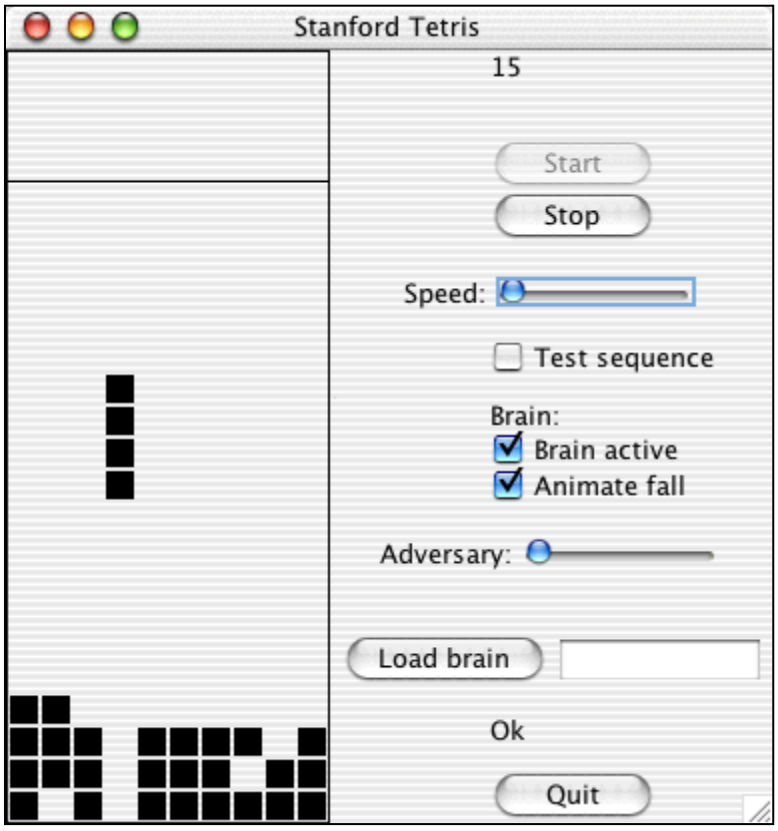
HW2 Tetris

Assignment by Nick Parlante

For HW2 you will build up a set of classes for Tetris. This assignment emphasizes the basic Divide and Conquer strength of OOP design -- using encapsulation to divide a big scary problem into several slightly less scary and independently testable problems. Tetris is a large project, so the modular OOP design matters.

This assignment is due at Midnight ending the evening of Monday January 30th. Do not underestimate the difficulty of this assignment – it is much, much longer than HW1 was.

For reasons that will become clear later, there is a theme of efficiency in this design. We are not just writing classes that implement Tetris. We are writing classes that implement Tetris **quickly**.

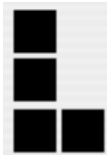
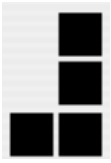


For us old-timers who grew up before the popularization of Doom and Quake, Tetris is one of the coolest things around. Try playing it for 27 hours and see if you don't agree. If you play Tetris enough, you may begin to have Tetris dreams (http://www.sciam.com/article.cfm? articleID=0001F172-55DA-1C75-9B81809EC588EF21, or just type "tetris dreams" into Google).

Part A -- Piece

There are seven pieces in standard Tetris.

The Stick  (also known as "the long skinny one")

The L  and its mirror L2 

The S  and its mirror S2 

The Square 

The Pyramid 

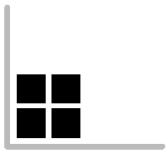
Each standard piece is composed of four blocks. The two "L" and "S" pieces are mirror images of each other, but we'll just think of them as similar but distinct pieces. A chemist might say that they where "isomers" or more accurately "enantiomers" (note: I only looked that word up in an effort to make the handout more impressive.).

A piece can be rotated 90 degrees counter-clockwise to yield another piece. Enough rotations get you back to the original piece — for example rotating an S twice brings you back to the original state. Essentially, each tetris piece belongs to a family of between one and four distinct rotations. The square has one, the S's have two, and the L's have four. For example, here are the four rotations (going counter-clockwise) of the L:

Our abstraction will be that a piece object represents a single Tetris piece in a single rotation, so the above diagram shows four different piece objects.

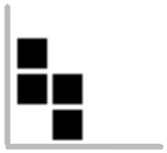
Body

A piece is defined by the coordinates of the blocks that make up the "body" of the piece. Each Piece has its own little coordinate system with its (0,0) origin in the lower left hand corner and with the piece positioned as low and as left as possible within the coordinate system. So, the square tetris piece has the body coordinates:



```
(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
```

In rare cases, a piece will not even have a block at (0, 0). For example, this rotation of the S…



has the body:

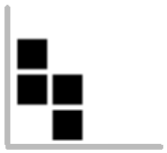
```
{(0,1),(0,2),(1,0),(1,1)}
```

A piece is completely defined by its body -- all its other qualities, such as its height and width, can be computed from the body.

We will measure the "width" and "height" of a piece by the rightmost and topmost blocks in its body. The above S has a width of 2 and height of 3. Another quality that turns out to be useful for playing Tetris quickly is the "skirt" of a piece....

Skirt

The skirt is an int[] array with as many elements as the piece is wide. The skirt stores the lowest y value that appears in the body for each x value in the piece. The x values are the index into the array. So for example, for this S...



the skirt is {1, 0}. That is, at x=0, the lowest y value in the body is y=1, and for x=1, the lowest y value is y=0. We assume that pieces do not have holes in them -- for every x in the piece coordinate system, there is at least one block in the body for that x.

Rotation, Version 1

The Piece class needs to provide a way for clients to access the various piece rotations. The piece class supports rotations in two ways. The first and most straightforward way is the computeNextRotation() method, which computes and returns a new piece which represents a 90 degrees counter-clockwise rotation of the receiver piece. Note that the piece class uses the "immutable" style -- there is no method that **changes** the receiver piece. Instead, computeNextRotation() creates and returns a new piece.

Rotation Code

For computeNextRotation(), work out an algorithm that, given a piece, computes the counterclockwise rotation of that piece. Draw a piece and its rotation and list out the x,y values of their body points. Get a nice, sharp pencil, and think about a sequence of operations on the body points of the first body that will yield the second body. Hint: write out the coordinates of each of the body points before and after the rotation and determine what the formula for calculating the new x and new y is.

Rotation, Version 2

The problem with computeNextRotation() is that it's a little costly if we want very quickly to look at all of the rotations of a piece. It's costly because it re-computes the rotated body every time, and it allocates a new piece with "new" every time. Since the piece objects are immutable, we can just compute all the rotations once, and then just store them all somewhere.

To do this, we will use a ".next" pointer in each piece that points to its pre-computed next counter-clockwise rotation. The list is circular, so the .next pointer in the last rotation points back to the first rotation. The method fastRotation() just returns the .next pointer. So starting with any piece in the list, with fastRotation() we can quickly cycle through all its rotations.

For a newly created piece, the .next pointers are null. The method makeFastRotations() should start with a single piece, and create and wire together the whole list of its rotations around the given piece.

The Piece class contains a single, static "pieces" array that contains the "first" rotation for each of the 7 pieces. The array is set up (see the starter code) with a call to makeFastRotations(), so all the rotations are linked off the first for each piece.

The array is allocated the first time the client calls getPieces(). This trick is called "lazy evaluation" -- build the thing only when it's actually used. The array is an example of a static variable, one copy shared by all the piece instances. In OOP, this is sometimes called the "singleton" pattern, since there is only one instance of the object and clients are always given a pointer to that once instance.

Piece.java Code

The Piece.java starter files has a few simple things filled in and it includes the prototypes for the public methods you need to implement. Do not change the public prototypes or constants, so your Piece will fit in with the later components and with our testing code. You will want to add your own private helper methods that can have whatever prototypes you like. The end of this section also describes setting up unit-tests for the Piece class.

Piece.java

Here are a few notes on the features you will see in the Piece.java starter file.

The provided TPoint class is a simple struct class that contains an x,y and supports equals() and toString().

Constructors

The main constructor takes an array of TPoint, and uses that as the basis for the body of the new piece. The constructor may assume that there are no duplicate points in the passed in TPoint array. There is a provided alternate constructor that takes a String like "0 0 0 1 0 2 1 0" and parses it to get the points and then calls the main constructor. A parsePoints() method is provided in the starter file for that case.

Piece.equals()

It's standard for the .equals() code to start with an "==this" test and an "(object **instanceof** Piece)" test for the passed in object. The starter code has this standard code already included.

Private Helpers

As usual, use decomposition to divide the computation into reasonable sized pieces and avoid code duplication. You may declare any helper methods "private".

Generality

Our strategy uses a single Piece class to represent all 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 -- the constant "4" should not be used in any special way in your code.

Unit Testing

Create a PieceTest JUnit class (use Eclipse command: New > JUnit Test Case). Look at all public methods supported by Piece: getWidth(), getHeight(), getSkirt(), fastRotation(), equals() -- the output from each of these should be checked a few times. Rather than check the raw output of getBody(), it's easier to check the computed width, height, skirt, etc. are derived from the body. Likewise, checking that fastRotation() is correct works as a check of computeNextRotation().

Basic testing plan: Get a few difference start pieces -- create some with the constructor and get some from the getPieces() array. Test some of the attributes of the start pieces. Then get some other pieces that are rotated a few times away from the start pieces, and check the attributes of the rotated pieces. Also check that the getPieces() structure looks correct and has the right number of rotations for a couple pieces. You can use the constants, such as PYRAMID, to access specific pieces in the array. The pictures on page 2 show the first "root" rotation of each piece.

Work on your unit-tests **before** getting in too deep writing your Piece code. Writing the tests first helps you get started thinking about Piece methods and what the various rotations, skirt, etc. cases look like before you write the code. Then, having the tests in place makes it easy to see if your code is working as you build out the code for the various cases.

Your unit-tests are part of the homework deliverable, and we will look at them in two ways...

- First there's just a baseline of having a decent looking unit-test: The Piece unit test should look at 5 or more different piece objects, and should call and check the output of each method -- getWidth(), getHeight(), getSkirt(), fastRotation(), equals() -- at least 5 times for each method.

Then, we try running your unit-test against various Piece classes we have to see how it sifts the good from the ugly.

- When run against a correct Piece class, the unit test should not report any problems. This is very important.

- We have many Piece implementations that have one or more bugs in them. When run against a buggy piece, it's good if your unit-test can notice that the piece has a problem. Your unit test does not need to articulate what the bug is -- just discern correct from buggy piece implementations.

Of course the other advantage to doing a good job with the piece unit-tests is that your own piece class has a great chance of being correct, since you will have worked it over so well with your unit tests.

Part A Milestone

You have a working Piece class that fills the pieces array with working tetris pieces. You have unit tests try all the different piece methods.

Part B -- Board

In the OOP system that makes up a tetris game, the board class does most of the work...

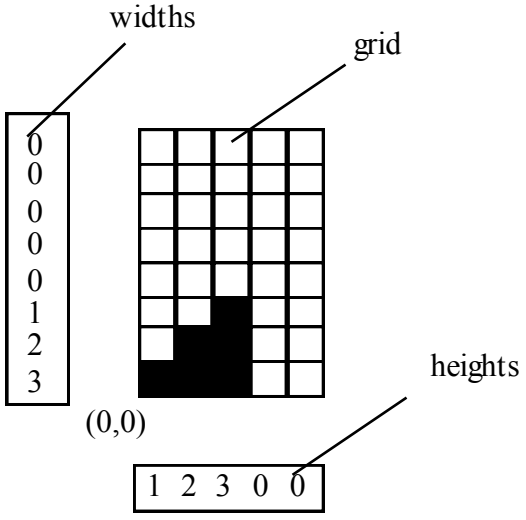
- Store the current state of a tetris board.
- Provide support for the common operations that a client "player" module needs to build a GUI version of the game: add pieces to the board, let pieces gradually fall downward, detect various conditions about the board. The player code is provided for you, but you need to implement Board.
- Perform all of the above quickly. Our board implementation will be structured to do common operations quickly. Speed will turn out to be important.

Board Abstraction

The board represents the state of a tetris board. Its most obvious feature is the "grid" – a 2-d array of booleans that stores which spots are filled. The lower left-corner is (0,0) with X increasing to the right and Y increasing upwards. Filled spots are represented by a true value in the grid. The place() operation (below) supports adding a piece into the grid, and the clearRows() operation clears filled rows in the grid and shifts things down.

Widths and Heights

The secondary "widths" and "heights" structures make many operations more efficient. The **widths** array stores how many filled spots there are in each row. This allows the clearRows() operation to know when a row is full. 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() operation to compute efficiently where a piece will come to rest when dropped in a particular column.



The main board methods are the constructor, place(), clearRows(), and dropHeight()...

Constructor

The constructor initializes a new empty board. The board may be any size, although the standard Tetris board is 10 wide and 20 high. The client code may create a taller board, such as 10x24, to allow extra space at the top for the pieces to fall into play (our player code does this).

In Java, a 2-d array is really just a 1-d array of pointers to another set of 1-d arrays The expression "new boolean[width][height]" will allocate the whole grid. With respect to undo() (below), the board must be in the committed state when it is created.

int place(piece, x, y)

Place() takes a piece, and an (x,y), and sets the piece into the grid with its origin at the given location in the board. The undo() operation (below) can remove the most recently placed piece.

Place() returns PLACE_OK for a successful placement, and PLACE_ROW_FILLED for a successful placement that also caused at least one row to become filled.

Error cases: It's possible for the client to request a "bad" placement -- one where part of the piece falls outside of the board or that overlaps blocks in the grid that are already filled. First, if part of the piece would fall out of bounds return PLACE_OUT_BOUNDS. Otherwise, if the piece overlaps already filled spots (discovered while modifying the board), return PLACE_BAD. A bad placement may leave the board in a partially invalid state -- the piece has been partly but not completely added for example. In all cases, the client may return the board to its valid, pre-placement state with a single undo() (described below). With respect to undo(), the board must be in the committed state before place() is called.

As place() iterates through the body of the piece, it should update widths[], heights[], and maxHeight as it goes. That way, no separate loop is required and the relevant piece and board memory is only touched once. Likewise, notice if the result is going to be PLACE_ROW_FILLED a the time you are updating the widths array, not by going back at looking at the widths arrays again later on.

int clearRows()

Delete each row that is filled all the way across, causing things above to shift down, and returns a count of the number of rows deleted. New rows shifted in at the top of the board should be empty. There may be multiple filled rows, and they may not be adjacent. This is a complicated little coding problem. Make a drawing to chart out your strategy. Note that classic tetris does not have "gravity" where blocks continue falling into empty spaces. Instead, it's very simple: every row above a deleted row moves down exactly one row.

clearRows() Implementation

The slickest solution does the whole thing in one pass — copying each row down to its ultimate destination, although it's ok if your code needs to make multiple passes. Single-pass hint: 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 goes up one row at a time, except it skips over filled rows on its way up. The contents of the widths array needs to be shifted down also. Empty rows need to be shifted in up at the very top of the board.

Since you know the maximum filled height of all the columns, you can avoid needless copying of empty space at the top of the board. This is a good optimization, since very often in tetris, the board is mostly empty.

int dropHeight(piece, x)

DropHeight() computes the y value where the origin (0,0) of a piece will come to rest if the piece is dropped straight down with its origin at the given x from infinitely high. DropHeight() should use the heights array and the skirt of the piece to compute the y value in O(piece_width) time. A single for(x=0; x<piece.width;x++) loop can look at the piece skirt and the board heights to compute the y where the origin of the piece will come to rest. DropHeight() assumes the piece falls straight down from above the board -- it does not account for moving the piece around things during the drop.

undo() Abstraction

The client code does not want to just add a sequence of pieces. The client code wants to experiment with adding different pieces. To support this client use case, the board will implement a 1-deep undo facility. This will be a significant complication to the board implementation that makes the client's life easier.

Functionality that meets the client needs while hiding the complexity inside the implementing class -- that's good OOP design in a nutshell.

undo()

The board has a "committed" state that is either true or false. Suppose at some point that the board is committed. We'll call this the "original" state of the board. The client may do a single place() operation. The place() operation changes the board state as usual, and sets committed=false. The client may also do a clearRows() operation. The board is still in the committed==false state. Now, if the client does an undo() operation, the board returns, somehow, to its original state. Alternately, instead of undo(), the client may do a commit() operation which marks the current state as the new committed state of the board. The commit() means we can no longer get back to the earlier "original" board state.

Here are the more formal rules...

- The board is in a "committed" state, and committed==true.
- The client may do a single place() operation which sets committed==false. The board must be in the committed state before place() is called, so it is not possible to call place() twice in succession.
- The client may do a single clearRows() operation, which sets committed==false. The client may or may not have called place() before clearRows() is called. It's a little silly to do a clearRows() without a place() first, but some path through the client logic might still do it, and it's our responsibility to still give correct results in that case.
- The client may then do an undo() operation that returns the board to its original committed state and sets committed==true. This is going backwards.
- Alternately, the client may do a commit() operation which keeps the board in its current state and sets committed==true. This is going forwards.
- The client must either undo() or commit() before doing another place().

Basically, the board gives the client the ability to do a place(), a clearRows(), and still get back to the original state. Alternately, the client may do a commit() which can be followed by further place() and clearRows() operations. We're giving the client a 1-deep undo capability.

Commit() and undo() operations when the board is already in the committed state silently do nothing. It can be convenient for the client code to commit() or undo() just to be sure before starting in with a place() sequence.

Client code that wants to have a piece appear to fall will do something like following...

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

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

Undo() Implementation

Undo() is great for the client, but it complicates place() and clearRows(). Here is one implementation strategy...

Backups

For every "main" board data structure, there is a parallel "backup" data structure of the same size -- for example, for the main "widths" array there's a backup "xWidths" array. Place() and clearRows() copy the main state to the backup if necessary before making changes. Undo() restores the main state from the backup.

Backup -- Copy

Use System.arraycopy(source, 0, dest, 0, length) to copy from the main to the backup arrays. System.arraycopy() is probably highly optimized by the JVM. Note that the 2-d grid is really essentially a 1-d array of pointers to 1-d arrays.

Undo -- Swap

For undo() the obvious thing would be to do an arraycopy() back the other way to restore the old state. But we can better than that. Cool trick: just swap the backup and main pointers. Suppose, for example, we want to restore the "widths" array with its backup "xWidths"...

```
// NO, not this way
System.arraycopy(xWidths, 0, widths, 0, widths.length); // NO


// This way -- just swap the pointers
int[] temp = widths;
widths = xWidths;
xWidths = temp;
```

This works very quickly. So the "main" and "backup" data structures swap roles each cycle. This means that we never call "new" once they are both allocated which is a great help to performance. So the strategy is arraycopy() for backup, and swap for undo().

Good Strategy

A good, simple strategy is to just backup all the columns when a place() or clearRows() takes us out of the committed state. This is a fine strategy.

More Complex Strategy

The more complex strategy for place() is to only backup the columns in the grid that the piece is in — a number of columns equal to the width of the piece (you do not need to implement the complex strategy; I'm just mentioning it for completeness). In this case, the board needs to store *which* columns were backed up, so it can swap the right ones if there's an undo() (two ints are sufficient to know which section of columns was backed up). With this strategy, if a clearRows() happens, the columns where the piece was placed have already been backed up, but now the columns to the left and right of the piece area need to be backed up as well.

Alternatives

You are free to try alternative undo strategies, as long as they are at least as fast as the good strategy above. The "articulated" alternative is to store what piece was played, and then for undo, go through the body of that piece and carefully undo the placement of the piece. It's more complex this way, and there's more logic code, but it's probably faster. For the row-clearing case, the brute force copy of everything is probably near optimal — too much logic would be required for the articulated undo of the deletion of the filled rows. The place()/undo() sequence is **much** more common in practice than other combinations like place()/clearRows()/undo(). Therefore, our official goal is to make place()/undo() as fast as possible, and just make sure all the other cases get the correct result.

While working through the commitment/undo code, your brain will naturally think of little puns such as "fear of commitment," "needing to be committed," etc. This is perfectly natural part of the coding process and is nothing to be ashamed of.

Performance

The Board has two design goals: (a) provide services for the convenience of the client, and (b) run fast. To be more explicit, here are the speed prioritizations...

1. Accessors: getRowWidth(), getColumnHeight(), getWidth(), getHeight(), getGrid(), dropHeight(), and getMaxHeight() — these should all run in constant time. They should just pull the answer out of a pre-computed ivar like maxHeight or heights. The place() and clearRows() methods should update the ivars efficiently when they change the board state.

2. The place()/clearRows()/undo() system can copy all the arrays for backup and swap pointers for undo(). That's almost as fast as you can get.

sanityCheck()

The Board has a lot of internal redundancy between the grid, the widths, the heights, and maxHeight. Write a sanityCheck() method that verifies the internal correctness of the board structures: iterate over the whole grid to verify that the widths and heights arrays have the right numbers, and that the maxHeight is correct. Throw an exception if the board is not sane: throw new RuntimeException("description"). Call sanityCheck() at the bottom of place(), clearRows() and undo(). A boolean static constant DEBUG in your board class should control sanityCheck(). If DEBUG is true, sanityCheck does its checks. Otherwise it just returns. Turn your project in with DEBUG=true. Put the sanityCheck() code in early. It will help you debug the rest of the board. There's one tricky case: do not call sanityCheck() in place() if the placement is bad -- the board may not be in a sane state, but it's allowed to be not-sane in that case.

The sanityCheck() is in addition to the unit tests below. The sanityCheck() has the advantage that it can check things **every time** place(), clearRows(), etc. are called, even while you are playing tetris.

Board Unit Test

Create a BoardTest JUnit test class. One simple strategy is to create a 3 x 6 board, and then place a few rotations of the pyramid in it. Call dropHeight() with a few different pieces and x values to see that it returns the right thing. Call place() one or two times, and then spot check a few qualities of the resulting board, checking the results of calls to getColumnHeight(), getRowWidth(), getMaxHeight(), getGrid(). Set up a board with two or more filled rows, call clearRows(), and then spot check the resulting board with the getters. Do a place()/clearRows() series, and then an undo() to see that the board gets back to the right state.

You are free to arrange the unit-test coverage as you like, so long as overall there are at least 10 calls each to getColumnHeight() and getRowWidth(), 5 calls to dropHeight() and getMaxHeight(), and at least 2 calls of everything else.

Your more advanced Board tests need to be more complex than just placing a single piece in a board. Stack up a few operations, checking a few board metrics (getGrid(), getColumnHeight(), …) at each stage, like this: Add a pyramid to a board. Add a second shape. Now row clear. Check post row clear that the state is right. Undo, and check the state again. With and without the undo, try adding a third piece to make sure the undo/rowClear operations haven't broken some part of the internal structure. When calling getColumnHeight() etc., you don't need to be comprehensive -- just check a couple columns or rows, and that will get the vast majority of bugs. Debugging the board "live" can be very difficult, so concentrating on a few hard board unit tests may be the easiest way to debug the whole thing. It is far easier to debug/step through the run of a unit test than use the debugger in a live game.

Note also that the provided board code includes a toString(), so you can println the Board state, which can be helpful to see a time-series of boards.

As with the piece unit-tests, we will try your unit tests on some broken board implementations, to see if your tests can sift the good from the bad.

JTetris

The provided JTetris class is a functional tetris player that uses your piece and board classes to do the work. Use the j, k, and l keys to play. You do not need to change the JTetris class at all. The "speed" slider adjusts how fast it goes. In the next section, you will create a subclass of JTetris that uses an AI brain to auto-play the pieces as they fall. For now, you can play tetris to check that your piece and board really do work right.

If while playing, you see a buggy behavior, for example a bug when row clearing the bottommost row in the board, try to add a unit test that exposes that case rather than trying to debug it live. Unit tests are more of an up-front cost, but they pay off.

One of the theories of unit tests, is that rather the invest effort in debugging a case -- the effort is used once and then forgotten -- you put that effort into making a unit test for each bug you work on. The unit test helps with that bug, and then it keeps helping for the rest of the lifetime of the code.

Milestone — Basic Tetris Playing

You need to get your Board and Piece debugged enough that using JTetris to play tetris works. Yay! Getting to this stage is most of the work of the assignment. Once your piece and board appear bug free, you can try the next step.

Part C -- Brain

For this part, you will build some neat features on top of the basic Piece and Board functionality.

Understand JTetris

Read through JTetris.java a couple times to get a sense of how it works. You will be writing a subclass of it, so you need to see how it works. Key points in JTetris...

main() creates a JTetris and the static createFrame() creates a frame housing the JTetris.

tick() is the bottleneck for moving the current piece

computeNewPosition() just encapsulates the switch logic to figure 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. The piece only lands for good when it cannot move down any more, and the player has stopped moving it around.

If the "Test sequence" checkbox is true when a game starts, the game uses a fixed sequence of 100 pieces and then stops. Seeing the same sequence of pieces every time can make debugging easier.

JBrainTetris

Create a JBrainTetris subclass of JTetris that uses an AI brain to auto-play the pieces as they fall. As usual for inheritance, your subclass should add new behavior but use the existing behavior in the superclass as much as possible. The provided DefaultBrain works fine, and you can work on your own if you wish.

This Is Your Brain

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

```
// Brain.java -- the interface for Tetris brains
```

```
public interface Brain {
    // Move is used as a struct to store a single Move
    // ("static" here means it does not have a pointer to an
    // enclosing Brain object, it's just in the Brain namespace.)
    public static class Move {
        public int x;
        public int y;
        public Piece piece;
        public double score;    // lower scores are better
    }

    /**
     Given a piece and a board, returns a move object that represents
     the best play for that piece, or returns null if no play is possible.
     The board should be in the committed state when this is called.

     limitHeight is the height of the lower part of the board that pieces
     must be inside when they land for the game to keep going
      -- typically 20.
     If the passed in move is non-null, it is used to hold the result
     (just to save the memory allocation).
    */
    public Brain.Move bestMove(Board board, Piece piece, int limitHeight,
                                       Brain.Move move);
}
```

DefaultBrain

The provided DefaultBrain class is a simple but entirely functional implementation of the Brain interface. Glance at DefaultBrain.java to see how simple it is. Given a piece, it tries playing the different rotations of that piece in all the columns where it will fit. For each play, it uses a simple rateBoard() method to decide how good the resulting board is — blocks are bad, holes are more bad. The methods dropHeight(), place(), and undo() are used by the brain to go through all the board combinations.

```
// DefaultBrain.java
/**
 A simple Brain implementation.
 bestMove() iterates through all the possible x values
 and rotations to play a particular piece (there are only
 around 10-30 ways to play a piece).

 For each play, it uses the rateBoard() message to rate how
 good the resulting board is and it just remembers the
 play with the lowest score. Undo() is used to back-out
 each play before trying the next. To experiment with writing your own
 brain -- just subclass off DefaultBrain and override rateBoard().
*/

public class DefaultBrain implements Brain {
    /**
     Given a piece and a board, returns a move object that represents
     the best play for that piece, or returns null if no play is possible.
     See the Brain interface for details.
    */
    public Brain.Move bestMove(Board board, Piece piece, int limitHeight,
                               Brain.Move move) {
        // Allocate a move object if necessary
        if (move==null) move = new Brain.Move();

        double bestScore = 1e20;
        int bestX = 0;
        int bestY = 0;
        Piece bestPiece = null;
        Piece current = piece;

        board.commit();
```

```
    // loop through all the rotations
    while (true) {
        final int yBound = limitHeight - current.getHeight()+1;
        final int xBound = board.getWidth() - current.getWidth()+1;

        // For current rotation, try all the possible columns
        for (int x = 0; x<xBound; x++) {
            int y = board.dropHeight(current, x);
            if (y<yBound) {    // piece does not stick up too far
                int result = board.place(current, x, y);
                if (result <= Board.PLACE_ROW_FILLED) {
                    if (result == Board.PLACE_ROW_FILLED) board.clearRows();

                    double score = rateBoard(board);

                    if (score<bestScore) {
                        bestScore = score;
                        bestX = x;
                        bestY = y;
                        bestPiece = current;
                    }
                }

                board.undo();    // back out that play, loop around for the next
            }
        }

        current = current.fastRotation();
        if (current == piece) break;    // break if back to original rotation
    }

    if (bestPiece == null) return(null);    // could not find a play at all!
    else {
        move.x=bestX;
        move.y=bestY;
        move.piece=bestPiece;
        move.score = bestScore;
        return(move);
    }
}


/*
 A simple brain function.
 Given a board, produce a number that rates
 that board position -- larger numbers for worse boards.
 This version just counts the height
 and the number of "holes" in the board.
*/
public double rateBoard(Board board) {
    final int width = board.getWidth();
    final int maxHeight = board.getMaxHeight();

    int sumHeight = 0;
    int holes = 0;
```

```
        // Count the holes, and sum up the heights
        for (int x=0; x<width; x++) {
            final int colHeight = board.getColumnHeight(x);
            sumHeight += colHeight;

            int y = colHeight - 2;    // addr of first possible hole

            while (y>=0) {
                if  (!board.getGrid(x,y)) {
                    holes++;
                }
                y--;
            }
        }

        double avgHeight = ((double)sumHeight)/width;

        // Add up the counts to make an overall score
        // The weights, 8, 40, etc., are just made up numbers that appear to work
        return (8*maxHeight + 40*avgHeight + 1.25*holes);
    }

}
```

Here's what your JBrainTetris needs to do...

- Building the JBrainTetris code relies on a basic understanding of the code up in JTetris -- that's the reality of complex inheritance.

- Add a main() which creates a frame containing a JBrainTetris rather than a JTetris.

- Override createControlPanel() to tack on a Brain label and the JCheckBox that controls if the brain is active. The checkbox should default to false. Just add the checkbox to the panel with code like this (also, take a look at the panel code in JTetris as a model)

  ```
  panel.add(new JLabel("Brain:"));
  brainMode = new JCheckBox("Brain active");
  panel.add(brainMode);
  ```

- A JBrainTetris object should own a single DefaultBrain object.

- The idea is that if the checkbox is checked, JBrainTetris will use the DefaultBrain object to auto play the piece as it falls. No listeners are required for the JCheckbox -- JCheckBox responds to an isSelected() message to see if it is checked at any time.

- The strategy is to override tick(), so that every time the system calls tick(DOWN) to move the piece down one, JBrainTetris takes the opportunity to move the piece a bit first. Our rule is that the brain may do up to one rotation and one left/right move each time tick(DOWN) is called: rotate the piece one rotation and move it left or right one position. With the brain on, the piece should drift down to its correct place. (optional) You can add an "Animate Falling" checkbox (default to true) to control how the brain works the piece once it is in the correct column but not yet landed. When animate is false, the brain can use the "DROP" command to drop the piece down into place. In any case, after the brain does its changes, the tick(DOWN) should have its usual effect of trying to lower the piece by one. So on each tick, the brain will move the piece a little, and the piece will drop down one row. The user should still be able to use the keyboard to move the piece around while the brain is playing, but the brain will move the piece back on course. As the board gets full, the brain may fail to get the piece over fast enough. That's ok. (We could have a mode where the brain just got to zap the piece into its correct position in one operation, but it's not as fun to watch.)

- JBrainTetris should detect when the JTetris.count variable has changed to know that a new piece is in play. Alternately, you could use overriding to detect the addition of each new piece, but the count variable works fine. At that point, it should use the brain to compute, **once**, where the brain says the piece should go -- the "goal". The brain needs the board in a committed state before doing its computation. Therefore, do a board.undo() before using the brain. You can see that this won't disrupt the JTetris logic, since JTetris.tick() does board.undo() itself. It's important for performance to compute the goal a single time for each new piece.

It can be sort of mesmerizing to watch the brain play; at least when it's playing well and the speed isn't too fast. Otherwise it can be kind of stressful.

Adversary

For this last step you will build what I think is the coolest example of modular code re-use that I have ever assigned.

- Modify JBrainTetris createControlPanel() to add a label that says "Adversary:", a slider with the range 0..100 and initial value 0, and a status label that says "ok".

- Override pickNextPiece(). It the slider is 0 (all the way left), the adversary should never intervene. If the slider is at 100, the adversary should always intervene. Create a random number between 1 and 99. If the random number is >= than the slider, then the piece should be chosen randomly as usual (just "super" on up). But if the random value is less, the mischief begins. In that case the "adversary" gets to pick the next piece. When the piece is chosen at random, setText() the status to "ok", otherwise set it to "*ok*". (We don't want the feedback to be too obvious so the roommate test below can work nicely.)

```
// make a little panel, put a JSlider in it. JSlider responds to getValue()
little = new JPanel();
little.add(new JLabel("Adversary:"));
adversary = new JSlider(0, 100, 0);    // min, max, current
adversary.setPreferredSize(new Dimension(100,15));
little.add(adversary);
// now add little to panel of controls
```

- The adversary can be implemented with a little JBrainTetris code that uses the brain to do the work. Loop through the pieces array. For each piece, ask the brain what it thinks the best move is. Remember the piece that yielded the move with the worst (largest) 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! "Oooh tough break, another S. Gosh that's too bad. I'm sure the long skinny one will be along real soon."

- I just love the abstraction here. The Brain interface looks so reasonable. Little does the brain realize the bizarre context where it will be used — just the way modular code is supposed to work. Also notice how vital the speed of Board is. There are about 25 rotations for each piece on a board, so the adversary needs to be able to evaluate 7*25=175 boards in the tiny pause after a piece has landed and the next piece is chosen "at random". That's why the place()/undo() system has to be fast. Row clearing will be rare in all those cases, but we need to be able race through the placements.

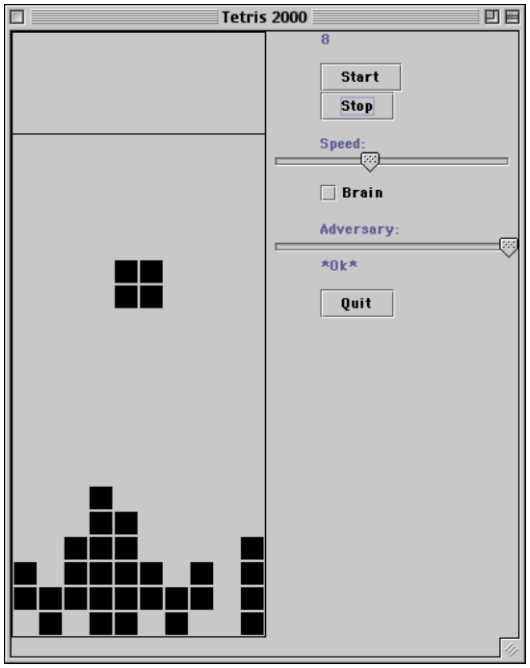
- It's absolutely vital that once you have the adversary working, you go test it on your roommate or other innocent person. Leave the adversary set to around 40% or so, and leave the speed nice and slow. "Hey Bob, I understand you're pretty good at Tetris. Could you test this for me? It's still pretty slow, so I'm sure you'll have no problem with it."

- For ironic enjoyment, have the brain play the adversary.

Deliverables

- Your piece and board should work correctly and should have unit tests.
- Your board should have the correct internal structure -- efficient place(), rowWidth(), undo() etc. and a functioning sanityCheck().
- We should be able to play tetris using the keyboard in the usual way, running either the JTetris main() or JBrainTetris main().
- We should be able to use the brain and the adversary in JBrainTetris.

Ahhhh -- good old adversary -- always able to find the perfect piece for an occasion...



Appendix A -- AI Ideas

There's not time to think about this now, but someday it could be fun to play around with making a better tetris brain. Putting together a better tetris brain is a fascinating algorithmic/AI problem. If you want to try, create your own subclass of DefaultBrain and use the Load Brain button to load get JTetris to use it. Two things that the default strategy does not get right are: (a) avoiding creating great tall troughs so only the long skinny one will fit, and (b) accounting for things near the top edge as being more important things that are deeply buried. There's also a problem in tuning the weights. If this were your thesis or something, you would want to write a separate program to work on optimizing the weights -- that can make a big difference. Here's the GUI code for a Load Brain button...

```
JButton button = new JButton("Load brain");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            Class bClass = Class.forName(brainText.getText());
            Brain newBrain = (Brain) bClass.newInstance();
            // here change Brain ivar to use newBrain
            status.setText(brainText.getText() + " loaded");
        }
        catch (Exception ex) {
```

```
            ex.printStackTrace();
        }
    }
});
```