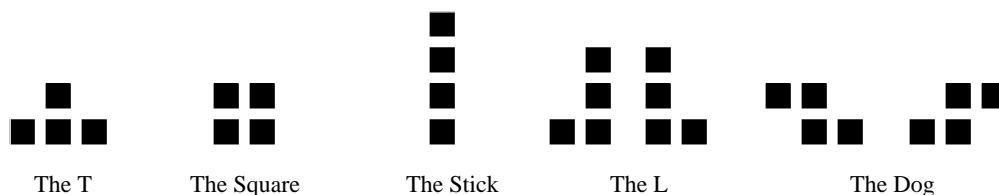


In this assignment you will work in pairs to implement Tetris, a game invented by Alexey Pazhitnov at the Moscow Academy of Science. This assignment will emphasize the idea of decomposing a large problem into smaller problems that can be independently tested. This assignment will also ask you to build portions of a very simple graphical user interface, and it will also require you to write portions of code that are very efficient. The first part of the assignment builds the TetrisPiece class and will be due **October 3**. The second part builds the TetrisBoard class, along with some other open-ended goodies, and will be due **October 10**. The first deadline represents considerably less than half of the work, so we encourage you to finish Part 1 as soon as possible so that you can give yourself more than a week for Part 2.

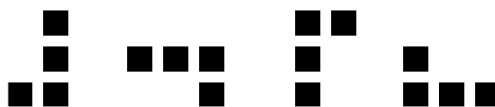
The game of Tetris consists of a 2D grid and a stream of various-shaped pieces that fall, one at a time, onto the grid. The goal of the game is to rotate and move the pieces, so that as they fall, they are tightly packed and form entire rows. The rest of the game is best described by playing it (be sure to play the traditional version of the game, in which blocks can only move down by a distance equal to the height of the rows that are cleared below them). If you have never played Tetris before, numerous versions can be found online.

1 The Pieces

There are seven pieces in standard Tetris. As shown below, each piece consists of four blocks.



A piece can be rotated counter-clockwise 90 degrees to form another piece. Of course, enough rotations get you back to the original piece—for example, rotating a Dog twice brings you back to the original state. Essentially, each Tetris piece belongs to a family of between one and four distinct rotations, and your program should represent the distinct rotations as distinct objects. The Square has one member, the Dogs have two, and the L's have four. For example, the following figure shows the four rotations of the left-handed L, which will be represented by 4 Piece objects.



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 x coordinate in the piece. The skirt will be represented by an array of integers, which is as long as the piece is wide. The skirt for the Dog above is (1,0). We will assume that pieces do not have holes in them—that is, for every x position in the piece's coordinate system, there is at least one block in the piece for that x.

1.3 Rotations

The Piece class needs to provide a way for clients to access the various piece rotations. The client can ask each piece for a reference to the *next rotation*, which yields a reference to a Piece object that represents the next counter-clockwise rotation. Note this immutable paradigm—rather than provide a `rotate()` method that changes the state of the Piece, the Piece objects are read-only, and the client can iterate over different instances of them. The String class is another example of this immutable paradigm.

1.4 Rotation Strategy

The overall piece rotation strategy uses a simple, static array that contains the first rotation for each of the 7 pieces. Each of the first pieces is the first node in a small circularly linked list of rotations of that piece. The client uses the `nextRotation()` method to iterate through all rotations of a Tetris piece. The array is allocated the first time the client calls `getPieces()`; thus the piece is only built when it's actually used.

1.5 Rotation Tactics

You will need to devise an algorithm to perform the actual rotation. Get a nice sharp pencil. Draw a piece and its rotation. Write the coordinates of both bodies. Think about the transformation that converts from a body to the rotated body. The transformation uses reflections around various axes.

1.6 Private Helpers

You will want private methods to compute the rotation of pieces and to place all of the rotations onto a list. The computation of the width, height, and skirt are needed when making new pieces and when computing rotations, but you should not have two copies of the code.

1.7 Generality

You should use a single `Piece` class to represent all of the different pieces, distinguished only by the different state in their body arrays. The code should be general enough to deal with body arrays of different sizes, so the constant 4 should not be sprinkled throughout your code.

1.8 The `TetrisPiece` Class

You should write the `TetrisPiece` class, which extends the abstract class `Piece`. The `Piece` class has three important members that your `TetrisPiece` class will inherit. The first is the method `parsePoints`, 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 class variable `pieceStrings`, which contains the `String` representations of the seven kinds of pieces used to play Tetris. The last 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.

In addition to the abstract methods declared in `Piece`, which you'll need to implement in `TetrisPiece`, you also need to write the following methods: (1) a private constructor that takes an array of `Point` objects and constructs a `TetrisPiece` object with the body specified by those `Points`, and (2) a static method `getPieces` which returns an array of all the possible un-rotated Tetris Pieces. The `getPieces` method has a big job, constructing all the un-rotated `Piece` objects (recall the `pieceStrings` and `parsePoints` members) as well as setting up the circularly linked list of rotations for each type of piece. For efficiency, the `getPieces` method should do all this work only the first time it is called. If it were called more than once, it should not do the same work all over again, but simply return the array of previously constructed Pieces.

You may want to write helper methods to handle common repetitive tasks, such as generating all the rotations of an arbitrary piece.

1.9 The `JPieceTest` Class

The last part of the `Piece` class is the test program that draws all of the pieces in a window, as shown in Figure 1. The details of how your `JPieceTest` should draw your pieces are as follows (see Figure 2):

- Divide the component into 4 sections. Draw each distinct rotation in its own section, from left to right, stopping when there are no more distinct rotations.
- Allow space for a Square that is 4 blocks \times 4 blocks at the upper left of each section. Draw the blocks of the piece starting at the bottom of that 4 \times 4 Square. The Square won't fit the section exactly, since the section might be rectangular. This is so that the pieces don't run into each other. See Figure 2.
- When drawing the blocks of the piece, leave a one-pixel border, which is not filled in, around each block; this will leave some space around each block. Each block should be drawn as a black square, except the blocks that comprise the skirt, which should be drawn as yellow squares. Draw the blocks in yellow by bracketing it with `g.setColor(Color.yellow); <Draw block> g.setColor(Color.black);`
- At the bottom of the square, draw in red a string that indicates the width and height of the block, such as `w:3 h:2`.

We provide some skeleton code to get you started. You will need to complete the `paintComponent` and `drawPiece` methods.

1.10 The `PieceTest` Milestone

With `JPieceTest`, you should be able to load and compute all the piece rotations and see that your piece code is working correctly. **Reaching this milestone will complete Part 1.**

Note that `JPieceTest` and `TetrisPiece` are essentially independent—either one should work with alternative implementations of the other. If you implement the interfaces correctly and do not introduce unnecessary dependences between these classes, we should be able to substitute our reference `TetrisPiece` for yours and vice versa and expect everything to work.

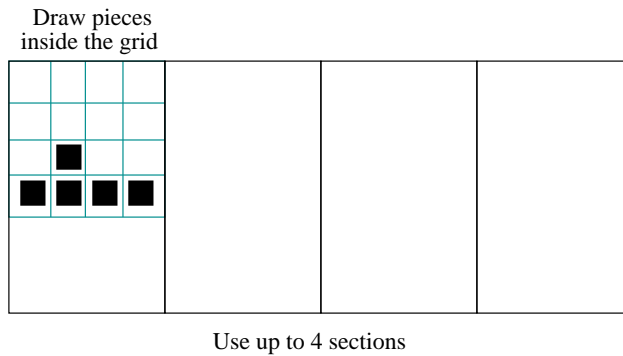


Figure 2: Details about drawing each piece.

2 The Board

In your Tetris game, the `TetrisBoard` class does most of the work and is the hardest part of the assignment:

- 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 performs all of the above quickly. The board implementation should be structured to do common operations quickly.

2.1 The Board Abstraction

The board represents the state of a Tetris board. Its most obvious feature is the grid, a 2D array 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. The `place()` method allows a piece to be added the grid, and the `clearRows()` method clears filled rows in the grid and shifts the relevant pieces downward.

Before describing the main Board methods, we first describe a handy auxiliary structure.

2.2 Widths and Heights

The secondary *widths* and *heights* structures make many operations efficient. As shown in Figure 3, the `widths` array stores the number of squares that are filled in each row, which allows the `place()` method (described below) to efficiently detect if the placement has caused a row to become filled. Likewise, the `heights` array stores the height to which each column has been filled. The height will be the index of the open spot which is just above the top filled spot in that column. The heights array allows the `dropHeight()` method to efficiently compute the location where a piece will come to rest when dropped in a particular column.

3 The TetrisBoard Class

You should write the `TetrisBoard` class, which implements the `Board` interface. In addition to the abstract methods declared in `Board`, you should write a constructor which takes parameters for the width and height of the Tetris board.

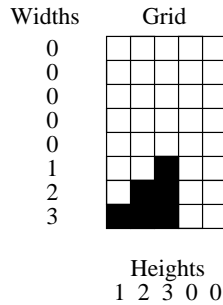


Figure 3: Illustration of the `widths` and `heights` arrays.

3.1 The Board Constructor

The constructor initializes an empty board. The board may be of any size, although the standard Tetris board is 10 wide and 20 high. The client code may create a taller board, such as 10×24 , to allow extra space at the top for the pieces to fall into play (the provided player code does this).

In Java, a 2D array is really a 1D array of references to 1D arrays. The expression `new boolean [width][height]` will allocate the whole grid.

3.2 `int place(piece, x,y)`

The `place()` method takes a piece and an (x,y) coordinate, and sets the piece onto the grid with the origin—i.e., the lower-left corner of the piece—at the (x,y) location of the board. The `undo()` method (described below) can remove the most recently placed piece.

The `place()` method returns `PLACE_OK` if the placement is successful. It returns `PLACE_ROW_FILLED` for a successful placement that also causes at least one row to become completely filled.

Error cases: It's possible for the client to request a bad placement—one where part of the piece falls outside the board or overlaps spots in the grid that are already filled. These bad placements leave the board in a partially invalid state. If part of the piece would fall out of bounds, this method should return `PLACE_OUT_BOUNDS`. Otherwise, if the piece overlaps spots that are already filled, this method should return `PLACE_BAD`. The client should be able to return the board to its valid, pre-placement state with a single invocation of the `undo()` method.

3.3 `clearRows()`

This method deletes each row that is completely filled, causing items above to shift downward. There may be multiple filled rows, and these rows might not be adjacent. New rows shifted in at the top of the board should be empty. This method is a complicated coding problem. You should probably make a drawing to help you chart your strategy. Use the `JBoardTest()` (described below) to generate a few of the weird row-clearing cases.

3.4 The Implementation

The slickest solution does everything in one pass—it copies each row down to its ultimate destination. A hint for implementing this as a single pass: The *To* row is the row you are copying down to. The *To* row starts at the bottom filled row and proceeds up one row at a time. The *From* row is the row you are copying from. The *From* row starts one row above the *To* row and skips over filled rows on its way up. The contents of the `widths` array needs to also be shifted down. Be convinced that your solution works and describe it in your report.

By knowing the maximum filled height of all of the columns, you can avoid needless copying of empty space at the top of the board. Also, the `heights` array will need to be recomputed after each row clearing. The new value for each column will be lower than the old value (not necessarily just 1 lower), so just start at the old value and iterate down to find the new height.

3.5 `int dropHeight(piece, x)`

The `DropHeight()` method computes the `y` value at which the origin (0,0) of a piece will come to rest if dropped in the given column from an infinite height. This method should use the `heights` array and the skirt of the piece to compute the `y` value quickly— $O(\text{piece-width})$ time. This method assumes that the piece falls straight down; it does not account for any movement of the piece during the drop.

4 The Undo Abstraction

The problem is made more difficult because the player (the client code) doesn't want to just add a sequence of pieces. The player instead wants to experiment with different locations and rotations. To support this experimentation, the board will implement a 1-deep undo facility. This facility will add significant complexity to the board implementation but will make the client's code simpler. This design illustrates an object oriented design principle: You will provide functionality that meets the client needs while hiding the complexity inside of the implementing class.

4.1 `undo()`

The board has a *committed* state, which is either true or false. Suppose that the board is originally in a committed state, which we will refer to as the *original* state of the board. The client may perform a single `place()` operation, which will change the board state as usual and set the committed state = false. The client may also perform a `clearRows()` operation, which still leaves the board's committed state = false. However, if the client performs an `undo()` operation, the board should return to its original state. If instead of the `undo()` operation, the client performs the `commit()` operation, the current state of the board will be marked as the committed state of the board, which means that the client will no longer be able to get back to the original board state.

Basically, the board gives the player the ability to place a single piece, perform one `clearRows()` operation, and still get back to its original state with a 1-deep undo capability. By performing a `commit()` operation, the player can perform additional `place()` and `clearRows()` operations.

Stated more formally, the rules are as follows:

- Initially, the board is in a committed state, and `committed = true`.
- The client may perform a single `place()` operation, which sets `committed = false`. The board must be in the committed state before the `place()` method is called, so it is not possible to call `place()` twice in a row.
- The client may perform a single `clearRows()` operation, which also sets `committed = false`.
- The client may do an `undo()` operation, which returns the board to its original committed state and sets `committed = true`.
- Alternately, the client may do a `commit()` operation, which keeps the board in its current state and sets `committed = true`.
- The client must either perform an `undo()` or `commit()` operation before performing another `place()` operation.
- When in the committed state, `commit()` and `undo()` operations have no effect.

Thus, to make a piece appear to fall, the client will execute code that looks something like this:

```
place the piece at the top of the board
<pause>
undo
place the piece one row lower
<pause>
undo
place the piece one row lower
. . .
```

```

detect that the piece has hit the bottom because place() returns PLACE_BAD
or PLACE_OUT_OF_BOUNDS
undo
place the piece back at its last valid position
commit
add a new piece to the top of the board
. . .

```

4.2 undo() Implementation

The `undo()` method is great for the client, but it complicates the `place()` and `clearRows()` methods. Here is one implementation strategy that uses the concept of backup data structures.

Backups. For every board data structure, which we will refer to as *primary* data structures, you should maintain a backup data structure of the same size. The `place()` and `clearRows()` methods can then copy the primary state to the backup before making changes. The `undo()` method can then restore the primary state from the backup when necessary.

Backing up Widths and Heights. For the `widths` and `heights` arrays, the board has backup arrays called `xwidths` and `xheights`. When `place()` is invoked, copy the current contents of the two arrays to their backups. Use `System.arraycopy(source, 0, dest, 0, length)`, which is a pretty efficient method.

Swap trick. For the undo operation, the obvious implementation would invoke `arraycopy()` to restore the old state, but it's much more efficient to just swap pointers, which means that the primary and backup data structures swap roles with each undo. This scheme not only avoids the copy, but it means that we never have to allocate more than a single primary and a single backup.

Backing up The Grid. The grid needs to also be backed up. The simplest strategy is to just back up all of the columns when the `place()` operation is performed. This is an acceptable strategy. In this case, no further backup is required for `clearRows()`, since the `place()` method has already backed up the entire grid.

4.3 Sanity Check

The board has considerable internal redundancy between the grid, the `widths`, the `heights`, and `maxHeight`. Write a `sanityCheck()` method that verifies the internal consistency of the board structures: The `widths` and `height` arrays should have the right numbers, and the `maxHeight` should be correct. There are *many* other checks that you could perform (be sure to discuss these in your report). Throw an exception if the board is in an inconsistent state—throw `new RuntimeException("description")`. Call the `sanityCheck()` method at the bottom of your `place()`, `clearRows()` and `undo()` methods. A static boolean called `DEBUG` should be used to control the use of `sanityCheck()`. When `DEBUG = true` the sanity check should be performed. Turn your project in with `DEBUG = false`. Put the sanity check in early to help you test and debug your code.

There's one bit of trickiness: Do not call `sanityCheck()` in the `place()` method if the placement is bad, since a bad placement represents a temporarily allowed inconsistent state.

4.4 Performance

The Board class has two design goals. First, it should provide services for the convenience of the client. Second, it should run reasonably fast. With respect to speed, here are the two primary considerations.

1. Accessors: `getRowWidth()`, `getColumnHeight()`, `getHeight()`, `getGrid()`, `dropHeight()`, and `getMaxHeight()` should all be made extremely fast, essentially constant time.
2. The `place()`, `clearRows()`, and `undo()` system can copy all of the arrays for backup and swap pointers for the undo operation. This is about as fast as you can make it.

4.5 The JBoardTest Milestone

Once you have the TetrisBoard class largely written, you can use the JBoardTest class for testing. This “test mule” (See Figure 4) is simply a GUI driver for the board interface. It lets you add pieces, move them around, place pieces, and clear rows. To support debugging, it also displays the hidden board state of the `widths` and `heights` arrays. Use the mule to try out basic test scenarios for your board. It’s much easier to use the test mule than to try to observe and debug your code while playing Tetris in real time. Note that the *drop* button will only work if there’s a straight-down vertical path for the piece to fall from at an infinite height. If there’s an obstructing overhang above the piece, drop will not do anything—see the source code for *drop* in JBoardTest.

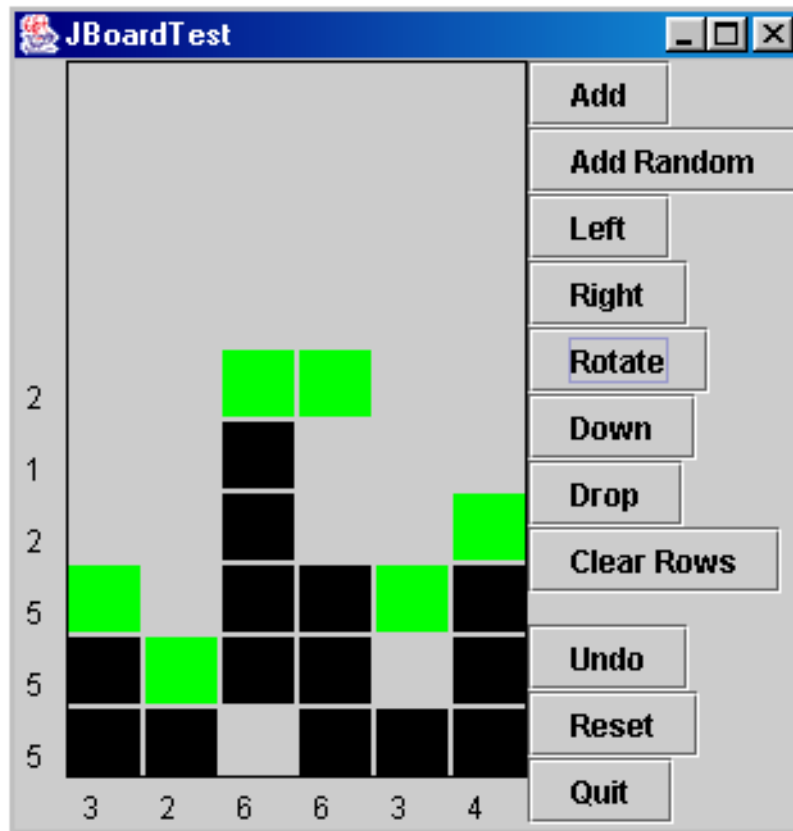


Figure 4: Screenshot of the test mule.

Although you will not be handing in JBoardTest, feel free to modify it with your own debugging code. For example, you might create a `printBoard()` method that prints out all of the state of the board to standard output, as this will allow you to examine log files that capture the state of the board over many time steps. Once you’ve tested and debugged your TetrisBoard, you’re ready to finish the Tetris game itself.

5 Tetris

The provided JTetris class is a functional Tetris player that uses your TetrisPiece and TetrisBoard classes to do the work. Use the keys `4`, `5`, `6` to move the piece, and use the key `0` to drop the piece. The *speed* slider adjusts the speed of the pieces. You will create a subclass of JTetris that uses an artificial brain to play the pieces as they fall. Finally, you will add the much needed adversary feature that allows you to vary the game’s level of difficulty.

5.1 Milestone—Basic Tetris Playing

You need to get your `TetrisBoard` and `TetrisPiece` code sufficiently debugged that `JTetris` can play Tetris. If it's too fast to debug, go back to the test mule. You should convince yourself that the `TetrisBoard` and `TetrisPiece` code are free of bugs before continuing with the next step.

5.2 Stress Test

The provided `JBrainTetris` gives you a simple-minded computer player with which to test your implementation.

Use the test mode (by giving the `test` parameter on the command line) to force `JBrainTetris` to use the fixed sequence of 100 pieces. If your program is correct, the test mode with the unchanged `LameBrain` and the fixed sequence of 100 pieces should lead to the exact board configuration shown below:

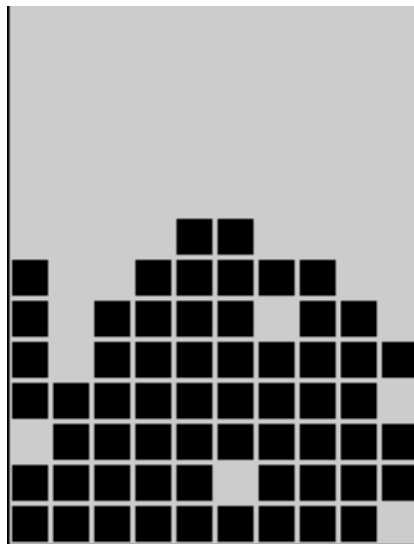


Figure 5: Screenshot of the test mule using `LameBrain` and the fixed test sequence.

This is an extremely rigorous test of your Board. Although there are only 100 different boards displayed, one for each piece, the brain has to explore thousands of boards that are never displayed on the screen. If the `getColumnHeight()`, `clearRows()`, or `undo()` methods is incorrect even once, the entire configuration the board could be drastically changed.

If the stress test is not coming out correctly, you could try the following:

- Look at the `JPieceTest` output to verify that the pieces are correct in every detail. (Hopefully you already did this long ago!)
- Put in additional sanity checks: Check that `clearRows()` changes the number of blocks by the correct number (which should be a multiple of the board width), check that the `undo()` method is restoring state exactly.
- Try to find out where your code deviates by trying shorter test sequences.

5.3 Understanding Tetris

Read the `JBrainTetris.java` code to make sure you understand how it works, because you will be writing a subclass of it. To help you get started, here are a few points worth noting:

- `tick()` is the bottleneck for moving the current piece.
- `computeNewPosition()` simply encapsulates the switching logic to determine the new (x,y,rotation) that is one move away from the current one.

- `tick()` detects that a piece has landed when it won't go down any more.
- If the command line argument `test` is present, the boolean `testMode` is set to `true`, in which case the game plays the same sequence of pieces every time. This mode is quite useful for debugging.

As usual for inheritance, your derived class should reuse as much inherited code as possible.

5.4 Building a Better 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 methods and tactics it deems are appropriate as long as it abides by the Brain interface.

The Brain interface defines the `bestMove()` method that computes what it thinks is the best available move for a given piece and a given board.

The provided `LameBrain` class is a simple implementation of the Brain interface. Take a look at `LameBrain.java` to see how simple it is. Given a piece, it tries playing the different rotations of that piece in all of the columns where it will fit. For each play, it uses a simple evaluation function, `rateBoard()`, to decide how desirable the resulting board is—blocks are bad, holes are bad. The brain uses the `Board.dropHeight()`, `place()`, and `undo()` methods to cycle through the different possible board configurations.

To create your brain, build a subclass of `LameBrain`. Two tactics that the default brain doesn't get right are (1) it doesn't avoid creating tall troughs that can only be filled by a Stick, and (2) it does not realize that things near the top edge are more important than things that are buried in lower layers. Finally, a lot of the parameters can be tuned to improve the brain. An ideal solution would follow a systematic approach to tuning these parameters (and an ideal report would explain this systematic approach).

We will not place restrictions on the strategies that your brain employs except that it must be more interesting and more clever than `LameBrain` and similar approaches.

5.5 Hooking up the new Brain

In order to test your brain, you will need to write a `JBetterBrainTetris` class that extends `JBrainTetris`. Override what you need to in order to get it to use your brain instead of `LameBrain`. In all other respects, you should preserve the same behavior. The changes involved should be very minimal.

Your report should include an extensive discussion on the strategies that your brain employs. Is your brain better than `LameBrain`? Does it eliminate more lines before dying than `LameBrain` or yourself? What are the strengths and weaknesses of your brain's strategy?

5.6 Adversary

For the last step, you will build an adversary that uses your Brain 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.

Create a subclass of `JTetris` called `JAdversaryTetris`. `JAdversaryTetris` should incorporate the following changes.

- Modify `createControlPanel` to add a label that says, “Adversary:”, to add a slider with the range 0..100 and initial value 0, and to add a status label that says, “ok”.
- Override the `pickNextPiece()` method. Generate a random number between 1 and 99. If the random number is greater than the slider's value, then the next piece should be chosen randomly as usual. But if the random value is less than the slider value, the adversary gets to cruelly pick the next piece. When the piece is chosen at random, the `setText()` method should set the status to “ok;” otherwise, it should set the status to “*ok*”.
- The adversary can be implemented with a little code that uses the brain to do the work. Loop through the array of pieces. For each piece, ask the brain what it thinks the best move is. Remember the piece that yielded the move with the worst score. When you've figured out which is the worst piece—the piece for which the best possible move is bad, then that's the piece the player gets!

- Abstraction is key here. The Brain interface is so clean that it can be used for both good and evil. Notice how important the speed of the Board class is here. There are about 25 rotations for each piece on a board, so the adversary needs to be able to evaluate $7 \times 25 = 175$ boards in the tiny pause after a piece has landed and the next piece is “randomly” chosen, so we need to be able to process the placements very quickly.
- Try your adversary on a friend. Leave the adversary at around 40% and the speed nice and slow. Turn the adversary up to 100% and see if he or she complains about the game being “not fair.”
- For fun, try the classic battle of good vs. evil and have the brain play the adversary.

5.7 Notes on Correctness

- We should be able to run `JPieceTest` and it should look right.
- Your board should have the correct internal structure, with a functioning `sanityCheck()` and with efficient board manipulation methods as discussed earlier.
- We should be able to play Tetris using the keyboard in the usual way, or we should be able to watch your `JBetterBrainTetris` play.
- We should be able to run your application with `java JBrainTetris test` to perform the stress test.
- We should be able to use the slider to activate the adversary feature in `JAdversaryTetris`.
- We should be able to swap out parts of your code with parts of our code (pieces, board, etc) that implements the same interface and have it function as expected. This should be a freebie if you stick to the interfaces.

6 Extra Fun

- You can do any number of things to make your brain more intelligent. For instance, your brain might “play it safe” by choosing a placement that will allow for better opportunities on the next turn regardless of the next piece chosen. Such a brain might be more resistant to a malicious adversary of lesser intelligence. Similarly, the adversary can choose pieces to minimize opportunities up to n turns out.

There is a technique in artificial intelligence known as alpha-beta pruning to make searching the space of moves more practical. With each turn, the player is attempting to maximize the objective function while the adversary is attempting to minimize the objective function. Thus, for a 1-level search, the player tries to pick the move that gives him the best board assuming the adversary then picks the worst piece. The trick is that if you see a better move than the best move from the worst piece you’ve seen so far, you don’t have to consider it, as the adversary will not give you that opportunity. This lets you skip large chunks of the game tree, typically allowing you to look twice as far ahead in the same time.

For more information on alpha-beta pruning, do a search or come to office hours.

- Tetris strategies might be different if the board had different topological characteristics. Consider a Tetris board where the left and right sides are “taped together” and wrap around. How does this change the strategy of your brain? If you implement this, you will need to implement a separate board as well, since this is a pretty serious change of game rules.
- `JTetris` keeps “score” by counting the number of lines eliminated. What if extra credit was given for eliminating lines simultaneously? For example, suppose eliminating one line was worth one, eliminating two was worth four, eliminating three was worth nine, and eliminating four worth a whopping sixteen. How does this change your brain’s strategy? Maybe having long troughs that can only be filled by a stick might not be so bad! Does having an adversary change the situation? Note that `LameBrain` isn’t smart enough to grasp this, as it doesn’t try to build for future opportunities.

- 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.

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 the stress test because you changed the rules and forgot about it.

7 What to Turn In

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 part of the assignment? What sort of testing is not possible with the default testing implementation? What improvements could (or did) you make to the `JPieceTest` class and `JBoardTest` class?

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

7.1 Part 1

Report: Write a report in the usual format.

Source code: Turn in only `TetrisPiece.java` and `JPieceTest.java`. Other source files will be disregarded. Submit this as `Proj4a.zip`

7.2 Part 2

Report: Write a report in the usual format.

Source code: Turn in only `TetrisBoard.java`, `JBetterBrainTetris.java`, `JAdversaryTetris.java`, your brain code, and your `TetrisPiece.java` from Part 1. Other non-karma source files will be disregarded. Submit this as `Proj4b.zip`

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