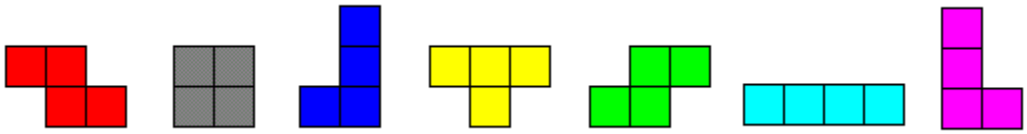
FINAL PROJECT COSC 1430

Overview

You will be implementing the game Tetris. Tetris is a deceptively simple, and addictive puzzle game. Small pieces fall from the top of the grid to the bottom. The pieces are comprised of 4 squares arranged into 7 different patterns. Players must rotate the pieces as they fall and fit them together to complete lines. **When the player fills an entire line with blocks, that line is removed from the screen**. If the player cannot complete lines, the blocks will eventually fill to the top of the screen and the game ends.



Concepts

The purpose of this assignment is to gain experience with the following concepts:

- class relationships
- understanding and modifying provided code
- 2-dimensional arrays

Program Synopsis

In the real game of Tetris, a tetris piece, composed of 4 contiguous squares, falls from the top of the grid. The piece can be moved left, right, or down as well as rotate clockwise. When the piece lands on the bottom of the grid or another piece it becomes frozen, that is to say, a part of the grid, and then another piece is created. If an entire row fills up with parts of different pieces, then the entire row is removed and the rows above it are moved down. Play continues until the pieces pile up to the top and a new piece is created on top of already frozen pieces.

The starter code for this project implements a simple game that only creates a single L-shaped piece that can move downwards. It uses elementary keyboard input and graphics classes. For the **First part**, you will add specific features to make it a more complete Tetris game. (The **Second Part** will ask you to produce a complete Tetris game that uses different shapes.)

I strongly recommend you do the first part to understand the code.

Download the starter code: EventController.java, Game.java, Grid.java, LShape.java, Square.java, Tetris.java, and Direction.java (can be found on BlackBoard)

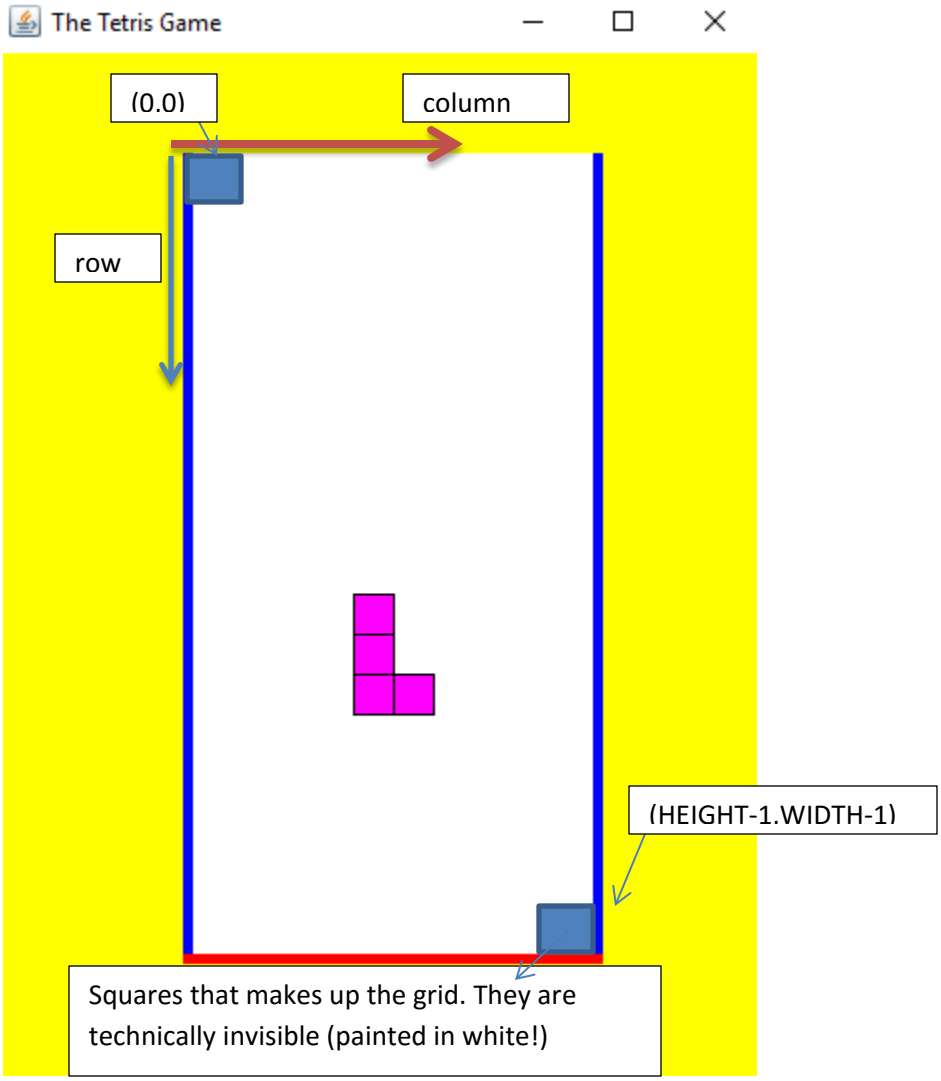
FIRST PART

Here is a description of the different classes. These first 2 classes have to do with the user interface

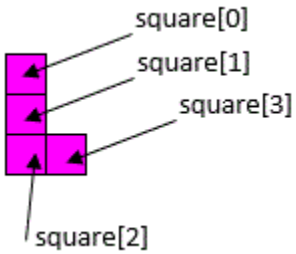
- Tetris - class to handle the graphics display. Also contains the main method to start the game. You do not have to modify this class at all. There is a lot of graphics code in here that you probably are not familiar with. Don't worry; we will learn about it as the quarter progresses.
- EventController - class to react to events (like key presses and timer events) and tells the game what to do. You will not add any methods to this class, but will have to modify an existing method. There is alot of event handling code in here that you probably are not familiar with. Don't worry; we will learn about it as the quarter progresses.

The rest of the classes have to do with modelling the game. Except for the draw(Graphics g) methods, you should be able to read and understand the rest of the code.

- Game - code that keeps track of the current piece and the board. It contains the public methods to actually "play" the game.
- Grid - the grid is actually the board area on the screen where the piece can move. Each square of the grid is represented as an element of a 2D array of Square objects. The upper left Square is at (0,0). The lower right Square is at (HEIGHT -1, WIDTH -1) where HEIGHT and WIDTH are the numbers of rows and columns in the grid. .The grid also keeps track of where previous pieces have "frozen."



- LShape - the game piece made of 4 squares stored in a 1D array of Square objects as shown in the picture.



- Square - the building block for the LShape and the Grid
- Direction - an enum (short for enumeration) that lists all possible directions in the game of Tetris. An enum is a convenient way to group constants that are

related. It also ensures that all of the variables of type Direction will have a value equal to one of the constants. More information about enums is given here.

Read through and become familiar with the sample code provided. Determine what the different relations are between different classes and what the different methods do. Drawings would come in handy here.

REQUIREMENT OF FIRST PART (50 points)

- In order to get another piece to appear on the screen, the following line of code needs to be added:

```
piece = new LShape(1, Grid.WIDTH/2 -1, theGrid);
```

  Read through the starter code and find the correct place to put this line.

- The game piece also needs to be able to move LEFT and RIGHT if the corresponding arrow keys are pressed. In order to handle these keys, you will need to modify the method `keyPressed(keyEvent e)` in the EventController class. You will find the correct constant names for the associated keys in the KeyEvent class in the Java API.

SECOND PART (50 points)

You will be completing the game Tetris. The features needed are

1. Adding the other 6 pieces
2. Adding the ability to rotate the pieces

This part will utilize these important concepts:

- inheritance
- class hierarchy
- dynamic dispatch
- algorithms

OTHER PIECES: The first step in the assignment is to add the other six pieces. Let's think a little about the planning though; we don't want the Game class to have to know about all 7 different piece types (we wouldn't want Game to have 7

different instance variables, 1 for each type. How would it keep track of which one was current?) Instead, Game should know about 1 type (a super type) and let dynamic dispatch do the work for us.

So to start, we need to redesign (or **refactor**) the current code. To do this, we want a super type that contains all of the behaviors common to all of the pieces. This can be achieved with an interface Piece. Then sub types will implement the interface for the individual pieces and their individual needs. However, some implementations will be the same for all of the pieces (e.g. getColor()). To avoid repeating code, the implementation of the methods common to all of the pieces can be written in an abstract class AbstractPiece that implements the Piece interface. This abstract class will also list the fields that are common to all of the pieces.

So what is the same about all of the pieces?

- All of the method signatures currently in LShape are common to all of the pieces (except the constructor of course). They should be listed in the interface Piece. You will also add a rotate method since for this assignment we want the pieces to be able to rotate.
- All of the instances variables currently in LShape are common to all of the pieces. They should be declared in AbstractPiece as protected. Make sure that you delete them from LShape.
- The method implementations in LShape should be moved to AbstractPiece (except for the constructor code.)

What is different about each piece:

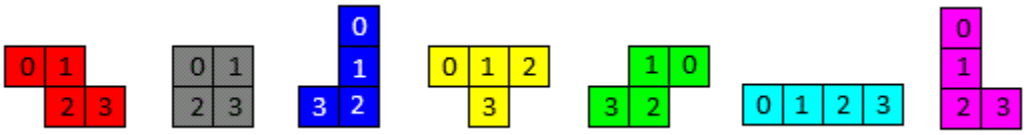
- How each individual playing piece is constructed. That's it! Even though we may initially think that the rotation of a piece is specific to that piece and should be coded in its class, it is actually possible to write all of the rotation code in AbstractPiece as we will see below.

1) Therefore, start by breaking up the LShape.java class into an interface (Piece), an abstract class (AbstractPiece) and a sub class (LShape with most of its code removed).

- The only method in LShape should be the constructor: It should call the super class constructor and initialize the square array. Nothing else! For instance, the initialization of the grid instance field should be done in AbstractPiece and not in LShape.

At this point, test your program. It should run as it did before.

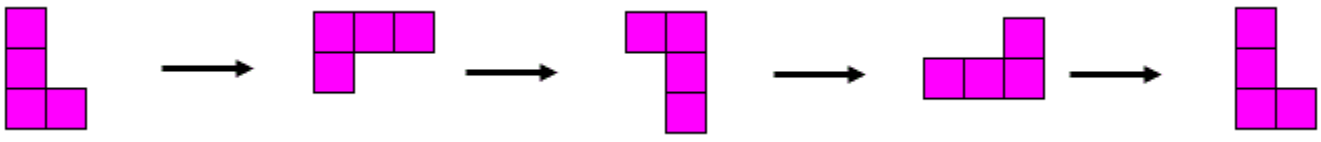
2) Now it is time to add the other pieces. You should create these new classes by deriving them from the AbstractPiece class. In the order they appear in the picture below, you can name the classes ZShape, SquareShape JShape, TShape, SShape, BarShape. In particular, you will need to figure out how to initialize the pieces. This will be similar to how the L-shaped piece was done, and, in fact, you may find it helpful to start each new class by copying the code from a previous shape and modifying it. The pieces should be initalized in the following **orientations**:



where the numbers refer to the index of each square in the 1D array of squares (Square[] square). The actual values of the indices don't really matter at this point, but they will become relevant when we talk about rotation. In the constructors of the pieces, take (r,c) to be the row and column indices of square[1].

- In the Game class, the piece instance variable must have a new type. What do you think it should be? Answer: Piece. The Game class shouldn't have to know about any specific implementation and should only work with the interface type.
- You'll need to modify the Game class so that it doesn't always make an LShape piece, but randomly chooses which shape to instantiate. (I strongly recommend creating 1 new shape at a time and testing.)

ROTATION: The current tetris piece needs to be able to rotate in quarter turns clockwise each time the down arrow key is pressed. As an example, see the figure below illustrating how the L-shaped piece rotates one full turn:



We will take the convention that all pieces rotate about their square at index = 1 (except for the gray square shape that shouldn't rotate). That is, after a rotation, the

square at index 1 should not have moved in the grid. All of the other squares of the piece will have moved clockwise by 90 degrees. The image below shows how the L shape piece rotates within the grid: notice that the square at index 1 stays at the same location.

To implement the rotation feature in your code, do the following:

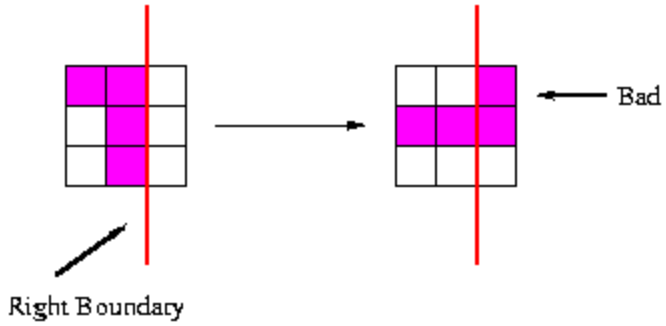
1. Add the abstract method `void rotate();` to the Piece interface.
2. Implement the rotate() method in AbstractPiece. A piece should only rotate if the grid squares that it would pass through and occupy are all empty and in addition are all within the boundaries of the grid (i.e. the row index is greater than or equal to zero and less than Grid.HEIGHT and the column index is greater than or equal to zero and less than Grid.WIDTH). To check if a square can move, you can model its path as a square path about the square at index 1 (and not as an actual circular path, in which case the square could sweep through part of a square of the grid.)

   (There is a public method `rotatePiece()` to the Game class. This method is very similar to the `movePiece()` method except that it tells the piece to rotate.)

3. Add a new case to the `keyPressed` method in the EventController class to react to the down arrow key.

   If you look at the constructor for the LShape, the (r, c) parameters define the position of the middle square on the long side. This is the square the shape is rotated around. In all cases except for the square gray shape, the square to rotate around is the one established by the (r,c) parameters.

   Here's an example of the sort of thing that boundary checking on rotation should prevent:



The other consideration when implementing this feature is "how to check if the rotate is legal? Are the appropriate squares on the grid empty?" Implementation is left as a developer's choice, but there are some guidelines you must follow:

- Notice that in the move() method of a piece, the piece queries the individual squares if they can move, and then tells them to move. Implement rotate the same way. **Find out if ALL individual squares of the shape can move as needed, then move them**. You can do this with the existing Square class interface, or you can add more methods to its public interface. In some rotations, a Square actually has to move up. You may want to add an UP constant (to the appropriate class) and modify the appropriate methods/comments in the Square class accordingly.

- **I recommended write a rotate function for LShape class first and test it out.**
- The gray square piece should never rotate. One approach would be to query the dynamic type of the piece being rotated in rotate of AbstractPiece and do nothing if it is the gray square piece. However, that would require an inelegant line of code using instanceof or getClass. An easier and cleaner approach is to override rotate in the class of the gray square piece and do nothing!

Note: As mentioned before, even though we may initially think that the rotation of a piece is specific to that piece and should be coded in its class, it is actually possible to write all of the rotation code in AbstractPiece. If you are not sure how, it’s okay to override rotate for each class.