CS108, Stanford
Winter, 2000-01

Handout #12
Nick Parlante

HW2 Tetris

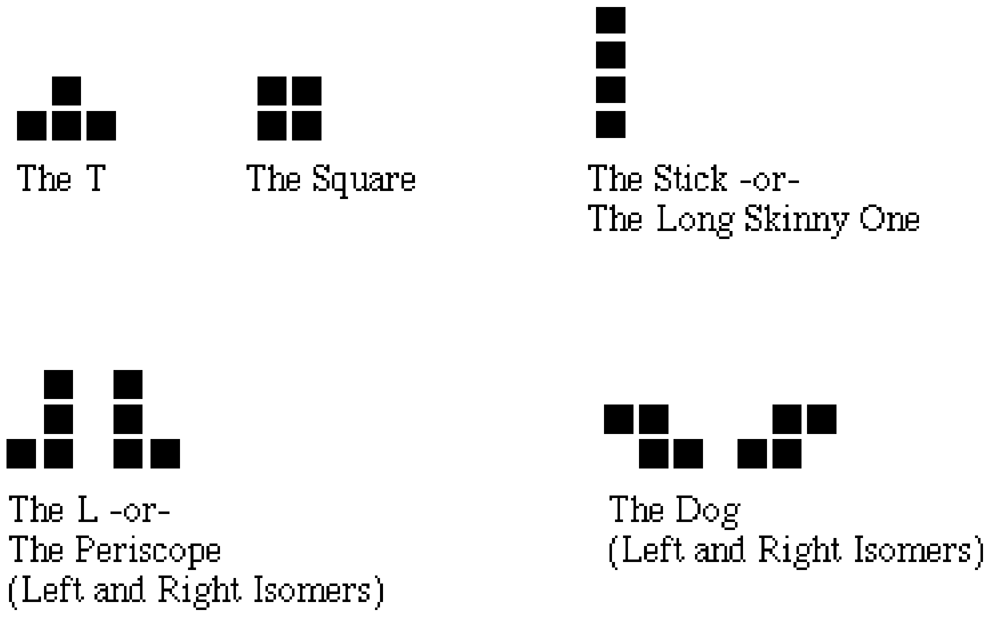
(This is the handout we use for the Tetris assignment for our 2nd year CS undergraduates. It explains how to solve Piece and Board.)

For HW2 you will build up a set of classes for Tetris. This assignment will emphasize elemental OOP design -- using encapsulation to divide a big scary problem into many friendly little independently testable problems. The first part of the assignment sets up the Piece class. The second part builds the Board class and some other fun bits. The whole thing is due Thu Feb 1st. 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/explorations/2000/101600tetris/).

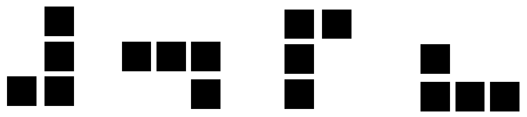
Piece

There are seven pieces in standard Tetris.



Each standard piece is composed of four blocks. The two "L" and "dog" 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" (Not that I actually know that word -- I looked it up to make the handout more impressive.).

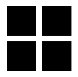
A piece can be rotated 90˚ counter-clockwise to yield another piece. Enough rotations get you back to the original piece — for example rotating a dog twice brings you back to the original state. Essentially, each tetris piece belongs to a family of between one and four distinct rotations. The square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

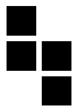
Body

A piece is represented by the coordinates of its blocks which are known as the "body" of the piece. Each Piece has its own little 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 the origin of the piece. So, the four points of the square piece are then:



```
(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 actually have a block at `(0,0)`. For example, the body of the following rotation of the right dog



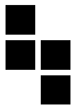
has the body:

```
[(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. The above right dog was a width of 2 and height of 3. Another quality which turns out to be useful for playing Tetris quickly is the “skirt” of a piece....

Skirt

The skirt will be an int[] array, as long as the piece is wide, that stores the lowest y value for each x value in the piece coordinate system.



The skirt of this piece is {1, 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 piece for that x.

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 pointer to the "next rotation" which yields a pointer a piece object that represents the next rotation. This is the "immutable" paradigm -- there is not a rotate() message that changes the receiver. Instead, the piece objects are read-only, and the client can iterate over them (the String class is another example of the immutable paradigm).

For efficiency, we will pre-compute all the rotations just once. Given a piece object, the client will be able to get a pointer to the "next" piece object which represents the next rotation. In essence, this allows the client to obtain each rotation in constant time.

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 so your Piece will fit in with the later components. You will want to add your own private helper methods which can have whatever prototypes you like.

```
// Piece.java

import java.awt.*;
import java.util.*;

/**
 An immutable representation of a tetris piece in a particular rotation.
 Each piece is defined by the blocks that make up its body.
 See the Tetris-Overview.html for more information.

 This is the starter file version -- a few simple things are filled in already

 @author Nick Parlante
 @version   1.0, Mar 1, 2001
*/
```

```
public final class Piece {
/*
 Implementation notes:
 -The starter code does out a few simple things for you
 -Store the body as a Point[] array
 -The ivars in the Point class are .x and .y
 -Do not assume there are 4 points in the body -- use array.length
 to keep the code general
*/
   private Point[] body;
   private int[] skirt;
   private int width;
   private int height;
   private Piece next;  // "next" rotation

   static private Piece[] pieces;   // singleton array of first rotations


   /**
    Defines a new piece given the Points that make up its body.
    Makes its own copy of the array and the Point inside it.
    Does not set up the rotations.

    This constructor is PRIVATE -- if a client
    wants a piece object, they must use Piece.getPieces().
   */
   private Piece(Point[] points) {
   }

    /**
     Returns the width of the piece measured in blocks.
    */
   public int getWidth() {
      return(width);
   }

    /**
     Returns the height of the piece measured in blocks.
    */
   public int getHeight() {
      return(height);
   }

    /**
     Returns a pointer to the piece's body. The caller
     should not modify this array.
    */
   public Point[] getBody() {
      return(body);
   }

    /**
     Returns a pointer to the piece's skirt. For each x value
     across the piece, the skirt gives the lowest y value in the body.
     This useful for computing where the piece will land.
     The caller should not modify this array.
    */
   public int[] getSkirt() {
```

```
   return(skirt);
}


/**
 Returns a piece that is 90 degrees counter-clockwise
 rotated from the receiver.

 <p>Implementation:
 The Piece class pre-computes all the rotations once.
 This method just hops from one pre-computed rotation
 to the next in constant time.
*/
public Piece nextRotation() {
   return next;
}


/**
 Returns true if two pieces are the same --
 their bodies contain the same points.
 Interestingly, this is not the same as having exactly the
 same body arrays, since the points may not be
 in the same order in the bodies. Used internally to detect
 if two rotations are effectively the same.
*/
public boolean equals(Piece other) {
}




/**
 Returns an array containing the first rotation of
 each of the 7 standard tetris pieces.
 The next (counterclockwise) rotation can be obtained
 from each piece with the {@link #nextRotation()} message.
 In this way, the client can iterate through all the rotations
 until eventually getting back to the first rotation.
*/
public static Piece[] getPieces() {
/*
 Hint

 My code to produce the array of the pieces looks like the following.
 -parsePoints computes the Point[] array
 -The Piece constructor builds a single piece but not the rotations
 -The helper function piecerRow() computes all the rotations of that piece
 and connects them by their .next fields.
```

```
    Your pieces must be in the same 0..6 order to get the same output as
    the sample solution. Only compute the array when it is first asked for.
    Then just re-use that array for later requests.

      pieces = new Piece[] {
         pieceRow(new Piece(parsePoints("0 0 0 1   0 2      0 3"))),    // 0
         pieceRow(new Piece(parsePoints("0 0 0 1   0 2      1 0"))),    // 1
         pieceRow(new Piece(parsePoints("0 0 1 0   1 1      1 2"))),    // 2
         pieceRow(new Piece(parsePoints("0 0 1 0   1 1      2 1"))),    // 3
         pieceRow(new Piece(parsePoints("0 1 1 1   1 0      2 0"))),    // 4
         pieceRow(new Piece(parsePoints("0 0 0 1   1 0      1 1"))),    // 5
         pieceRow(new Piece(parsePoints("0 0 1 0   1 1      2 0"))),    // 6
      };
   */
   }


   /**
    Given a string of x,y pairs ("0 0  0 1   0 2   1 0"), parses
    the points into a Point[] array.
    (Provided code)
   */
   private static Point[] parsePoints(String string) {
       // could use Arraylist here, but use vector so works on Java 1.1
      Vector points = new Vector();
      StringTokenizer tok = new StringTokenizer(string);
      try {
         while(tok.hasMoreTokens()) {
            int x = Integer.parseInt(tok.nextToken());
            int y = Integer.parseInt(tok.nextToken());

            points.addElement(new Point(x, y));
         }
      }
      catch (NumberFormatException e) {
         throw new RuntimeException("Could not parse x,y string:" + string);  //
cheap way to do assert
      }

      // Make an array out of the Vector
      Point[] array = new Point[points.size()];
      points.copyInto(array);
      return(array);
   }

}
```

Rotation Strategy

The overall piece rotation strategy uses a single, static array with the "first" rotation for each of the 7 pieces. Each of the first pieces is the first node in a little circular linked list of the rotations of that piece. The client uses nextRotation() to iterate through all the rotations of a tetris 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.

Rotation Tactics

You will need to figure out an algorithm to do the actual rotation. Get a nice sharp pencil. Draw a piece and its rotation. Write out the co-ordinates of both bodies. Think about the transform that converts from a body to the rotated body. The transform uses reflections (flipping over) around various axes.

Equals

For the .equals() test, use the (object **instanceof** Piece) boolean test to see what sort of object you have before trying to cast it. See String.java for an example. equals() implementation. It's a standard trick to optimize the (this==other) (pointer comparison) case to return true without looking at the bodies.

Private Helpers

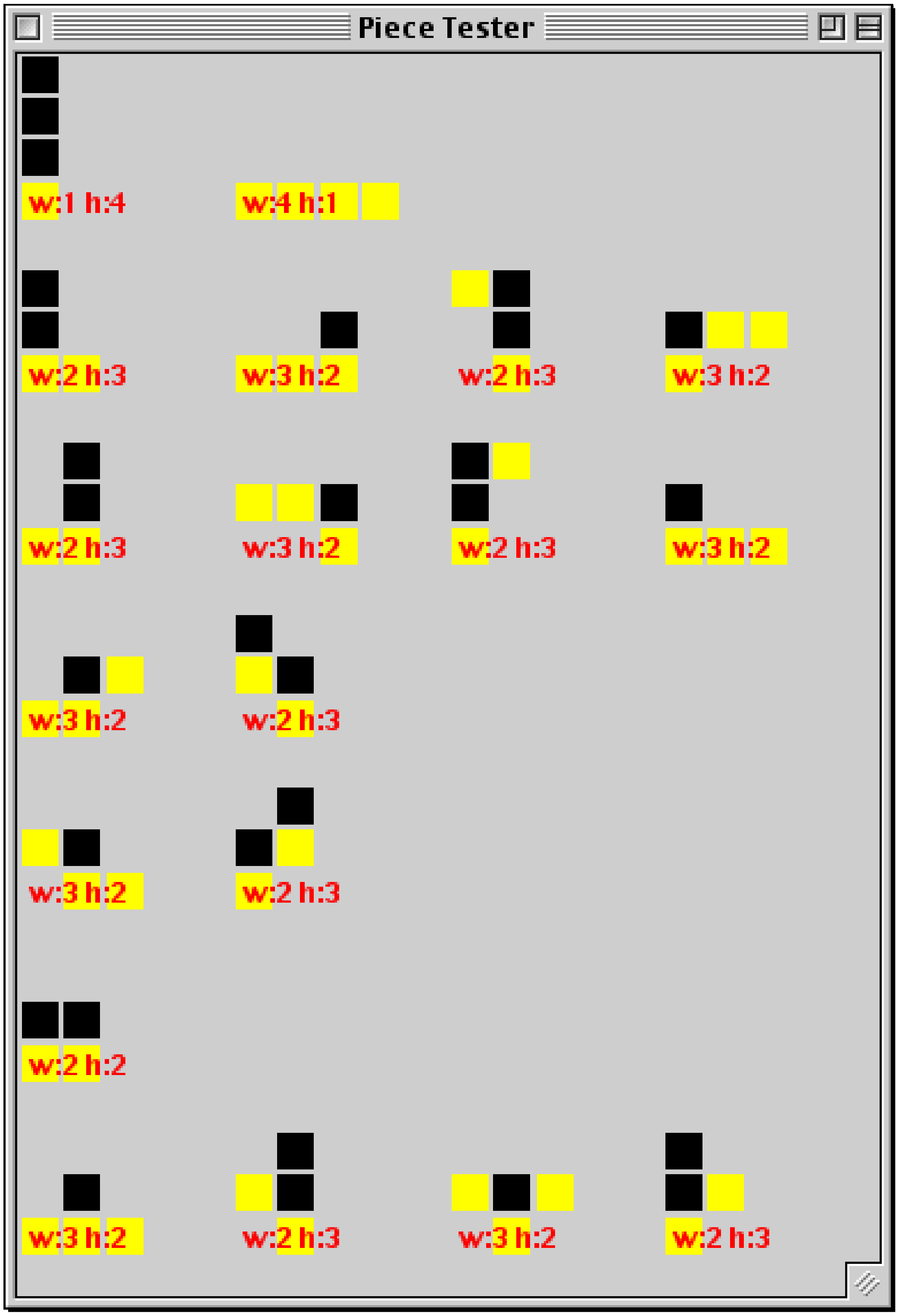
You will want private methods behind the scenes to compute the rotation of piece and assemble all the rotations into a list. Also notice that the computation of the width/height/skirt happens when making new pieces and when computation rotations -- don't have two copies of that code.

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.

JPieceTest

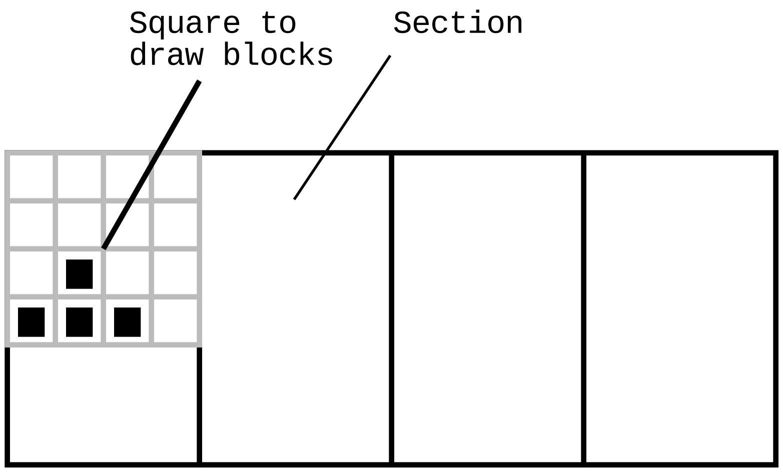
The last part of the Piece class is a test main that draws all the pieces in a window which looks like...



JPieceTest

Each row in the window is an instance of JPieceTest. Each JPieceTest component takes a single piece, and draws all its rotations in a row. The code to set up the 7 JPieceComponents in a window is provided for you.

Here's how the JPieceTest should draw...



- Divide the component into 4 sections. Draw each rotation in its own section from left to right, stopping when getting to the end of the distinct rotations.

- Allow space for a square 4 blocks x 4 blocks at the upper-left of each section. Draw the blocks of the piece starting at the bottom of that square. The square won't fit the section exactly, since the section may be rectangular.

- When drawing the blocks in the piece, leave a one-pixel border not filled in around each block, so there's a little space around each block. Each block should be drawn as a black square, except the blocks which are the skirt if of the piece -- they 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 the width and height of the block with a string like "w:3 h:2". The string should be red.

The JPieceTest code should be written as a plain client of Piece -- just use the same methods available to other clients. See the JPieceTest.java starter file for more information. The JPieceTest class represents the "debugging through transparency" strategy -- make the state of the objects apparent. Once your Piece is working and drawing itself, it'll be time to get started with the tetris board.

PieceTest Milestone

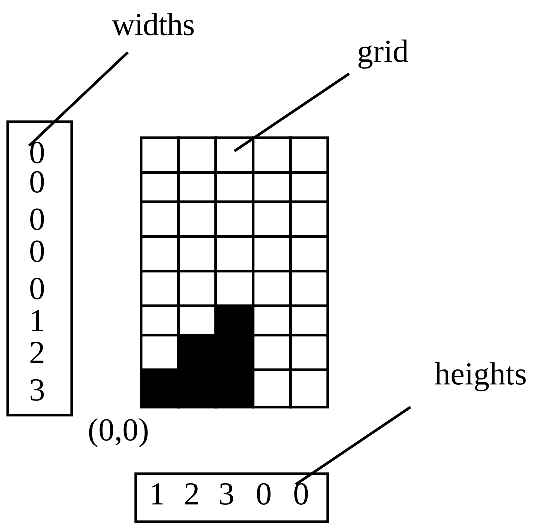
You should be able to load and compute all the piece rotations and see that it's working correctly through JPieceTest.

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 place() operation to detect efficiently if the placement has caused a row to become filled. 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.

int place(piece, x, y)

This takes a piece, and an (x,y), and sets the piece into the grid with the origin (the lower-left corner) of the piece at that location in the board. The undo() operation (below) can remove the most recently placed piece.

Returns PLACE_OK for a successful placement. Returns 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 spots in the grid that are already filled. Such bad placements may leave the board in a partially invalid state -- the piece has been partly but no completely added for example. If part of the piece would fall out of bounds return PLACE_OUT_BOUNDS . Otherwise, if the piece overlaps already filled spots, return PLACE_BAD . The client may return the board to its valid, pre-placement state with a single undo().

clearRows()

Delete each row that is filled all the way across, causing things above to shift down. 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. Use JBoardTest (below) to generate a few of the weird row-clearing cases.

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 skips over filled rows on its way up. The contents of the widths array needs to be shifted down also.

By knowing the maximum filled height of all 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 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.

int dropHeight(piece, x)

DropHeight() computes the y value where the origin (0,0) of a piece will come to rest if dropped in the given column from infinitely high. Drop height should use the heights array and the skirt of the piece to compute the y value quickly -- O(piece-width) time. DropHeight() assumes the piece falls straight down -- it does not account for moving the piece around things during the drop.

Part 2 -- undo() Abstraction

The problem is that the client code doesn't want to just add a sequence of pieces, the client code wants to experiment with adding different pieces. To support this client use, 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 -- OOP encapsulation design in a nutshell.

undo()

The board has a "committed" state which 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.

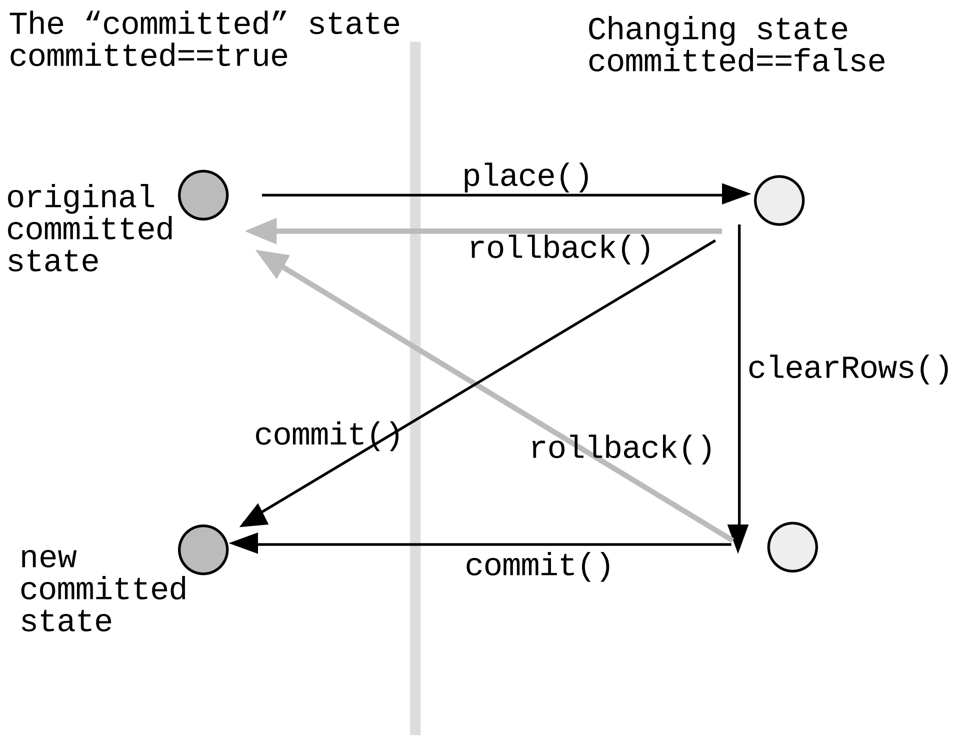
- The client may do a single clearRows() operation, which also sets committed==false

- 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().

There is some debate among the staff as to whether the following little state change diagram helps, but here it is anyway...



Basically, the board gives the client the ability to do a single place, a single 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 don't do anything. It can be convenient for the client code to commit() 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. place() and clearRows() operations copy the main state to the backup before making changes. undo() restores the main state from the backup.

1. Widths and Heights

For the int[] widths and heights arrays, the board has backup arrays called xWidths and xHeights. On place(), copy the current int contents of the two arrays to the backups. Use System.arraycopy(source, 0, dest, 0, length). System.arraycopy() is pretty optimized as Java runtime operations go.

Swap trick

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...

```
// perform undo...

System.arraycopy(xWidths, 0, widths, 0, widths.length);     // NO


int[] temp = widths; // YES just swap the pointers
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().

2. Grid

The grid needs to be backed up for a place() operation...

Simple

The simplest strategy is to just backup all the columns when place() happens. This is an acceptable strategy. In this case, no further backup is required for clearRows(), since place() already backed up the whole grid.

Less Simple

The more complex (faster) strategy is to only backup the columns 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 run of columns was backed up).

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.

As with the widths and heights, a copy-on-backup, the swap-pointers-on-undo strategy works nicely.

Alternatives

You are free to try alternative undo strategies, as long as they are at least as fast as the simple 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 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 than place() / clearRows() / undo(), so concentrating on making place() / undo() fast is a good strategy.

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.

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: 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 called DEBUG in your board class should control sanityCheck(). If debug is on, 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 in that case.

Performance

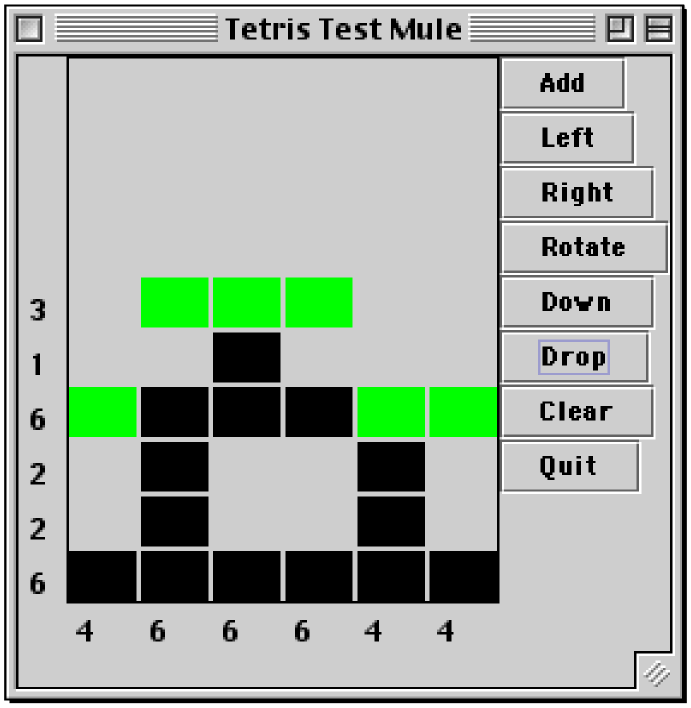
The Board has two design goals: (a) provide services for the convenience of the client, and (b) run fast. To be more explicit, here are the speed prioritizations...

1. Accessors: getRowWidth(), getColumnHeight(), getWidth(), getHeight(), getGrid(), dropHeight(), and getMaxHeight() — these should all be super fast. Essentially constant time.

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.

Milestone — JBoardTest (The Mule)

Once you have Board somewhat written, you can use the provided JBoardTest class ("the mule") for testing. The mule is just a GUI driver for the board interface. It lets you add pieces, move them around, place, and clear rows. In the spirit of "transparency" 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 than trying to observe your board while playing tetris in real time. You can also write your own debugging helper code such as a printBoard() method that prints out all of the state of the board to standard output -- sometimes it's handy to have a log you can scroll back through to see all the state over time. The "drop" button in our applications will only work if there's a straight-down vertical path for the piece to fall from an infinite height. If there's an overhang in the way above the piece, drop will not do anything (look at the JBoardTest source code for "drop").



Here I've set up a case to test clearing multiple rows.

You can look at the JBoardTest sources to see how it's implemented. It's a fairly thin layer built on the board. Once your board appears to be working correctly in the mule, it's time for the next step...

JTetris

The provided JTetris class is a functional tetris player that uses your piece and board classes to do the work. Use the keys "4 5 6" to position and "0" to drop (j k l, n -- are alternatives). The "speed" slider adjusts how fast it goes. You will create a subclass of JTetris that uses an AI brain to auto-play the pieces as they fall. Finally, you'll add the much needed adversary feature.

Milestone — Basic Tetris Playing

You need to get your Board and Piece debugged enough that using JTetris to play tetris works. If it's too fast to see what's going on, go back to the mule. Once your piece and board appear bug free, you can try the next step.

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...

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

If the command line argument "test" is present, the boolean testMode is set to true. In that case, the game plays the same sequnce of pieces every time, which can help with debugging.

As usual for inheritance, your subclass should add new behavior but use the existing behavior in the superclass as much as possible.

JBrainTetris

You first job will be to write a JBrainTetris subclass of JTetris that uses an AI brain to auto-play the pieces as they fall. We provide a simple brain for you, but 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 although it could have been defined as a class.

```
// 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 bottom section of the board that where pieces must
     come to rest -- 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);
}
```

LameBrain

The provided LameBrain class is a simple implementation of the Brain interface. Glance at LameBrain.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 bad. Board.dropHeight(), place(), and undo() are used by the brain to go through all the board combinations.

(Brain work for sometime in the future) 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 (the Load Brain code is at the end of the handout). 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.

You can copy ideas from JTetris source to build your JBrainTetris. Here's what your JBrainTetris needs to do...

- Change JTetris.main() so it creates an instance of your JBrainTetris instead of JTetris.

- JBrainTetris should own a single instance of LameBrain.

- The idea is that if the checkbox is checked, JBrainTetris will use the DefaultBrain to auto play the piece as it falls.

- 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. The brain may do up to one rotation and one left/right move each time tick() is called: rotate the piece one rotation and move it left or right one position. The piece should drift down to its correct place. (optional) You can add the "Animate Falling" checkbox to control if the brain is willing to use the "DROP" command to drop the piece down once it is lined up. After the brain moves, the tick(DOWN) can go through as usual. 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.)

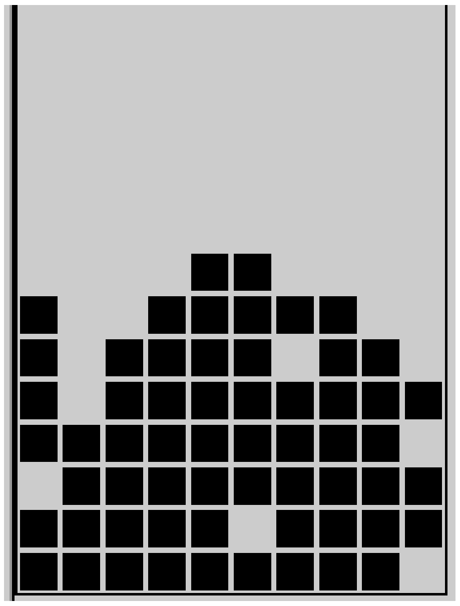
- Override createControlPanel() to tack on a Brain label and the JCheckBox that controls if the brain is active. The checkbox should default to false, unless testMode is true from the command line "test" switch.

- JBrainTetris should detect when the JTetris.count variable has changed to know that a new piece is in play. At that point, it should use the brain to compute, once, where the brain says the piece should go -- the "goal". It's important for performance to compute the goal a single time for each new piece. (Alternately, you could use overriding to detect the addition of each new piece, but the count variable works fine.)

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 sort of stressful to watch.

Stress Test

Use test mode (use the arg "test" on the command line) to force JTetris to use the fixed sequence of 00 pieces. Test mode with the unchanged DefaultBrain and with the pieces array built in the standard order should lead to exactly the following board after the 100 pieces have been played...



This is an extremely rigorous test of your Board. Although there were only 100 different boards with a piece landed on screen, the brain explored thousands of boards that were never on screen. If getColumnHeight() or clearRows() or undo() were wrong once among the thousands of boards, it could change the course of the whole thing. If you can't get the stress test working, don't worry about it too much; it's not the only thing we grade on. On the other hand, if your Board passes the stress test, it's probably perfect.

If the stress test is not coming out right, you could...

- Look at the JPieceTest output to verify that the pieces are correct in every detail.

- Put in additional sanityCheck() style code: check that clearRows changes the number of blocks by the right number (should be multiple of the board width), check that undo() is really restoring exactly the old state.

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 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(). Create a random number between 1 and 99. If the random number is greater 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.) It the slider is 0 (all the way left), the adversary should never intervene.

- 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 dog. 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 so fast. Row clearing will be rare in all that, 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

- We should be able to run JPieceTest and it should look right.

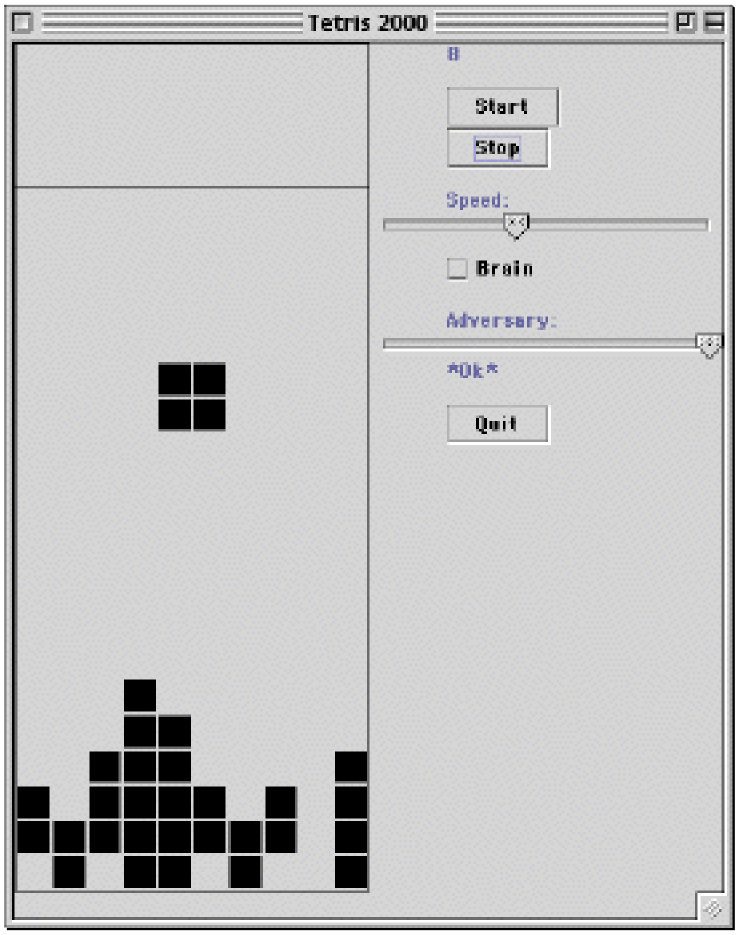
- Your board should have the correct internal structure -- efficient place(), rowWidth(), undo() etc. and a functioning sanityCheck().

- We should be able to play tetris using the keyboard in the usual way and turn the brain on and off.

- We should be able to run your app with "java JTetris test" and so do the stress test. Use the provided LAmeBrain unchanged so the stress test has a chance to come out right.

- We should be able to use the slider to activate the adversary "feature".

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



Appendix -- Load Brain

Here's the GUI code to have your own Load Brain button...

```
brainText = new JTextField();
brainText.setPreferredSize(new Dimension(100,20));

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