Omino – Process

- what do we add on to? what is provided? what do we add on to? what do we have to create from scratch?
  - write classes `Block`, `Polyomino`, `Piece`, `Board` from scratch
  - write most of `Game` from scratch
  - add on to `Omino`
  - write most of `OminoTester` from scratch for testing
  - specifically what to do is covered in the handout
- do we have to follow the instructions exactly?
  - yes – you need to have the specific classes, instance variables, and methods described, with exactly the names specified
    - if names are not specified, use descriptive names and follow standard conventions
    - conventions to follow: see “Coding Standards” on the main course webpage

Omino – Process

- this is big and complicated…
  - the handout is long because it breaks down the task into smaller chunks and gives you a plan of attack – follow that!
    - for each class, tackle the elements in the order listed – declare instance variables, write the constructor(s), write each method, write (and run) test cases
  - tackle each chunk (class, method) separately
    - when working on one chunk, don't worry about how that chunk is used by the rest of the program – focus on that chunk's task as described in the handout
      - the contract defines that interface between the chunk and the rest of the program – it's what the rest of the program expects of the chunk, so the chunk just has to live up to it
  - start early!

Omino – Working With Classes and Objects

- write the class `Block`

`Block`

A polyomino is made up of a bunch of squares (or *blocks*). `Block` encapsulates the position (row and column) of a polyomino block.

Instance variables:

- the block's row
- the block's column

Constructor:

- takes a row and column as parameters and initializes the instance variables accordingly

Methods:

- a getter for the row
- a getter for the column

Testing:

`Block` just holds and provides access to row and column values, and its method bodies are short and straightforward. There's not really any value in writing a tester for `Block`.

Omino – Working With Classes and Objects

- start with the class header

`Block`

A polyomino is made up of a bunch of squares (or *blocks*). `Block` encapsulates the position (row and column) of a polyomino block.

```
/**
 * A block of a polyomino.
 *
 * @author Stina Bridgeman
 */
public class Block {
}
```

Omino – Working With Classes and Objects

- declare instance variables
- write the constructor(s)

Instance variables:

- the block's row
- the block's column

Constructor:

- takes a row and column as parameters and initializes the instance variables accordingly

```
public class Block {

  private int row_, col_; // position of the block

  /**
   * Create a block with the specified position.
   *
   * @param row
   *          row
   * @param col
   *          column
   */
  public Block ( int row, int col ) {
    super();
    row_ = row;
    col_ = col;
  }
```

note naming convention – end instance variable names with _

class invariants?
preconditions?

not in this case – while we won't create blocks with negative values for row or column, these values are relative positions and so negative values are valid

Omino – Working With Classes and Objects

- write methods, one by one

Methods:

- a getter for the row
- a getter for the column

```
/**
 * Get the block's row.
 *
 * @return row
 */
public int getRow () {
  return row_;
}

/**
 * Get the block's column.
 *
 * @return column
 */
public int getCol () {
  return col_;
}
```

identify and check preconditions

no parameters, so none to worry about here

check class invariants

none to check for this class, also getters don't change anything so it's not useful to check

Omino – Working With Classes and Objects

- implement test cases

Testing:

`Block` just holds and provides access to row and column values, and its method bodies are short and straightforward. There's not really any value in writing a tester for `Block`.

- tester subroutine to run a test case for each method being tested
- test cases in `main`
- similar to testers from labs 2 and 4

Omino – Working With Classes and Objects

- how do the various classes relate to each other? when do I make use of their instance variables and methods?
  - reference to a class as a type
    - recognize when the kind of thing matches with a class
  - call a method to manipulate an object
    - recognize when a task needs or manipulates information belonging to another object

Instance variables:

- the board
- the polyominoes (as an array of `Polyomino`)
- the current score
- the number of pieces played
- the number of rows cleared
- the current piece
- the current position (row and column) of the current piece
- whether or not the game is in progress (started but not yet over)
- whether or not the game is over (a piece extends past the top of the board)

```
private Board board_;
```

"if it is legal to place the new piece in the new position" →

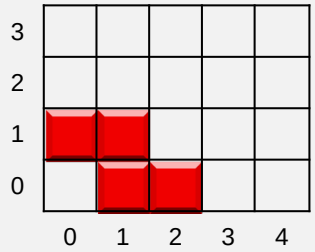
```
if ( board_.canPlace(newpiece,
                     newrow,newcol) ) { … }
```

In particular, `movePiece` should:

- determine the new piece and position resulting from the action (`LEFT`, `RIGHT`, `DOWN`, `DROP` just result in a new position; `ROTATE` results in a new piece as well as a new position) - but don't update the current piece or position yet
- if it is legal to place the (new) piece in the new position, update the current piece and position to the new information
- if the piece has landed (meaning the action was `DROP` or the action is `DOWN` and the piece couldn't be placed in its new position) --
  - score points for landing the piece
  - update the game over status if the landed piece sticks out above the top of the board (game is over)
  - add the piece to the board
  - clear any filled rows and update the score accordingly
  - if the game isn't over, start a new piece at the top of the board

Omino – Blocks and Polyominoes

- what are the rows and columns for blocks?
- how does the coordinate system work for polyomino blocks?
- how does a string represent the blocks of a polyomino?



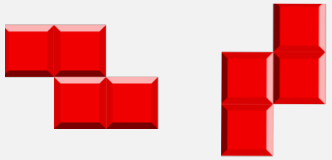
imagine a grid superimposed on the polyomino, fit as tightly as possible to the left and bottom edges of the shape
- there may not be a block at (0,0) but there will be at least one block in row 0 and in column 0

the blocks are (1,0), (0,1), (1,1), (0,2)
- block (1,0) is in row 1, column 0

written as a string: `1 0  0 1  1 1  0 2`

Omino – Pieces and Rotating

- how does rotation work? what is the "next rotation"?
- what does `getNumRotations()` in `Polyomino` mean?



orientation: 0 orientation: 1

number of rotations is 2, because there are two different possible orientations for this polyomino

`Polyomino` constructor is given an array of strings containing the possible orientations:
```
{ "1 0  0 1  1 1  0 2",
  "0 0  1 0  1 1  2 1" }
```

index refers to which slot in the array of possible orientations

the next rotation is the next one in the sequence:
- orientation 0 → 1, orientation 1 → 0

- why does rotation result in a new piece instead of changing the positions of the blocks in the current piece?
  - the piece should look like it is being rotated around its center, which may require changing its position on the board

Omino – 2D Arrays

- how do 2D arrays work?
  - section 3.8.5 in the textbook
  - a 1D array is like a row of boxes, with which box indexed by an integer starting at 0
    ```
    int[] arr1 = new int[10];
    arr1[3] = 20;
    ```
    - loop to go through each slot
  - a 2D array is like a grid of boxes, with row and column each indexed by an integer starting at 0
    ```
    int[][] arr2 = new int[10][5];
    arr2[3][1] = 20;
    ```
    - nested loops to go through each slot

2D Arrays

For the polyomino definition array, keep in mind that a 2D array is really a 1D array where each slot holds a 1D array - and that you don't have to have the same number of elements in each row. For example, the following defines the two one-sided trominoes (three squares) using the initializer list syntax:

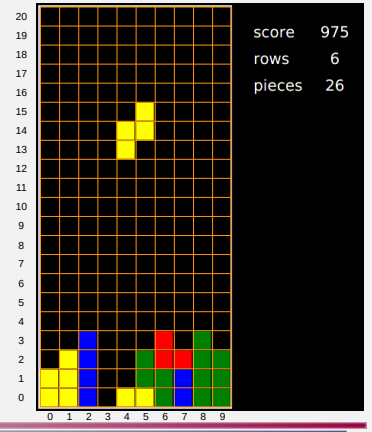
```
private static final String[][] POLYOMINOES =
  { { "0 0  1 0  2 0", "0 0  0 1  0 2" },
    { "0 0  1 0  0 1", "0 0  0 1  1 1", "1 0  0 1  1 1", "0 0  1 0  1 1" } };
```

Omino – 2D Arrays

- how to get blocks into the coordinate system of the board?
  - draw pictures (label what you know) to help you figure it out!

current piece position (13,4)
– location in board coordinates of where the piece's (0,0) block is

block (2,1) in the piece is at what position on the board?



Omino – 2D Arrays

- how do you clear rows?
  - update the contents of the array
  - draw before and after pictures to help you figure it out!

which cells of the board need to be updated?
- go through each of them

what's the new value for each?

Omino – Scoring

- how do you reward players for clearing multiple rows at once?
  - `Board`'s `clearRows` method does the clearing and returns the number of rows cleared

- `clearRows`, which removes rows that are filled all the way across, moving higher rows down, and returns the number of rows cleared (0 if no rows were cleared)

  - `Game`'s `movePiece` method calls `clearRows`

In particular, `movePiece` should:

- determine the new piece and position resulting from the action (`LEFT`, `RIGHT`, `DOWN`, `DROP` just result in a new position; `ROTATE` results in a new piece as well as a new position) - but don't update the current piece or position yet
- if it is legal to place the (new) piece in the new position, update the current piece and position to the new information
- if the piece has landed (meaning the action was `DROP` or the action is `DOWN` and the piece couldn't be placed in its new position) --
  - score points for landing the piece
  - update the game over status if the landed piece sticks out above the top of the board (game is over)
  - add the piece to the board
  - clear any filled rows and update the score accordingly
  - if the game isn't over, start a new piece at the top of the board

  - `Game` has an array which relates the number of rows cleared to the points scored

Constants:

- the width and height of the board, in blocks
- the points earned for landing a piece
- an array of the point values for rows cleared (slot i of the array should hold the points earned for clearing i rows at once)
- a 2D array of strings which are the polyomino definitions

```
private static final int[] ROW_SCORE =
  { 0, 100, 250, 400, 800 };
```


Omino – GUI and JavaFX

- how do you manage the user interaction? how does pressing 'k' rotate a piece? how do you create all the methods that handle moving blocks?
  - the provided code handles detecting the user's key presses
  - you write methods of the `Game` class that have the functionality of what happens when the user presses keys
    - the handout describes what these are and what they should do
  - you add some code to the `Omino` class to link the key presses to the `Game` methods
    - the handout says what to do

Omino – Correctness and Robustness

- does "testers" refer to test cases?
  - "tester" refers to a class with a main program where test cases for another class are implemented
    - e.g. StringSet`Tester`
  - writing testers for `Block`, `Polyomino`, `Piece`, and `Board` is important so you know that those classes work before you put together the actual game play

Omino – Correctness and Robustness

- how do we come up with test cases? how do we know what the starting state would be?
  - start with black box tests (what are the different behaviors/outcomes?), then expand to white box tests (covering all of the code in the method)
    - for boolean methods, there are two behaviors: a “true” answer and a “false” answer
  - since you are testing class methods, the starting state is the object the method is called on
    - e.g. for `Board`'s `canPlace`, the starting state would be a `Board` containing some blocks (the input would be the parameters – a piece and a position for the piece)
      - you can use the same starting state and piece for several different test cases by choosing different positions

Omino – Correctness and Robustness

- when checking for things like trying to move a piece off the side of a board, should we throw an exception, use an assertion, or just not allow the action to happen?
  - is this a correctness issue or a robustness issue?
    - robustness, because it is about how the program is being used (the user tries to move left too many times)
    - → assertion or throwing `RuntimeExceptions` are not the right mechanisms
  - can the problem be handled in the same place it is detected?

In particular, `movePiece` should:

- determine the new piece and position resulting from the action (`LEFT`, `RIGHT`, `DOWN`, `DROP` just result in a new position; `ROTATE` results in a new piece as well as a new position) - but don't update the current piece or position yet
- if it is legal to place the (new) piece in the new position, update the current piece and position to the new information
- if the piece has landed (meaning the action was `DROP` or the action is `DOWN` and the piece couldn't be placed in its new position) --
  - score points for landing the piece
  - update the game over status if the landed piece sticks out above the top of the board (game is over)
  - add the piece to the board
  - clear any filled rows and update the score accordingly

    - yes, the “is legal” check is detecting the problem and not updating the current piece and position is how a problem is handled
      - → no need to throw or catch exceptions

Constants

- see section 4.8.3 in the book

`Game`

`Game` brings together the individual elements — the board, the current piece, piece movement, scoring — into the whole game.

You've been provided with a skeleton of part of the full `Game` class. Fill in and add to this skeleton as described in this section instead of creating a brand new class.

The board dimensions (in blocks), polyomino definitions and colors, and scoring information will be hardcoded into the game. Defining these things as constants brings the definitions together in one place and makes them easy to change.

Constants:

- the width and height of the board, in blocks
- the points earned for landing a piece
- an array of the point values for rows cleared (slot *i* of the array should hold the points earned for clearing *i* rows at once)
- a 2D array of strings which are the polyomino definitions
- an array of colors (type `Color`) where slot *i* holds the color for polyomino *i*

While constants are often `public`, these can be `private` since they exist only to make it easier to change the definitions within `Game`.

```
private static final int BOARD_WIDTH = 10, BOARD_HEIGHT = 21;
```


enums

- see section 2.3.5 in the book
  - for the project, you only need to use enums – you don't need to be able to write your own
- using `Action`
  - enums can be used just like a class type for variable and parameter declarations

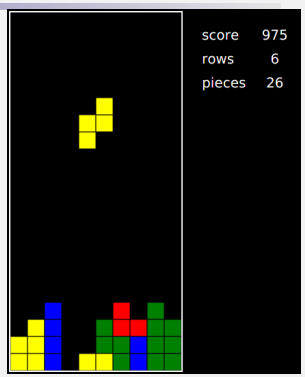
```
public void movePiece ( Action action ) { … }
```

  - enum values are referenced like constants

```
game_.movePiece(Action.DROP);
if ( action == Action.LEFT ) { … }
```

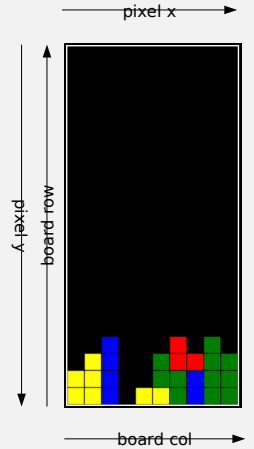
    - possible values are `Action.LEFT`, `Action.RIGHT`, `Action.ROTATE`, `Action.DOWN`, `Action.DROP`

Omino – Graphics and JavaFX



- a simple representation of the pieces is fine
  - draw a solid color rectangle outlined with black so you can tell the blocks apart
  - a fancier look is extra credit
- see section 3.9.1 for information drawing shapes
  - set fill color and outline color
  - fill and outline rectangles
  - the `GraphicsContext` object needed is a parameter to the `drawBoard` and `drawCurrentPiece` methods in `Omino`
  - a color is type `Color` (from a `javafx` package, not `java.awt`)

Omino – Coordinates



- board positions are (row,col) in a 2D array
- drawing on the screen requires (x,y) coordinates within the drawing area

Omino – Coordinates

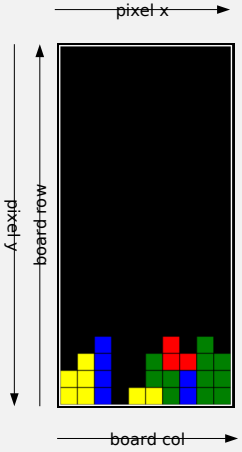
drawing the board means going through each square of the board and, if occupied, drawing a rectangle in the right spot

what's the pattern for going through every slot of a 2D array?

```
for ( int row = 0 ; row < height ; row++ ) {
  for ( int col = 0 ; col < width ; col++ ) {
  }
}
```

need upper left corner of each rectangle – if we just had variables with the right values...

```
for ( int row = 0 ; row < height ; row++ ) {
  for ( int col = 0 ; col < width ; col++ ) {
    …
    g.fillRect(top,left,BLOCK_SIZE,BLOCK_SIZE);
    …
  }
}
```



Omino – Coordinates

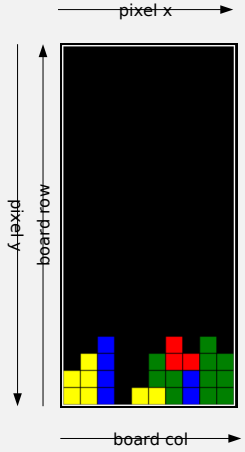
```
for ( int row = 0, top = ?? ; row < height ;
      row++, top = ?? ) {
  for ( int col = 0, left = ?? ; col < width ;
        col++, left = ?? ) {
    …
    g.fillRect(top,left,BLOCK_SIZE,BLOCK_SIZE);
    …
  }
}
```

`top` and `left` can be added as loop variables since we only need them for these loops

to initialize – what are the right values for the rectangle at row 0, col 0 on the board?

to update – how does `top` change when the row increases? how does `left` change when the col increases?

from this you might be able to work out formulas for how to compute top and left given row and col – you don't need those formulas to draw the board, but they might be handy for drawing the current piece...



Omino – Graphics and JavaFX

Technical note –

- to make the display update when you change values in `Game`, use `firePropertyChange`

```
firePropertyChange(BOARD_PROPERTY);     // change to board contents
firePropertyChange(CURPIECE_PROPERTY);  // change current piece (piece and/or position)
firePropertyChange(SCORE_PROPERTY);     // change to the score
firePropertyChange(NUMPIECES_PROPERTY); // change to the number of pieces played
firePropertyChange(NUMROWS_PROPERTY);   // change to the number of rows cleared
```