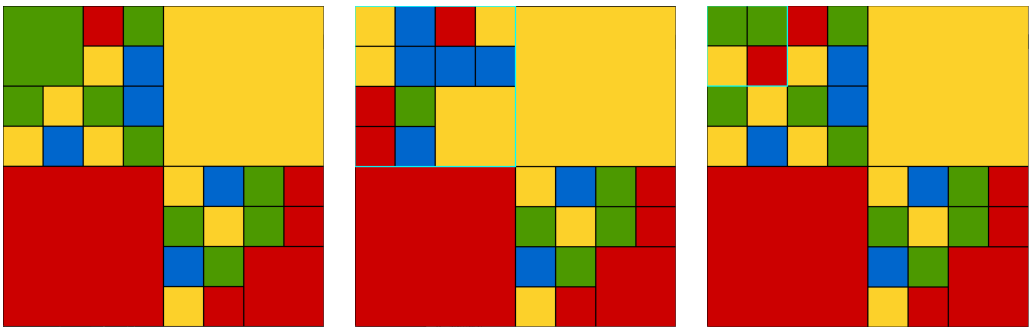
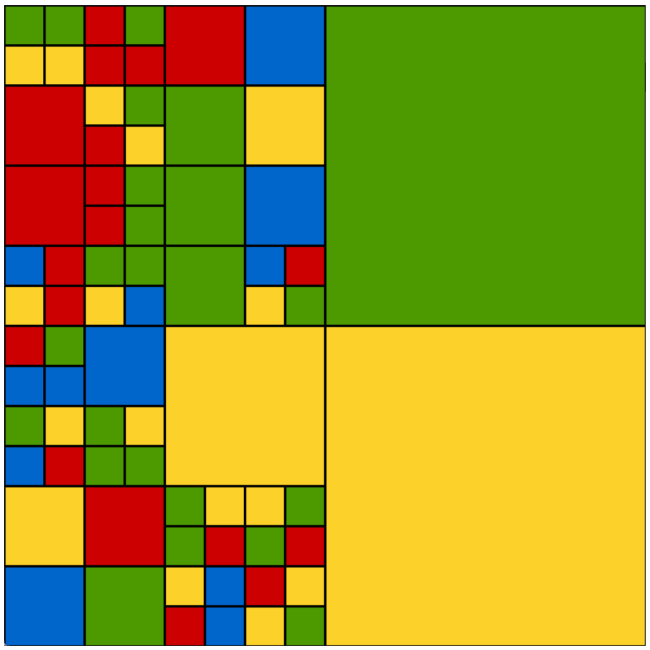


Figure 4: Left: Original board. Center: Board after smashing the level 1 top-left block. Right: Board after smashing the level 2 top-left block.

PART - III: Implementing scoring (40 points)

Now that player can play the game, let's get our scoring system working.

The unit we use when scoring against a goal is a unit cell. The size of a unit cell depends on the maximum depth in the Block. For example, with maximum depth of 4, we might get this board (which you can obtain using the seed 123 for the `Random` field in `Block.java`):



If you count down through the levels, you'll see that the smallest blocks are at level 4. Those blocks are unit cells. It would be possible to generate that same board even if maximum depth were 5. In that case, the unit cells would be one size smaller, even though no Block has been divided to that level.

Notice that the perimeter or the largest "blob" may include unit cells of the target color as well as larger blocks of that color. When computing the score for a perimeter goal, if a larger block of the target color is on the perimeter, only the unit-cell-sized portions on the perimeter counts.

When computing the score for a "blob" goal, a larger block of the target color has a value that is equivalent to the number of unit cells that could fit in that block.

For example, suppose the maximum depth were 4, the target color were green, and the board were as displayed above. Then the score for the perimeter goal would be 24 while the score for the blob goal would be 65. Notice that, for instance, the large block on the top-right corner is not divided into 64 unit cells, but we still score as if it were. That is, that block counts for 16 of the points in the perimeter goal score (remember that corner unit cells count double!), while it counts for 64 of the points in the blob goal score.

Now that you understand these details about scoring, you can go ahead and implement the last few methods of the assignment.

[15 points] It is very difficult to compute a score for a perimeter goal or a blob goal by walking through the tree structure. (Think about that!) The goals are much more easily assessed by walking through a two-dimensional representation of the game board. To make this possible, implement the method `flatten()` from the `Block` class. This method takes no input and returns a two-dimensional array `arr` of `Color`s representing this Block as rows and columns of unit cells. Note that `arr[i]` represents the unit cells in row `i`, and `arr[i][j]` is the color of the unit cell is row `i` and column `j`.

So, for instance, `arr[0][0]` is the color of the unit cell in the upper left corner of this Block. Once `flatten()` is implemented, you can use the method `printColoredBlock()` provided to you to display a colored text representation of the flattened block.

For example, if you generate a Block with level `0` and maximum depth `2` when the seed of the random generator is set to `2` (i.e. the same block from Figure 1), then calling `printColoredBlock()` will display the following:

RRGG
RRGG
YYRB
YYYB

[5 points] Open the file `PerimeterGoal.java` and implement the method `score()`. This method takes as input a `Block` representing a board and returns the score of this goal on that board based on its target color. You can start by flattening the Block to make your job easier.

Now, let's move into implementing scoring for the other goal. Scoring with a blob goal involves flattening the tree, iterating through the cells in the flattened tree, and finding out, for each cell, what size of blob it is part of (if it is part of a blob of the target color). The score is the biggest of these.

But how do we find out the size of the blob that a cell is part of? Let's start by implementing a

helper method that allows you to do exactly this.

[**14 points**] Implement the method `undiscoveredBlobSize()` from the class `BlobGoal`. The method takes as input two integers `i` and `j`, a 2D array of `Color`s representing unit cells, and a 2D array of `boolean`s. It returns the size of the blob of the target color of which the unit cell in position `(i, j)` is part of. If this unit cell it's not the target color, then it is not in a blob of the target color, so the method returns 0. If it is of the target color, then it is in a blob of the target color.

It might be a very small blob consisting of itself only, or a bigger one. The method should find out the of blob the neighbours cells are in, and use this information to figure out the blob size of the current cell (sounds recursive, right?). Be careful, a potential problem with this is that when the method computes the blob size of the neighbour cells, we do not want the current one to be counted in, otherwise this cell will end up being double counted (or worse). To avoid such issues, we need to keep track of which cells have already been "visited" by the algorithm. This is what the array of `boolean`s is for! So, to be more precise, the method returns the number of unit cells of the target color that are orthogonally connect to the the unit cell in position `(i, j)` and have not been visited yet (i.e. their corresponding `boolean` value is `false`). This number should include the unit cell in position `(i, j)`. If this cell is not of the target color, or it has already been visited, then the method returns 0. You can assume that the two input arrays have exactly the same size.

[**6 points**] Finally, implement the method `score()` from the class BlobGoal. This method takes as input a `Block` representing a board and returns the score of this goal on that board based on its target color.

Conclusions

Congratulations for having completed the third and last assignment of this semester. Take pride in your gorgeous code and enjoy the game! If you had fun coding this assignment and you are left wanting more, note that you can expand and create many variations of the game. You could for instance add different types of goals, or implement computer players and challenge yourself while playing against an AI. The possibilities are endless, let your imagination run wild! :)