

219113 Object-Oriented Programming II

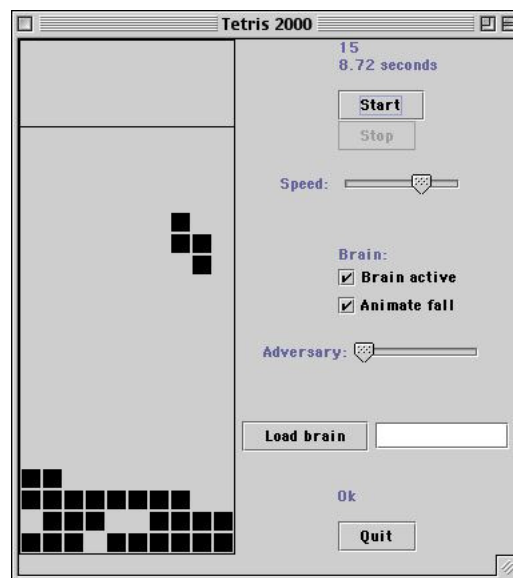
Tetris Project

Due: about the beginning of the first semester in your sophomore year

Acknowledgement: this assignment is taken from Nick Parlante's 2001 Nifty Assignment session, which has been used as one of the projects in Stanford's CS108 class.


Instructions: This project is to be done individually. Once you are done with both part A and part B, schedule an appointment with the instructor for a one-on-one interview. The interview must take place on or before the due date.


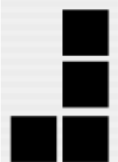
For this project, you will build up a set of classes for Tetris. This assignment emphasizes the basic Divide and Conquer strength of OOP design -- using encapsulation to divide a big scary problem into several slightly less scary and independently testable problems. Tetris is a large project, so the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






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 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 "==" test and an "(object instanceof Piece)" test for the passed in object. The starter code has this standard code already included.

Note: Piece does not include an implementation of hashCode. This means that it will inherit hashCode from object. We do not recommend using Piece in a manner in which using a hashCode would be necessary. However, if you decide to use a Piece as a hashmap key or in other situations in which a hashCode would be necessary, you'll need to write your own hashCode implementation.

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 project deliverable, and it is important that you do a good job on them

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.

Do not proceed to part B unless you have thoroughly completed your work in part A.

Part B -- Board

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

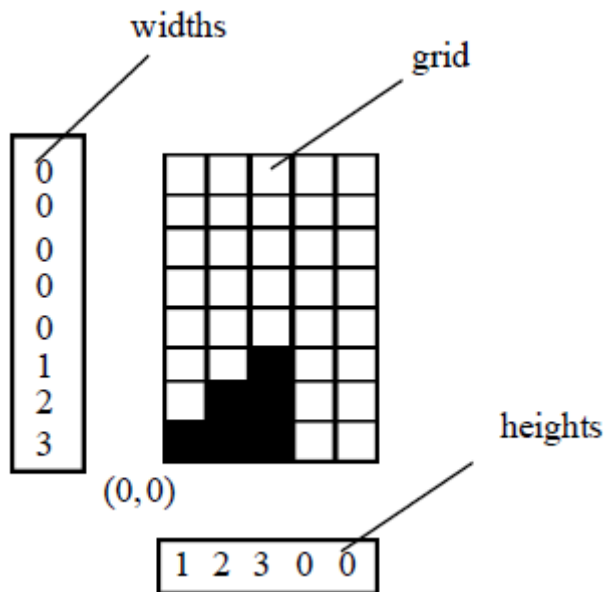
- Store the current state of a tetris board.
- Provide support for the common operations that a client "player" module needs to build a GUI version of the game: add pieces to the board, let pieces gradually fall downward, detect various conditions about the board. The player code is provided for you,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



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 at 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 which we are implementing 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 $O(\text{piece_width})$ 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.

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.