Lab: Tetris

Background: `Block`

The `Block` class represents a single 1x1 block in the game of Tetris. Here is a summary of its constructor and methods:

`Block` Class

```
Color getColor()
void setColor(Color newColor)
BoundedGrid<Block> getGrid()
Location getLocation()
void putSelfInGrid(BoundedGrid<Block> gr,
                   Location loc)
void removeSelfFromGrid()
void moveTo(Location newLocation)
String toString()
```

Background: `BlockDisplay`

The `BlockDisplay` class represents a window that shows the contents of a particular `BoundedGrid<Block>`. Here is a summary of its constructor and methods:

`BlockDisplay` Class

```
public BlockDisplay(BoundedGrid<Block> grid)
public void showBlocks()
public void setTitle(String title)
public void setArrowListener(ArrowListener listener)
```

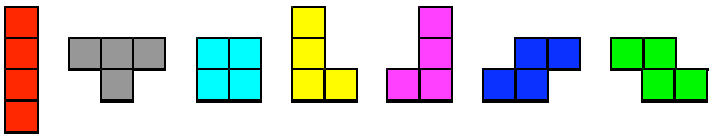
Exercise 1: Block Pop-Up Window

Create a new class called `Tetris`, which will be the main class of our Tetris game. It should have instance variables for keeping track of a `BoundedGrid<Block>` and a `BlockDisplay` (which displays the contents of the `BoundedGrid`). In the constructor, create the `BoundedGrid<Block>` to have 20 rows and 10 columns, and create the display. Use `BlockDisplay`'s `setTitle` method to set the window title to be "Tetris". Test to make sure your empty Tetris board is displayed correctly when you

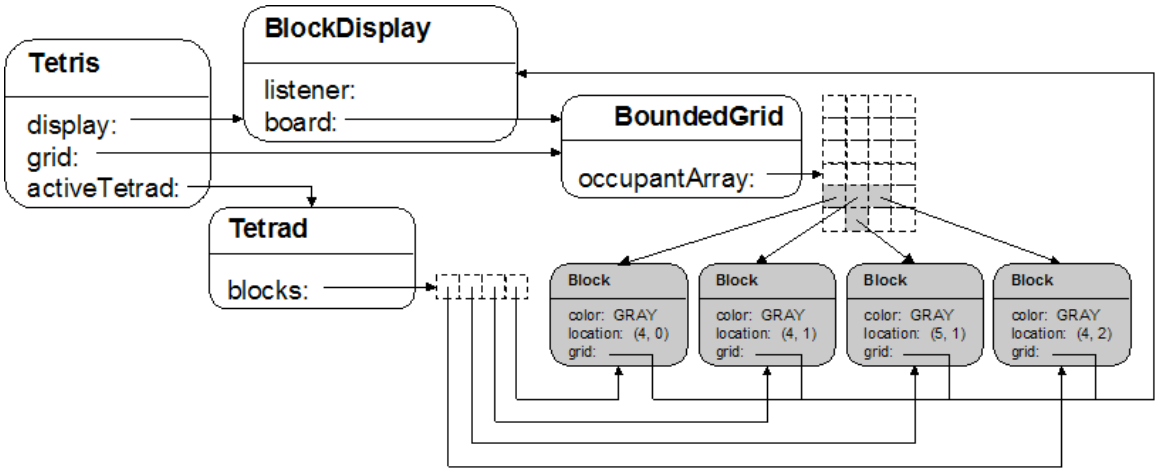
call your `Tetris` class's constructor.

Exercise 2: The Communist Bloc

Shapes composed of four blocks each are called *tetrads*, the leading actors of the Tetris world. Tetrads come in seven varieties, known as *I*, *T*, *O*, *L*, *J*, *S*, and *Z*. They are shown here.



You will therefore need to create a `Tetrad` class, which will keep track of an array of four `Blocks`. An instance diagram showing the role of the `Tetrad` class appears on the next page. (Note that the environment only keeps track of the blocks, and does not know anything about tetrads. Instead the `Tetris` class will eventually keep track of the tetrad being dropped.)



Go ahead and create the `Tetrad` class with a single instance variable of type `Block[]`. Also, write a helper method in the `Tetrad` class called `addToLocations`, which should behave as follows.

```
//precondition:  blocks are not in any grid;
//               locs.length = 4.
//postcondition: The locations of blocks match locs,
//               and blocks have been put in the grid.
private void addToLocations(BoundedGrid<Block> grid,
                            Location[] locs)
```

(We won't be able to test this code just yet.)

Exercise 3: I, Tetrad

Now create the constructor, which should take in the grid and initialize this `Tetrad` as an I-shaped block in the middle of the top row of the grid. Here is a suggested outline to follow in writing your constructor:

1. Initialize the `Tetrad`'s array of `Block`s so that it contains four new `Block` objects.
2. Declare a local temporary variable of type `Color` (which means you'll need to import `java.awt.Color` at the top of the file).
3. Declare a local temporary variable of type `Location[]`, capable of storing four `Location`s.
4. Assign the color `Color.RED` (for example) to your temporary `Color` variable.
5. Assign the locations (0, 3), (0, 4), (0, 5), and (0, 6) to the temporary array of `Location`s. These locations will ultimately make your `Tetrad` appear in a horizontal I-shape at the top of the grid.
6. Set the color of each `Block` to be the color indicated by your temporary `Color` variable.
7. Call `addToLocations` to add the blocks to the temporary array of `Location`s you created.

Finally, add an instance variable to the `Tetris` class to store the "active tetrad" (the one that's currently falling and being maniupulated with the arrow keys, eventually). In the `Tetris` class's constructor, initialize this instance variable to refer to a new `Tetrad`, and then finally call the `BlockDisplay`'s `showBlocks` method, so that you can actually see the `Tetrad` you've created. Go ahead and test that you can see your I-shaped `Tetrad` appear.

Exercise 4: Four Blocks and Seven Shapes Ago, ...

Modify the `Tetrad` class's contructor so that it first picks a random number from 0 to 6. If it picks 0, create a red I-shaped tetrad as before. But, for each of the other numbers, pick one of the other 6 shapes. For that shape, pick an appropriate color (such as `GRAY`, `CYAN`, `YELLOW`, `MAGENTA`, `BLUE`, or `GREEN`) and 4 starting locations approximately in the top middle of the board. Now run your program a few times to test that a random tetrad appears at the top of your tetris window.

Exercise 5: Unblocking

Add the following two helper methods to your `Tetrad` class.

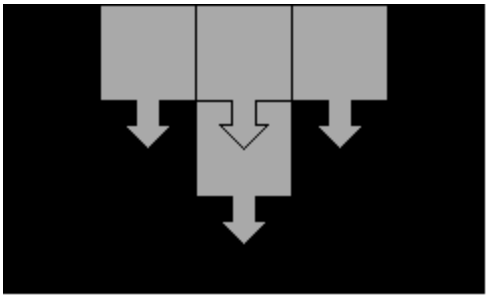
```
//precondition:  Blocks are in the grid.
//postcondition: Returns old locations of blocks;
//               blocks have been removed from grid.
private Location[] removeBlocks()

//postcondition: Returns true if each of locs is
//               valid (on the board) AND empty in
//               grid; false otherwise.
private boolean areEmpty(BoundedGrid<Block> grid,
                         Location[] locs)
```

(We won't be able to test this code just yet.)

Exercise 6: Lost in Translation

We'll now write `Tetrad`'s method `translate` method, which will shift the `Tetrad` over and down by the given amount, first making sure that the `Tetrad`'s potential new position is valid and empty. For example, suppose we have a T-shaped `Tetrad` in the top middle of an empty grid, and we wish to move it down by one row, as shown below.



Clearly, we ought to allow this move. But notice that one of the locations we're attempting to move this `Tetrad` into is *already occupied by another block in the `Tetrad` itself.* So, it's not enough to make sure that the new locations are empty.

To make sure we handle this situation correctly, we'll always translate a `Tetrad` as follows:

1. Ask any block for its grid, and store it in a temporary variable.
2. Remove the blocks in the `Tetrad`.
3. Check if the new locations are empty.
4. If the new locations are empty, add the `Tetrad` to the new locations. Otherwise, add the `Tetrad` back to its original locations.

Go ahead and write the `Tetrad` class's `translate` method, making use of `addToLocations`, `removeBlocks`, and `areEmpty`.

```
//postcondition: Attempts to move this tetrad deltaRow
//               rows down and deltaCol columns to the
//               right, if those positions are valid
//               and empty; returns true if successful
//               and false otherwise.
public boolean translate(int deltaRow, int deltaCol)
```

Now temporarily modify the `Tetris` constructor to translate the active tetrad, and make sure the tetrad appears in the correct location. Be sure to test a translation to an illegal position.

Exercise 7: Block Party

Now that blocks have the potential to move, let's teach them to dance! The `BlockDisplay` class keeps track of an `ArrowListener`. Whenever the `BlockDisplay` window has "focus" and an arrow key is pressed, a message is sent to the `ArrowListener`. The `BlockDisplay` class doesn't actually care what the `ArrowListener` chooses to do with this message, and hence `ArrowListener` has been defined as an interface, shown here.

```
public interface ArrowListener
{
     void upPressed();
     void downPressed();
     void leftPressed();
     void rightPressed();
}
```

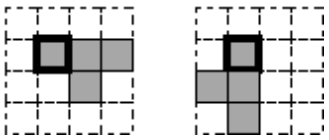
Modify the `Tetris` class so that it implements the `ArrowListener` interface. Each `ArrowListener` message should cause the `Tetris` game's active tetrad to move one row or column in the indicated direction (including up, for now). Be sure to call the display's `showBlocks` method to tell it to redraw itself whenever your tetrad moves.

In the `Tetris` class's constructor, when you create the `BlockDisplay`, call the `BlockDisplay`'s `setArrowListener` method, to tell the display that your `Tetris` object itself would like to be notified whenever an arrow key is pressed. This will tell the display to call the methods you just implemented.

Now go ahead and test your code. A random tetrad should appear at the top of your Tetris window. You should be able to move it around with the arrow keys. Your program should prevent you from moving the tetrad outside of the window.

Exercise 8: Spin Cycle

Next, we'll make our tetrads rotate. We'll have them rotate clockwise by 90 degrees around the location of the tetrad's first block (`blocks[0]`) in the tetrad. The following diagram shows a single rotation of a T-shaped tetrad.



Given a block at position (*row*, *col*), there is a surprisingly simple formula (shown below) to find its new location (*row'*, *col'*), following a 90 degree clockwise rotation about a point at (row_0, col_0).

$$row' = row_0 - col_0 + col$$
$$col' = row_0 + col_0 - row$$

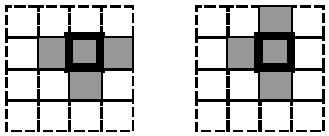
Use this formula to add the following `rotate` method to the `Tetrad` class. Your `rotate` code should wind up looking very similar to your `translate` code.

```
//postcondition: Attempts to rotate this tetrad
//               clockwise by 90 degrees about its
//               center, if the necessary positions
//               are empty; returns true if successful
//               and false otherwise.
public boolean rotate()
```

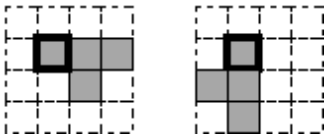
Now modify the `Tetris` class so that it calls `rotate` to rotate the active tetrad clockwise by 90 degrees whenever the up arrow is pressed (instead of shifting the tetrad up). Test to make sure your tetrads rotate appropriately (but sometimes off-center), and that your game prevents you from rotating the tetrad off the edge of the window.

Exercise 9: A Pivotal Mistake

You've probably noticed that some of your tetrads seem to be rotating around the wrong block. What we'd like to do is make sure that each tetrad rotates around one of its center-most blocks, as shown below for the T-shaped tetrad.



good block to rotate around



bad block to rotate around

Fortunately, this problem is easily fixed by going back to the `Tetrad` class's constructor, and simply changing the order of the locations for each block, so that the first location in each `Tetrad` is approximately at the center of the `Tetrad`. Make these changes and test that your tetrads rotate correctly now.

Exercise 10: The Sky Is Falling

Implement a method `play` in `Tetris`, which should repeatedly (forever)

1. pause for 1 second,
2. move the active tetrad down one row, and
3. redraw the display.

You can tell Java to pause for a second by inserting this hideous line of code:

```
try { Thread.sleep(1000); } catch(Exception e) {}
```

Call the `play` method after constructing a new `Tetris` (but don't call `play` from inside the constructor itself). When you test your program, you should find that you can still shift and rotate the tetrad, but that it will now slowly drop on its own. When it gets to the bottom, the tetrad should stop falling (although you'll still be able to slide it around for now).

Exercise 11: Tetrad Comrades

Now modify the `play` method so that, when it is unable to shift the active tetrad down any further, it creates a new active tetrad. (Hint: Check `translate`'s return value, but make sure you only call `translate` once per loop!) Test your game, and see how much you've accomplished!

Exercise 12: Death Row

We would like to clear any rows the user completes. First, go ahead and implement the following helper methods in the `Tetris` class.

```
//precondition:  0 <= row < number of rows
//postcondition: Returns true if every cell in the
//               given row is occupied;
//               returns false otherwise.
private boolean isCompletedRow(int row)

//precondition:  0 <= row < number of rows;
//               given row is full of blocks
//postcondition: Every block in the given row has been
//               removed, and every block above row
//               has been moved down one row.
private void clearRow(int row)
```

Now use the above helpers to implement the following `Tetris` method.

```
//postcondition: All completed rows have been cleared.
private void clearCompletedRows()
```

Whenever a tetrad stops falling, call `clearCompletedRows`. Now go play Tetris!

(Yes, your game will misbehave when you lose, but there's a simple work-around to this problem: Don't lose.)

Additional Suggestions

- Keep score and gradually increase the speed at which tetrads fall.
- Introduce levels. The game typically begins with level 1. Every time the player clears 10 rows, advance to the next level. Blocks should fall a little faster on each successive level. Clearing 1 row earns 40 * *level* points. Clearing 2 rows at once earns 100 * *level* points. Clearing 3 at once earns 300 * *level* points, and 4 earns 1200 * *level* points. Show the player's level and score in the window title.
- Immediately drop the active tetrad to the bottom when the spacebar is pressed.
- Identify when a player has lost, and respond accordingly.
- Show what tetrad will be falling next.
- When starting a new game, let the player choose to fill the bottom rows with random blocks.
- Drop special kinds of blocks that act as bombs, etc.
- Improve the artwork, animation, effects, etc.
- In a new directory, implement Super Puzzle Fighter, the game of Snake, or some other game that can make use of the `BlockDisplay` class.

Note About Threads

Tetris, as we've implemented it, is a multi-threaded application. One thread begins in the `main` method, creates the grid and display, and then loops around inside the play method, continually shifting the active tetrad down one row, and occasionally clearing rows. The other thread is started automatically by Java to draw the user interface and handle key presses. This means that it may sometimes happen that the main thread is shifting the active tetrad down or clearing rows *at the same time as* the user is pressing an arrow key to translate/rotate the active tetrad. If this happens, all sorts of things may go wrong. Maybe only a couple blocks in the active tetrad will shift down, or maybe an exception will be thrown. Solving this problem is extremely tricky, *so it's best to pray that you never need to know about this stuff.*

Thankfully, for the most part, this problem seems to be extremely rare. But every now and then, for whatever reason, it'll happen that some student hits this problem constantly. The problem can be fixed by declaring some methods in the `Tetris` class as `synchronized`, meaning that only one of those methods may be called at a time. For example, it may be best to synchronize `upPressed`, `downPressed`, etc., along with the inside of the `play` method's loop (in other words, put the inside of the loop in a `synchronized` helper method).