CS 314H Algorithms and Data Structures — Fall 2017	Programming Assignment #4
Tetris	Due October 9/October 13, 2017

In this assignment you will work in pairs to implement a variant of Tetris, a game invented by Alexey Pazhitnov at the Moscow Academy of Science. This assignment will emphasize the idea of decomposing a large problem into smaller problems that can be independently tested. It will also emphasize design issues, as we will give you plenty of latitude in designing your solution.

The game of Tetris consists of a 2D grid and a stream of various-shaped pieces that fall, one at a time, onto the grid. The goal of the game is to rotate and move the pieces, so that as they fall, they are tightly packed and form entire rows. Once entire an row is formed, the row collapses, providing room for additional pieces to fall, thereby allowing the player to move and place additional pieces. The rest of the game is best described by playing it. *There are many variations of Tetris, but we will be using the version available at the following site as our definition of correctness:*

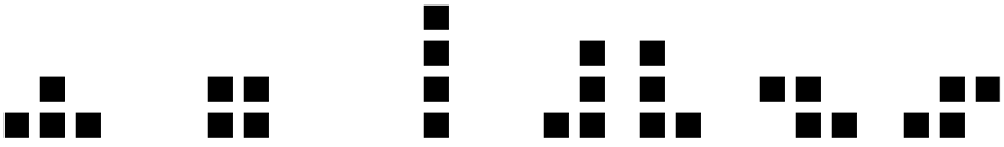
`http://tetris.com/play-tetris/`

We will provide a JTetris class, which is a functional Tetris player, and you will implement two classes, TetrisPiece and TetrisBoard, which will implement the actual game logic. To use the JTetris code to play the game, use the keys *A, S, D* to move the piece, the *Q* and *E* keys to rotate the piece, and use the *W* key to drop the piece.

This project is due on October 13^{th}, but you will receive bonus points if you submit largely functional versions of the TetrisPiece and TetrisBoard code by 5:00pm on October 9^{th}. You do *not* need to write a complete report to accompany your code, but you should make note of any correctness issues that you know of. Even if you submit something on October 9^{th}, you may continue to improve your code until your official submission on October 13^{th}.

1 The Pieces

In our version of Tetris, the pieces are composed of a connected non-zero number of blocks arranged in a grid. We will be using the standard 7 tetrominoes:



The T The Square The Stick The L The Dog

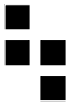
The provided code and interfaces lay out some conventions to follow in your implementation of these Tetris pieces, which are important for having a properly compliant implementation.

1.1 The Body

A piece is represented by the coordinates of its blocks, which are known as the *body* of the piece. Each piece has its own coordinate system with its (0,0) origin in the lower left hand corner of the rectangle that encloses the body. The coordinates of blocks in the body are relative to that piece's origin. Thus, the coordinates of the four points of the Square piece are as shown below:

```
(0,0)  <= the lower left-hand block
(0,1)  <= the upper left-hand block
(1,0)  <= the lower right-hand block
(1,1)  <= the upper right-hand block
```

Notice that not all pieces will have a block at (0,0). For example, the body of the following rotation of the Right Dog has the body as shown below:



```
[(0,1), (0,2), (1,0), (1,1)]
```

A piece is completely defined by its body—all of its other characteristics, such as its height and width, can be computed from the body. The Right Dog above has a width of 2 and a height of 3.

1.2 The Skirt

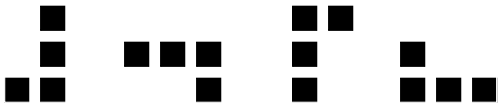
You will find it useful to maintain the *skirt* for each piece, which is the lowest y extent of each column in the piece. The skirt will be represented as an array of integers, which is as long as the piece is wide; for example, the skirt for the dog above is [1, 0]. All given Tetris pieces will be connected so you do not need to consider the case where there is an empty column.

1.3 The Rotations

Different versions of Tetris have different rules for rotations, but as specified in the Tetris Guidelines, your version will use the Super Rotation System (SRS). A reference to SRS as it is defined for the canonical pieces can be found here:

`https://tetris.wiki/SRS`

Note: Each rotation of piece should be represented as a separate Piece object, as you will see in the TetrisPiece section. For example, the figure below shows the four rotations of the L piece; each of these rotations should be represented by a different Piece object.



1.4 Wall-Kicks

Wall-kicks are a method of handling obstructed rotations in which a piece "kicks off" the wall or off of placed blocks to avoid going out of bounds or colliding with other pieces. Wall-kicks are usually implemented by attempting to shift the piece into a nearby empty space (based on some specific offsets and rules) if directly rotating it would cause it to collide with something.

The Super Rotation System (SRS) mentioned above provides standard rules for wall-kicks which you will be expected to implement. Note that properly handling wall-kicks is the responsibility of the GameBoard (when it resolves rotation actions), not of the piece itself.

2 The `TetrisPiece` Class

You should write the TetrisPiece class, which extends the abstract class Piece. The Piece class has two important members that your TetrisPiece class will inherit. The first is the `parsePoints` method, which takes a String representation of a piece's body and returns an array of Point objects which represent the same body. The second member is the instance variable `next`, which refers to another Piece object. You'll use `next` to chain Piece objects together, forming a circularly linked list of Piece objects which includes all the various rotations of a piece.

You will need to implement the static factory constructor for TetrisPiece, as well as the remaining abstract methods on the Piece object—`getWidth()`, `getHeight()`, `getBody()`, `getSkirt()`, and `equals()`. **Many of the methods that you implement are accessor methods whose details depend on design decisions that you make.** Each of these accessor methods should execute in constant time, so make an effort to perform any computation when a Piece is created. You may want to write helper methods to handle common repetitive tasks, such as generating all the rotations of an arbitrary piece.

3 The Board

The board represents the state of a Tetris board. Its most obvious feature is the grid, a 2D array of of booleans that indicates whether a square is filled. The lower left corner is position (0,0), with the x dimension increasing to the right and the y dimension increasing upwards. Filled squares are represented by a true value in the grid. In your Tetris game, the TetrisBoard class does most of the work:

- It stores the current state of the Tetris board.
- It provides support for the common operations that a client module (the player) needs to build a GUI version of the game. Namely, it adds pieces to the board, it lets pieces fall gracefully downwards, and it detects various conditions about the board.
- It will be used to analyze the state of the game for various Tetris AI.

Because of this, it needs to be efficient. As with the TetrisPiece class, try and do all of the computation as early as possible so that all of the accessor methods—`getLastAction()`, `getLastResult()`, `getRowsCleared()`, `getWidth()`, `getHeight()`, `getMaxHeight()`, `getRowWidth()`, `getColumnHeight()`, `getGrid()`—can be implemented in constant time.

3.1 Game Logic

While the client of the `Board` is responsible for deciding what moves the pieces make, the internal logic of the `Board` is responsible for enforcing legal moves and making sure that the appropriate reactions take place. When the client tries to move a piece, the movement is only successful if the piece is moving into empty space (or if a wall-kick can be applied to move the piece into empty space); if it hits something to the left or right then no movement occurs, and it if hits something below it then the piece is placed and cannot move anymore. When a line the width of the board is created out of placed pieces, it disappears, and any block above the deleted line will fall. Some games use different algorithms to determine how leftover pieces fall after a row is cleared, but in our Tetris game, blocks should only shift down by a distance exactly equal to the height of the cleared rows below them. This rule will sometimes cause pieces to appear to be "floating" above holes on the inside of the placed blocks.

The game client itself is responsible for things such maintain the current score and determining whether the player has won or lost. If you are curious about how these are implemented, you can look into the provided code, but you do not need to implement them yourself.

3.2 The Action/Result Abstraction

Most of the game's interactions with the board are through the `move()` method, which takes in a `Board.Action` as its parameter and returns a `Board.Result` when it is finished. During most of these moves, there will be a "current piece" on the board, which is controlled by the player.

These are the meanings of each of the `Action` enum values:

`LEFT`	Move the current piece to the left, unless it hits a wall or another piece.
`RIGHT`	Move the current piece to the right, unless it hits a wall or another piece.
`DOWN`	Move the current piece down, causing it to be placed if it has something directly below it
`DROP`	Cause the current piece to fall until it is placed.
`CLOCKWISE`	Rotate the current piece clockwise, applying a wall-kick if necessary, unless there is no space for the blocks of the piece after it is rotated.
`COUNTERCLOCKWISE`	Rotate the current piece counterclockwise, applying a wall-kick if necessary, unless there is no space for the blocks of the piece after it is rotated.
`NOTHING`	Do nothing.

And these are the meanings of each of the `Result` enum values:

`SUCCESS`	The last action was successful.
`OUT_BOUNDS`	The last action tried to move the current piece off of the board or into another placed piece (and wall-kicks could not successfully move the piece into empty space), so nothing was moved.
`NO_PIECE`	There is not a piece currently on the board to move. This lets the board's client know to add a new piece via `nextPiece()`.
`PLACE`	The last move caused the current piece to be placed. This result is a combination of `SUCCESS` and `NO_PIECE`

3.3 Design Considerations

You may design the internals of your Board however you wish, but be sure to justify your design decisions. Below are some questions that you might want to consider and to discuss in your report.

- What kinds of information do you need to store for the `Board` accessor methods (listed above) to run in constant time?
- Of the information listed in response to the previous question, how often does each one change?
- What attributes of the `Piece` object are important in calculating `dropHeight()`?
- How much of the information that you store about a `Board` needs to match another `Board` before they are equal?
- When implementing `testMove()`, should the new and old `Board` objects share any data?
- Does any `Action` always return a certain `Result`?
- When should rows be cleared?
- What is the most efficient way to clear rows?
- Are there any data structures in Java's standard library that are useful in building a `Board`?

4 Building a Brain

Perhaps the most interesting part of this assignment is the task of creating a good Tetris brain. Your Brain is free to use whatever approach and tactics you wish as long as they abide by the simple Brain interface.

The Brain interface defines the `nextMove()` method that computes what it thinks is the best available move for a given board. There are many tactics you could use to decide the next move for a given board, but the most common tactic is to enumerate the possible end locations for the active piece using the board's `testMove()` method, to use characteristics about each board to rank them, and to then select moves that will get you to the best possible board state for that piece.

The `LameBrain` class uses the above approach and has a very uninteresting way of ranking some boards as better than others. It uses two methods, `enumerateOptions()` and `scoreBoard()` to perform this work; both of the methods have fairly simple implementations and could be easily improved.

Your report should include an extensive explanation for your brain's strategy and its behavior. Remember to cite any AI related material you use as a resource.

To test your brain, you should write a JBrainTetris class that extends JTetris, overriding methods as necessary to get it to use your brain instead of user input. In all other respects, you should preserve the same behavior. Use inheritance where appropriate to reuse as much code as possible.

5 Testing

Much like the Critter assignment, you should implement some kind of a test harness to test your code. Think about how the real-time nature of the game makes it difficult to test your code, and try to isolate particular states within the game that might have interesting behavior. You might find that your `Piece`, `Board` and `Brain` each requires different methods of testing, as each one has different use cases and different expectations of its client code. Feel free to make use of JUnit and any other publicly available testing libraries, if they simplify the design or implementation of your test harness.

6 Karma

There are plenty of exciting changes you can make to this project! If your karma project changes the rules of the game, you should implement them separately from the required components. We wouldn't want your program to fail our tests because you changed the rules and forgot about it.

6.1 Written Rules of Tetris

In this assignment, we have defined the rules of Tetris by providing a reference implementation of the game and a few clarifications about rotations and wall-kicks. It would probably be more useful to have a full written specification of the rules. For significant Karma and possible posterity and fame, create a concise, clear, and comprehensive set of rules. If done right, such rules can not only help future generations of honors students as they complete this assignment, but they can help you better understand the design of your solution.

6.2 Adversary Tetris

For fun, you can use an adversary to increase the difficulty of the game. The adversary attempts to make life more difficult for the player by picking the "worst" possible next piece. To do this you'll need to create another subclass of JTetris that overrides the `pickNextPiece()` method, so that the adversary gets to cruelly pick the next piece. The adversary can be implemented with a little code that uses a brain to do the work. Think about which parts of your brain might be useful in judging pieces and then instead of picking the best, give the player the worst.

For fun, try the classic battle of good vs. evil and have the brain play the adversary.

6.3 Piece Holding

Many versions of Tetris allow the player to "hold" a piece for later. Implement the `HOLD` action for your board and then change `JTetris` to allow the player to hit another button to cause this action. If you want to get really fancy, try modifying the UI to allow the player to see the "held" piece.

6.4 Genetic Programming

If you want to challenge yourself, explore the notion of genetic algorithms, which uses biological evolution as a metaphor for optimization. The basic idea is to define a search space as set of genes, which mate and randomly mutate; an evaluation function favors the propagation of the better genes, ie, those that do better on the evaluation scores, which in your case will be the Tetris scores. With these ideas, see if you can use genetic algorithms to evolve a better brain.

7 What to Turn In

This assignment has a lot of code going into different files, so be sure to keep track of all of them:

- TetrisPiece.java
- TetrisBoard.java
- Your Tetris-playing Brain implementation.
- JBrainTetris.java
- Any classes you implement as part of Karma.
- Any helper classes you make for the above classes.
- All of the test code you write for the above classes.

If you want to submit code at the bonus deadline, you only need to supply implementations of TetrisPiece and TetrisBoard; there is no need to submit a report, Brain, or testing code at that deadline.

As always, you will also be turning in a report; include in your report a log of your time spent in the various aspects of this assignment—design, implementation, and debugging—along with the time spent driving or working separately. In your report, pay special attention to issues of decomposition, abstraction, and testing. What, if anything, have you learned about testing in this assignment?

As usual, all assignments are due at 5:00pm on the due date.

Acknowledgments. This assignment was originally produced by Nick Parlante of Stanford University. It has been modified by Matthew Alden, Walter Chang, Josh Eversmann, and Tres Brenan.