DUE DATE: OCTOBER 28, 2020 AT 11:59PM

## Instructions:

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment.
- Only submit the **java files**. Do ***not*** submit any other files, unless otherwise instructed.
- To submit the assignment, upload the specified files to the **Assignment 2** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn. You will lose marks for not following these standards.
- After the due date and time, assignments may be submitted but will be subject to a late penalty. Please see the ROASS document published on UMLearn for the course policy for late submissions.
- If you make multiple submissions, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and **pursue academic dishonesty vigorously**.
- Your Java programs must compile and run upon download, without requiring any modifications.
- Automated tests will be used to grade the output of your assignment. If you do not follow precisely the guidelines detailed below, these automated tests can fail and you will lose marks.
- ArrayLists are not allowed in this assignment. You must use arrays and partially filled arrays, as described.

## Assignment overview

In this assignment, you will write a set of interrelated classes that will support the implementation of the board game known as Reversi or Othello. It has a fairly simple set of rules, but they won't be described here. Instead, look at **Section 3 (Rules)** at https://en.wikipedia.org/wiki/Reversi . (**You can ignore the descriptions of time clocks, transcripts, row and column labels, and other items not part of the basic rules.**) There are plenty of implementations of this game, and you can play against other people online, if you'd like to try it. Here is a short video of how to play reversi (https://www.youtube.com/watch?v=Ol3Id7xYsY4) and a link to practice this game (https://cardgames.io/reversi/).

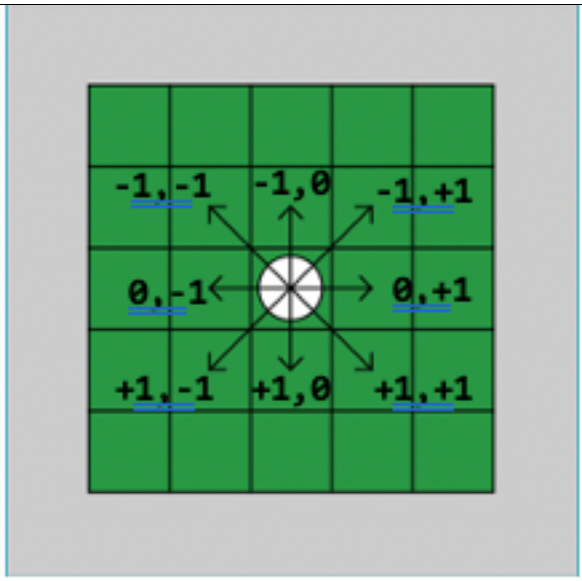
Keep all of your methods short and simple. In a properly-written object-oriented program, the work is distributed among many small methods, which call each other. Make sure to call methods already created when appropriate, instead of duplicating code for no reason. Unless specified otherwise, all instance variables must be **`private`**, and all methods should be **`public`**. Also, unless specified otherwise, there should be ***no*** **println** statements in your classes. Objects usually do not print anything, they only return **String** values from methods.

## Phase 1: A set of related classes

In this phase, you will create a set of four classes of objects that will represent individual moves in a Reversi game. Since a move must always "flip" some of the opponent's pieces in up to 8 different directions, some classes to represent these directions will also be useful.

### The **`Direction`** class:

An object of type **`Direction`** will hold two **`int`** values to represent one of the 8 possible vertical, horizontal, or diagonal directions on a game board, as shown below. (This is not only useful for Reversi, but could be used in many other games, too.) The two integers give the change to the row index, and to the column index, that would move from one square to the next in the indicated direction. These numbers will always be -1, 0, or +1. The two numbers should not both be 0 (since that wouldn't change any index at all), but the other 8 possible combinations are valid.




Implement the following things in your **Direction** class:

- Two instance variables (private, as usual) to hold the two integer values (the row change and the column change).
- A constructor with two integer parameters to initialize the two instance variables. Put the row first, and the column second.
- Two accessors to allow your other classes to obtain these two integer values. Do not supply mutators. The integers are never supposed to change after one of these objects is created.
- The usual **toString()** method. It should return a **String** containing words, not integers. Use the words "up", "down", "left", and "right" instead of -1 and +1. Use either one word, or two words, whichever is appropriate, and surround the words with "angle brackets" (< >). For example,

  **new Direction(-1,1).toString()** should give **"<up right>"**
  **new Direction(1,-1).toString()** should give **"<down left>"**
  **new Direction(1,0).toString()** should give **"<down>"**
  **new Direction(0,-1).toString()** should give **"<left>"**

## The **DirectionList** class:

An object of type **DirectionList** will hold a list of **Direction**s (0 to 7). You can use an array to store them. If you know how to use an **ArrayList** (a topic which won't be covered until later in the course), you can use that too.

Implement the following things in your **DirectionList** class:

- Instance variables (private, as usual) to store the list of **Direction**s. Use Array or ArrayList.
- A constructor to create an empty list.
- A method **public void addDirection(Direction d)** to add a **Direction** to the end of the list.
- A method **public int length()** that will return the size of the list.
- A method **public Direction getDirection(int i)** which will return the element at position **i** in the list (positions should start from 0 as usual).
- A **toString()** method, as usual. It should list the **Direction**s in the list, surrounded by **{ }** and separated by commas. For example, **{<right>,<down right>,<left>}**. Don't put a comma after the last one.
- A method **public static DirectionList allDirections()** which will return a list of all 8 possible directions. This will be very useful later in the assignment. Note that this is a **static** method.

## The **Move** class:

An object of type **Move** will hold information describing one possible move in a game of Reversi. Since a move in this game will always "flip" opponent pieces in one or more directions, a **Move** object will also keep track of these directions. (This class doesn't calculate what the directions are – it doesn't have any way to do that – it just stores them.)

Implement the following things in your **Move** class:

- Instance variables (private, as usual) to store the coordinates of the move (a row number and a column number), and a **`DirectionList`** giving the directions in which opposing pieces will be flipped.
- A constructor which accepts a row number, a column number, and a **`DirectionList`**.
- Accessors for the row number, the column number, and the **`DirectionList`**.
- A **`toString()`** method which returns a String in the following format:

```
(2,5) flips directions {<left>,<down left>}
```

The **`MoveList`** class:

An object of type **`MoveList`** will hold a list of 0 or more **`Move`**s.

Implement the following things in your **`MoveList`** class:

- Instance variable(s) (private, as usual) to store the list of moves. If you use an array, give it a maximum capacity of 32 moves.
- A constructor to create an empty list of moves.
- A method **`public void addMove(Move m)`** to add a **`Move`** to the list.
- A **`toString()`** method which will return a multi-line **`String`** using **`'\n'`** characters to separate the moves, so that if the **`String`** were printed it would look like this:

```
(1,2) flips directions {<down>}
(2,0) flips directions {<right>,<down right>}
(2,5) flips directions {<left>,<down left>}
(4,0) flips directions {<right>}
```

- A method **`public boolean isEmpty()`** which returns **`true`** if the list is empty.
- A method **`public Move randomMove()`** which returns one of the moves in the list, *at random*. If the list is empty, it should return **`null`**.

### Testing

There are four test classes that you can test your Direction, DirectionList, Move and MoveList classes. Make sure your classes work properly before starting the next phase as you need those classes in the next phase.

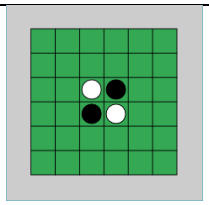
## Phase 2: Board class for a Reversi (Othello) game

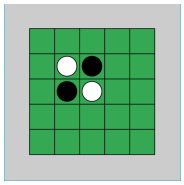
In this step, you will create a **`Board`** class that will handle all of the tasks needed to play a Reversi game. This includes finding valid moves, making moves, flipping pieces, detecting the end of the game, etc. But it does not contain any user interface code for actually playing the game.

Define a **`Board`** class. Objects of this type will store the current state of a Reversi board (which pieces are on the board, and where), provide relevant information about that state (e.g. what the valid moves are), and change that state (make moves and flip pieces).

Implement the following things in your **`Board`** class:

- Instance variable(s) (private) to store the board, using a ***multidimensional array*** of **`int`** values. The board will always be square, but it may be any size from 2x2 up. (A normal Reversi board is 8x8, but smaller boards will be convenient to test your program.)
- Any square on the board may be empty, or contain a black piece or a white piece. Assign three **`int`** values (0 for empty, 1 for black and 2 for white) to represent these possibilities, and define **`public`** constants **`EMPTY = 0`**, **`BLACK = 1`**, and **`WHITE = 2`**.
- A constructor which takes a single **`int`** parameter giving the desired size of the board, and sets up the board for the start of a game. There are always 2 black pieces and 2 white pieces in an X pattern in the centre of the board at the start of a game, as shown below (for 6x6 and 5x5 boards). If the size of the board is an even number, they should be in the exact centre. If the size of the board is an odd number, place them slightly up and to the left since they can't go in the centre. The piece closest to the top left should be a white one.





- A **`toString()`** method, which will return a suitable representation of the board, such as the one shown below. You could use X and O for Black and White:

```
...OOO
..X.OO
..XXOO
..XXXX
....O.
......
```

- A method **`public static int opponentOf(int player)`** which will return the integer representing the other player. So **`opponentOf(BLACK)`** should give **`WHITE`**, and vice-versa.
- A method **`public static String nameOf(int player)`** which will return a String giving the name of that player. Simply return "Black" or "White".
- A method **`public int getScore()`** which will return the score in the game, which is simply the number of black pieces on the board minus the number of white pieces. A positive number means Black is winning, and a negative number means White is winning.
- Now for the complicated one. Write a method **`public MoveList allValidMoves(int player)`** which will return a list of ***all*** of the valid moves that could be made by the indicated player. Remember that a **`Move`** object contains a list of all the directions in which opponent's pieces will be flipped, so calculating all of that information is part of the job. A move is valid if 1) it is played on an empty square, and 2) it flips at least one of the opponent's pieces. If there are no valid moves, it should return an empty list (not just **`null`**).
  - Do not write this as one big method!
  - Use **several smaller `private`** helper to break down this task into manageable sections of code.
  - You can do this in any way that you like. It would be sensible to make methods that focus on one particular square, or one particular direction from a square, or similar smaller tasks.
- Write a method **`public void makeMove(int player, Move theMove)`** which will make the given move, for the given player. This is the only method that will actually make changes to the board. It must place the new piece on the board, and flip all of the appropriate opponent pieces, in the directions indicated by **`theMove`**.

## Phase 3: I/O and Exceptions

In this phase we want to save the progress has been made and save it as a text file by calling **saveFile(String fileName)** method. This file has a special format. The first line of the file will be the dimension of the board and the following lines represent the board setup (**.** for Empty, **O** for Black, and **X** for White). Here is an example

```
4
..O.
.OO.
XXXX
.O.X
```

As the second option to start the game, instead of creating a new regular board, you can also use a method with which you can import a specific board setup. To do this, you need to define a method, **importBoardSetup(String fileName),** which reads the data from a text file specified by 'fileName' and set the dimension as well as set the values of the 2D array representing the board. The following is an example of an input text file:

```
..XXX.
O.XXX.
.OXOX.
..O.X.
....X.
```

To get full mark for this section you need to define the following methods inside your Board class:

- Your Board class should have **saveFile(String fileName)** method. You need this method to save your board (as a text file).
- You need to write **importBoardSetup(String fileName)** to read a text file that represents a Board setup (set the size of the board and set the values of the 2D array representing the board). This method should be able to throw IOException if the data in not valid. Special exception message includes:
  - "**The number of columns and rows doesn't match**"
  - "**Unrecognized character**", for example if there is a character(s) other than ., x, o in the file
  - "**number of rows/columns less than 2**"
  - Make sure that you are printing the messages mentioned above for special cases.
- You need to define a new constructor **Board(String fileName)** that is going to create a new Board based on the imported Board setup by calling the **importBoardSetup(String fileName)** inside your constructor.

## Test

You can use the supplied main program RandomOthello.java to test your Board class. This program plays games of Othello by selecting moves for Black and White at random. You can modify this program if you wish to allow human players to pick moves, but that is not required. (Don't hand in this program.)

## Hand in

Submit only the .java files for your four classes (**`Board.java`**, **`Direction.java`**, **`DirectionList.java`**, **`Move.java`**, **`MoveList.java`**). ***Do not submit .class or .java~ files!*** You do ***not*** need to submit the **RandomOthello.java** **`java`** files that were given to you. If your submitted code fails to compile and run, you will lose ***all*** of the marks for the test runs. The marker will ***not*** try to run anything else, and will ***not*** edit your files in any way. ***Make sure none of your files specify a package at the top, otherwise your code will not compile on the marker's computer!***