Lab: Tetris

Exercise 1: Block Pop-Up Window

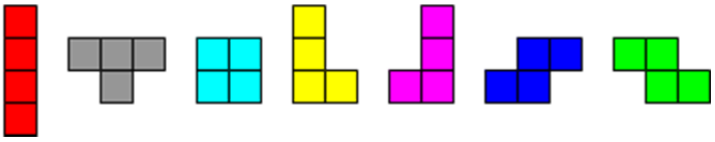
Do not create any packages. Put all of your code in the default package.

Create a new class called `Tetris` in the default package, which will be the main class of our Tetris game. It should have instance variables for keeping track of a `MyBoundedGrid<Block>` and a `BlockDisplay` (which displays the contents of the `MyBoundedGrid`). In the constructor, create the `MyBoundedGrid<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", and the `showBlocks` method to draw the window. Test to make sure your empty Tetris board is displayed correctly.

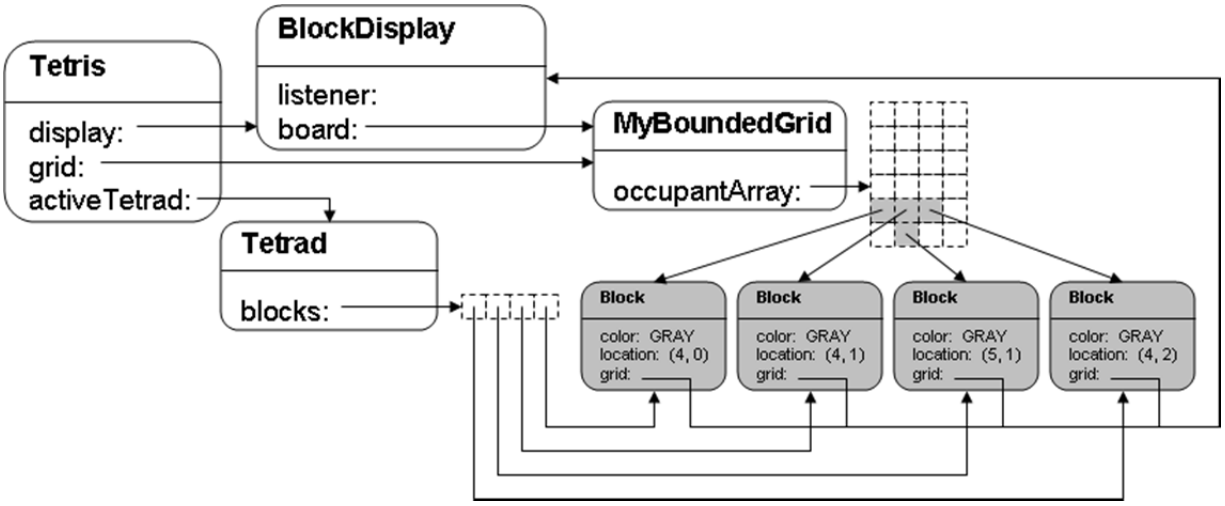
Exercise 2: The Communist Bloc

Do not create any packages. Put all of your code in the default package.

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 in their suggested colors.



You will therefore need to create a `Tetrad` class, which will keep track of two things: an array of four `Block`s, and the `MyBoundedGrid<Block>` in which they live. 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 appropriate instance variables. Give it a helper method called `addToLocations`, which should behave as follows.

```
/**
* (Your method documentation)
* 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(MyBoundedGrid<Block> grid,
                            Location[] locs)
```

Now create the constructor, which should take in the grid. The constructor should pick a random one of the seven tetrad shapes. Use a different color for each shape, such as `RED`, `GRAY`, `CYAN`, `YELLOW`, `MAGENTA`, `BLUE`, and `GREEN`. Initialize the tetrad to appear in the middle of the top row of the environment. Be sure to use the `addToLocations` method.

Finally, add an instance variable in the `Tetris` class to keep track of the active tetrad—the one currently falling, rotating, etc. Initialize the active tetrad in the constructor, and test that a random tetrad appears at the top of your Tetris window.

Exercise 3: Lost in Translation

Do not create any packages. Put all of your code in the default package.

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 and empty in grid;
//               false otherwise.
private boolean areEmpty(MyBoundedGrid<Block> grid,
                         Location[] locs)
```

Now write the following 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)
```

Test that you can translate a tetrad. 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 4: Block Party

Do not create any packages. Put all of your code in the default package.

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. 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 following method in `BlockDisplay` so that the display can find the methods you just implemented.

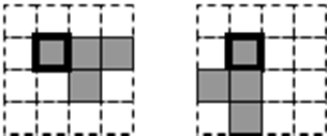
```
public void setArrowListener(ArrowListener listener)
```

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.

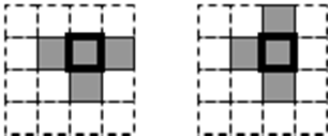
Exercise 5: Spin Cycle

Do not create any packages. Put all of your code in the default package.

Next, we'll make our tetrads rotate. We'll have them rotate clockwise by 90 degrees about a particular point P_0. The following diagrams show a single rotation of a tetrad for two different choices of P_0. You'll probably agree that it makes more sense to choose the middle block to be P_0 for the *T* tetrad.



Leftmost block as P_0



Middle block as P_0

In implementing your choice of P_0, one reasonable design is always to use `blocks[0]`. If you use this approach, you'll need to go back and look at your `Tetrad` constructor code, so that a centrally located block is always assigned to `blocks[0]`. Alternatively, you might keep track of the index in the `blocks` array where the P_0 block appears. However you approach this problem, be sure that each tetrad rotates about a centrally located point.

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 P_0 at (row_0, col_0).

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

Use these ideas to add the following method to the `Tetrad` class. Then modify the `Tetris` class so that it rotates the active tetrad clockwise by 90 degrees whenever the up arrow is pressed (instead of shifting the tetrad up).

```
//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()
```

Test to make sure your tetrads rotate appropriately, and that your game prevents you from rotating the tetrad off the edge of the window.

Exercise 6: The Sky Is Falling

Do not create any packages. Put all of your code in the default package.

Implement a method `play` in `Tetris`, which should repeatedly pause for 1 second (using the following code segment), move the active tetrad down one row, and redraw the display.

```
try
{
     //Pause for 1000 milliseconds.
     Thread.sleep(1000);
}
catch(InterruptedException e)
{
     //ignore
}
```

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).

You may encounter an unusual problem at this stage in the development of the game. When you press one of the arrow keys, the corresponding event listener method is called *from the event dispatch thread.* The arrow listener methods cause a modification of the existing tetrad by translating or rotating, both of which involve taking the blocks out of the grid and putting them back.

Suppose that in the middle of one of these modifications the main thread wakes up and tries to translate the tetrad while part of it is not contained in the grid. You can see the potential for lots of trouble. The same thing can happen if you press the arrow keys fast enough causing multiple events to be in progress at the same time.

We can fix this by introducing a *semaphore.* A semaphore is used to restrict access by requiring that a process acquire the semaphore before doing any work. We can use semaphores to prevent concurrent modification of a tetrad by having the `translate` and the `rotate` method acquire a semaphore before doing any work.

In the `Tetrad` class, add an instance variable for a semaphore called `lock` using the following code:

```
private Semaphore lock;
```

(You will need to import `java.util.concurrent.Semaphore`)
In the constructor, create a `Semaphore` object as follows:

```
lock = new Semaphore(1,true);
```

Now, change the `translate` and `rotate` methods as follows:

```
try
{
     lock.acquire();
     // your lousy code here
}
catch (InterruptedException e)
{
     // did not modify the tetrad
     return false;
}
finally
{
     lock.release();
}
```

This will prevent concurrent modification of the tetrad.

Exercise 7: Tetrad Comrades

Do not create any packages. Put all of your code in the default package.

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.) Test your game, and see how much you've accomplished!

Exercise 8: Death Row

Do not create any packages. Put all of your code in the default package.

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,` but make sure that there is no active tetrad while this happens. Now go play Tetris!

Additional Suggestions

- Keep score and increase the speed at which tetrads fall.
- Introduce levels. The game 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.
- 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.
- Use a gravity variant that supports chain reactions. See http://en.wikipedia.org/wiki/Tetris#Gravity.
- Drop special kinds of blocks that act as bombs, etc.

- Improve the artwork, animation, effects, etc.
- In a new directory, implement Super Puzzle Fighter, or some other puzzle game.