CMSC 132 Project #1 Fall 2012

Date due: Thursday, September 27, 10:00:00 p.m.

1 Introduction

In this project you will write a program that simulates playing the game of Othello®, a two–player game that will be explained in detail below. The purpose of the project is to get a lot more practice using two–dimensional arrays, which were covered near the end of CMSC 131.

Your program will not actually play the game of Othello, in that it will not have logic to allow it to be able to make moves on its own. In other words, a person will not be able to play against your program, but it will know the game rules and be able to keep track of the state of a game being played as moves are made. In this way one person could use your program to play both sides of a game of Othello, entering the moves for both players in alternation, or two people could use it to play a game of Othello, each entering moves in turn. It would not actually be extremely difficult to write a program that would really play Othello using a simple and naive strategy, but the project is probably difficult enough just as it is. Therefore you are strongly urged to begin working hard on it **right away**.

We are providing you with an interactive GUI (graphical user interface) that will allow you to easily exercise your code, using mouse clicks to specify moves. The GUI will not be used in grading, since obviously the submit server does not pop up with a GUI when a program is submitted. All the public and secret tests will exercise your methods using JUnit tests, as was the case in Homework #1. The GUI will be explained a bit more below.

Check out the project starting code using the procedure given in the first homework assignment. The project to check out is named `proj1`.

Due to time constraints and other factors it is not practical to provide detailed information or assistance regarding programming assignments via email, so please ask questions you may have about this project in person, either during office hours, or before or after class.

2 The rules of Othello

Some variations of the game exist, so to be sure that everyone is working from the same assumptions we explain our version of the rules here.

Othello is played on a board of eight by eight squares, like chess or checkers, except all the squares are the same color. There are two players, one of whom plays black and the other of whom plays white. Each player has game pieces with their respective color that are placed on the board to make moves. Initially the board is empty except for the center four squares, two of which are occupied by black's pieces, with the other two occupied by white's pieces; the starting board configuration is shown in Figure 1.

We will adopt the convention that board's rows are numbered from 0 to 7 starting at the top, and its columns are numbered from 0 to 7 beginning on the left.

The black player plays first and players alternate turns. During each turn a player plays or places one piece of their color on one square in the board, in any square where the move would be valid. A move is valid if:

- the square where the piece is placed is currently empty, and
- if placing the piece in that square would cause at least one of the opponent's pieces to be trapped.

A trapped piece is one that lies between the piece that is being played, and any other piece of the same color that was already on the board. Pieces may be trapped in any direction– horizontal, vertical, or diagonal– however, only any adjacent or contiguous sequences of the opponent's pieces are trapped. Assuming a move is valid and at least one opponent's piece would be trapped, all of the consecutive pieces that are trapped by the move are changed to the color of the player who made the move.

For example, given the initial state of the board, and the fact that black makes the first move, all of the squares marked with an $\times$ in Figure 2 are valid moves for black.

Suppose the board looked like Figure 3 at some point during the game when it was black's turn, and black decides to place his or her piece in the square marked $\times$. Then two of white's pieces directly above the new piece (at (3,3) and (4,3)) will be trapped and change to black, one white piece to the upper–right (at (4,4)) will be trapped, as well as two white pieces to the right of the new piece (at (5,4) and (6,4)), resulting in the board contents following the move as shown in Figure 4. Notice that there are two other white pieces below the newly–placed black piece, one on its upper left (at (4,2)), and one on its lower right (at (6,4)), but neither of them are flipped because they are not trapped between the new black piece and some other black piece. Notice that the white piece in square (1,3) was not flipped, because although it's between the piece being placed and another black piece (the one at square (0,3)), only the consecutive sequences of white's pieces are trapped, and the black piece in square (2,3) ends the vertical series of adjacent white pieces in squares (4,3) and (3,3).

The game continues until either:

- A player does not have any valid move when it is their turn; in this case they automatically lose (the other player wins).
- Every square of the board is occupied, in which case the number of squares occupied by pieces of each color determine which player wins. If there are more black pieces than white pieces on the board when the board becomes full the black player wins, while if there are more white pieces white wins. If there are exactly the same number of pieces of both colors (32 each) the game is a tie.

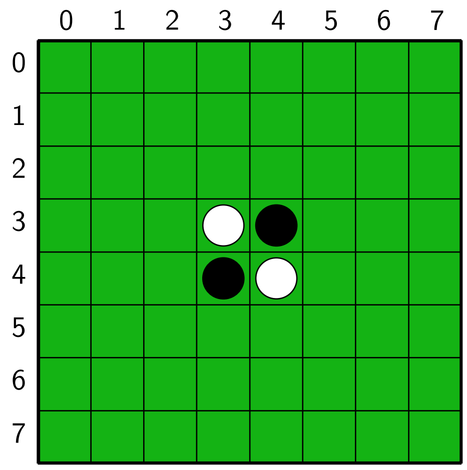

Figure 1

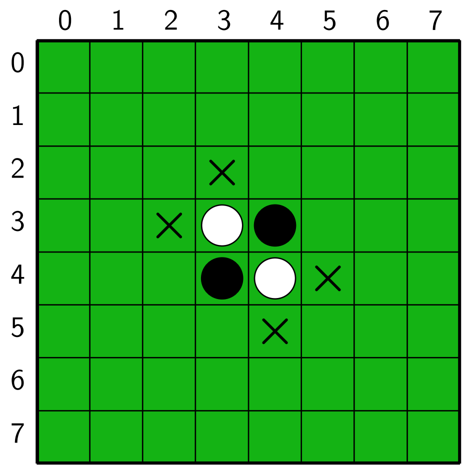

Figure 2

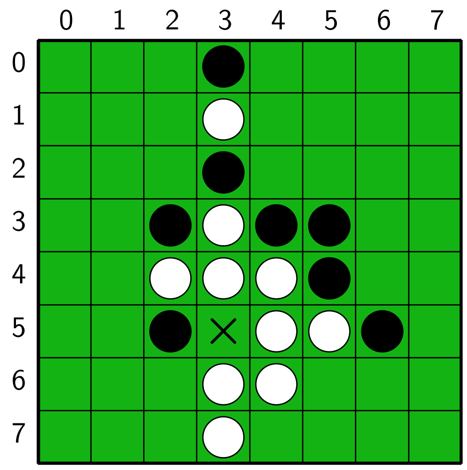

Figure 3

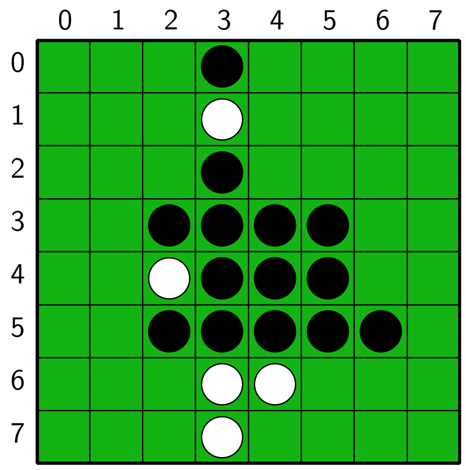

Figure 4

The objective is for each player to cause as many pieces of their opponent's color to be trapped and flipped during play as possible, so their opponent will have fewer pieces at the end. There are various strategies that come up in the game, which we don't need to worry about for our purposes. It's a fun game, and harder than it looks, because many pieces can change color every move, and pieces can be flipped many times during a game, so if this project makes you curious you can try playing it– after finishing the project, of course!

3 Description of methods to be written

All the methods described below that you must write are in the `Othello` class, which is in the `othello` package. There are three other things in that package: the GUI (a class `OthelloGUI` that is in a jarfile), a short `PlayGame` class that starts the GUI, and an enum named `Piece`. A `Piece` has three possible values–`Piece.BLACK`, `Piece.WHITE`, and `Piece.NONE`, that can be used to represent the contents of any square on the board. Although `Piece` is an enum it has a `toString()` method (that we wrote); although this wasn't shown in lecture enums in Java can have methods just like classes. This method allows "`black`" to be easily printed for a `Piece` whose value is `Piece.BLACK`, where "`BLACK`" would be printed by default if we hadn't overloaded `toString()` in the enum, and similarly for "`white`" and "`none`".

Some of the methods in the `Othello` class are present just to facilitate your (and our) testing of your methods. Ordinarily we would not allow the contents of a game board square to be arbitrarily changed; what it contains would be changed only when a valid move affecting it is made. However, we want to easily be able to construct board configurations that could require a number of moves to reach if the game was actually being played, so you and we can ensure that your methods are working correctly in different circumstances that may arise. Similarly some of the information about a game would be private and encapsulated in the game, but you will write a few methods that return some of this information just to make writing tests easier.

Some of the methods are specified to throw exceptions in certain cases. It doesn't matter what string or message these exception are constructed with, or even if they have a message string at all.

An `Othello` object will store the board contents for a game at some point during play, and probably some other necessary information. You **must** use a two–dimensional array (**not** an `ArrayList`) of `Piece` values in the `Othello` class to store the contents of the game board, since a major purpose of the project is to use two–dimensional arrays.

3.1 `Othello()`

This constructor will create a new `Othello` object, containing as discussed above a two–dimensional array (**not** an `ArrayList`) of `Piece` values. Although if this was a real program a new board would have four pieces on it in the initial configuration discussed in Section 1 above, your constructor will just create an object containing an empty board with no pieces. This will allow more easily creating boards with contents that will test specific situations. An `Othello` object must keep track at all times of the player whose turn is next; this method should cause the current object's next turn to be set to `Piece.BLACK`, meaning the next turn is to be made by the black player.

3.2 `public Othello(Othello otherObj)`

This copy constructor will create a new `Othello` object whose contents will be exactly the same as its parameter object's. This includes all contents of the board, as well as whose turn it is. After this method returns, the newly–created object and the parameter object `otherObj` must be completely **independent**, so that subsequently changing either one of them will have no effect on the other one.

3.3 `void restart(Piece color) throws IllegalArgumentException`

This method will reset an `Othello` object to the initial state given in Section 1, namely with white pieces in the squares with numbers or positions (3,3) and (4,4) and black pieces in squares (3,4) and (4,3). This method may be called at any point during a game, perhaps simulating that a player has resigned and wants to start a new game, and it should clear any current contents of the board and reset it to this initial state.

This method's parameter `color` will specify the player whose turn will be next in the reset object. If its value is `Piece.BLACK` then the next turn is to be made by the black player, while if it is `Piece.WHITE` then white makes the next move. If this was a real program, when a game is restarted it would always be black's turn, but this parameter just facilitates creating test cases with various configurations.

If the value of `color` is `Piece.NONE` this method should not change anything, and just throw an `IllegalArgumentException`; this is one of Java's library exceptions that we are using rather than defining one of our own; it is conventionally thrown to indicate that a method has been passed an illegal or inappropriate argument.

3.4 `void setTurn(Piece color) throws IllegalArgumentException`

This method simply sets the next player's turn to `color`. Ordinarily, if this was a real program, the turn would be changed only after each valid move, and code outside the class would not be able to arbitrarily modify it, but this is just for purposes of testing.

If the value of `color` is `Piece.NONE` the method should not change anything, and instead throw an `IllegalArgumentException`.

3.5 `Piece getTurn()`

This method simply returns the `Piece` value (`Piece.BLACK` or `Piece.WHITE`) of the player whose turn is next. For instance, if called immediately after `restart()` is called on an `Othello` object it would return whatever value was passed into `restart()`'s parameter `color`.

3.6 `void setEntry(Piece color, int row, int col) throws NoSuchElementException`

This method simply places a piece of color `color` in the square at `row` row and column `col` of the current object's board, regardless of the current contents of that square or any other square, and ignoring whose turn it is. If that square had a piece before this it is just overwritten or replaced. If this was a real program the contents of squares would be changed only by valid moves, and code outside the class would not be able to arbitrarily place pieces in squares, but this is just for purposes of testing.

If the values of `row` or `col`, or both, are invalid, meaning out of range for the cells of the board as defined above (both being associated with values between 0 and 7) this method should not change anything, and throw a `NoSuchElementException`; this is one of Java's library exceptions that is actually used to indicate that there are no more elements in an enumeration by the `nextElement()` method, but it seems appropriate and we prefer to reuse it than defining our own exception class.

3.7 `Piece getEntry(int row, int col) throws NoSuchElementException`

This method simply returns the contents of the square at `row` row and column `col`; the result will either be `Piece.NONE`, `Piece.BLACK`, or `Piece.WHITE`. If this was a well–designed real program the contents of squares would probably not be able to be arbitrarily accessed from outside the class but this is just to facilitate testing.

If the values of `row` or `col`, or both, are invalid this method should not change anything, and throw an `NoSuchElementException`.

3.8 `boolean validMove(Piece color, int row, int col)`

This very difficult and complex method, and the next one (which has two overloaded versions) are really the heart of the Othello game. This method will determine whether a piece of color `color` can be placed at the square indicated by row `row` and column `col`, given the contents of the board when it is called. If the values of either `row` or `col`, or both, are invalid the method will just return false (it will **not** throw an exception), and it will also return false if the indicated square is currently occupied, or if no pieces of the opposite color would be trapped and flipped by placing a piece of color `color` there. Otherwise it will return true. Determining whether at least one opposing piece would be trapped requires examining the cells of the current object's board in every direction from the square indicated by `row` and `col`, meaning both horizontal directions, both vertical directions, and all four diagonal directions, which is trickier than it sounds. You are encouraged to write code in this method first just for checking one direction, such as horizontally to the right and writing tests for that; once you're sure it works it'll be easier to adapt it for the other directions.

Note that this method doesn't actually make the move (doesn't place a piece in the indicated square); it doesn't actually change anything at all. It just returns a true or false value.

3.9 `void move(int row, int col)`

This method will place a piece for the player whose turn is next (which the current object will be storing) at the square on its board indicated by row `row` and column `col`, **if** that is a valid move, meaning if the values of `row` and `col` are valid, and at least one opponent's piece will be trapped by the move (i.e., the same conditions tested by `validMove()` described above). If the move is invalid the method will simply do nothing– the turn is not changed, and the invalid move is simply ignored. If the move is valid the method will place a piece for the player whose turn is next at the indicated location, flip all the trapped pieces in every direction (which is trickier than it sounds), and change the current object's turn to indicate that it's now the turn of the player with the other color.

3.10 `void move(Piece color, int row, int col)`

This method is very similar to the prior one, and is just for the purpose of more easily creating tests. It will do exactly the same thing as the previous version, except it will place a piece on the current object's board of color `color` at the square indicated by row `row` and column `col`, under the conditions described above. The only additional condition it must check for is whether the value of `color` is that of the player whose turn is next (which the current object will be storing)– if not the move is simply ignored. Other than these two differences it should be identical to the version above.

3.11 `String toString()`

The `toString()` method overrides the `toString()` method from the `Object` class, and should return a string representation of the current object. The string should be constructed as follows:

- The returned string should contain exactly 162 characters.
- Its initial eighteen characters should be (blank spaces are shown using ␣) `␣␣0␣1␣2␣3␣4␣5␣6␣7` (note the two initial blank spaces) and a newline.
- Each group of eighteen characters following that in the string should represent the contents of one row in the board, and there should be eight such groups of eighteen characters, representing the contents of the board's rows from the top of the board (first or zero'th row) on down. The eighteenth character of each group of eighteen characters must be a newline.
- Prior to the fifteen characters representing the contents for each row two characters should appear, a digit character (between 0 and 7) indicating the row, followed by a single blank space.
- For each square not containing a piece a single dash character should be added to the string. For each square containing a white piece an uppercase `'W'` should appear, while a `'B'` should be added for each square with a black piece.
- A single blank space should separate each pair of characters indicating the board's contents.
- As mentioned, immediately following the character indicating the contents of the rightmost column in each row a newline character should be added. **No blank space** should appear before the newline. The returned string must contain exactly nine newlines.

  (**Note**: your method cannot add a blank space after every row's characters and then remove the last one for each row; it should instead just not add a blank space after the last column's character.)

It would be difficult to show an example string with 162 characters here, but some of the public tests will illustrate what the returned string must look like.

As Section 5 says, the **only** `String` class method or operation that you can use in writing this method is string concatenation.

A real program to play Othello would likely not contain a `toString()` method, as the other methods would provide the only access to the state of an object necessary. However, being able to obtain a string representation of an `Othello` object will greatly shorten both our tests and yours.

3.12 `int count(Piece color)`

This method will return the number of squares in the current object's board that are occupied by `color` (which may be any of the values `Piece.BLACK`, `Piece.WHITE`, or `Piece.NONE`). The return value will necessarily be between 0 and 64 inclusive.

3.13 `boolean moreBlackPieces()`

This method will return true if there are more black pieces than white pieces on the board in the current object's board, and false otherwise. Note that the game may not be over when the method is called, and some (or all) of the squares may be empty. Also note that this method will not determine whether either player has won the game.

3.14 `boolean moreWhitePieces()`

This method will return true if there are more white pieces than black pieces on the board in the current object's board, and false otherwise. Note that the game may not be over when the method is called, and some or all of the squares may be empty. This method does not determine whether either player has won the game.

3.15 `boolean equalPieces()`

This method will return true if there are exactly the same number of black pieces as white pieces on the board in the current object's board, and false otherwise. The game may not be over when the method is called.

4 More about the GUI interface

The GUI interface is being supplied as class files contained in a Java jarfile; class files are provided, rather than Java source code, because the code is a bit complex and uses a variety of concepts we haven't covered. The project is set up so the jarfile contents will be seen by Eclipse, but you must be using at least Java version 1.6 (also called Java 6) in order to use the GUI.

The GUI is run by simply opening the class `PlayGame` in Eclipse, and choosing **Run → Run as → Java Application**. The GUI will pop up and start the game using the initial configuration described in Section 1. You enter a move by simply clicking on a square, so you must click on a black square first. Invalid moves are just ignored– but note that the GUI determines what moves are valid and invalid by calling your `validMove()` method (and it also makes moves by calling your `move()` method, determines who wins by calling your other methods, etc.), so if your methods have bugs the GUI won't work right. You enter moves for both players, alternating between them; a message in the bottom row tells you which player's turn it is. There is no mechanism to back up or cancel moves; the GUI is not intended to be a completely realistic implementation of Othello, but rather just an aid to allow you to easily test your code without having to write long JUnit tests that exercise making moves. If the game ends for any reason the GUI will print a message in its bottom row indicating which player won or whether the game was tied. There is no reset button, so just kill the window with the game, or click on the little red box in Eclipse that stops a running program, and rerun it if you want to start again.

Note that the GUI will fail if any methods that it's calling throw an `UnsupportedOperationException`, as they do in the initial code distribution.

5 Project requirements and related issues

1. Other than the exception classes described above, and the `String` class, you **may not use any library collection classes** (meaning **do not use** the `ArrayList` or `Arrays` classes), because the point of the project is to use two–dimensional arrays. The project can be written without using any Java library classes other than the two exceptions and `String`.

You may declare a `String` object in the `toString()` method, and perform string concatenation on it, but **no other** `String` class methods may be used.

2. You may not add anything public to the `Othello` class, and you may not add any other packages or classes. Do **not** modify the `Piece` enum.

3. Of course getting the project to work correctly is the first priority, but when you finish you should try to avoid duplication in your code. If you find that multiple methods contain the same or similar code you should try to turn it into a private helper method.

4. Grades for this project will be based on:

public tests	50 points
secret tests	30 points
student tests	10 points
source code and programming style	10 points

5. As in the first homework, all public and secret tests of your project will be JUnit tests. Be sure you write thorough tests for each method **as soon as you write it**, and ensure that it works correctly before going on! Do not wait until you're finished coding to try testing your methods, and do not rely on the public tests to thoroughly test your code!

   Note that you can debug JUnit tests, and JUnit tests can also produce output (if you want to add any print statements to them, for example).

6. As in Homework #1, your student tests **must** be in a Java class file named `StudentTests.java`, which **must** be in the `tests` package.

   Even for methods that have been tested in public tests you should still write your own student tests. The public tests are (intentionally) not exhaustive, and do not test methods in all the situations they need to work in. Your code may still have problems (that could result in losing credit on secret tests) even if it passes all the public tests.

7. If you have a bug in your code that you can't figure out, and need to come to office hours, you **must** have written at least one student test that illustrates the problem, otherwise we will not be able to assist you until you have written one or more such tests.

   If you have a bug requiring office hours assistance you must also be prepared to show the TAs how you tried debugging your code using Eclipse's debugger, and what results you found in the process.

8. If you ever get a popup message from Eclipse saying that automatic CVS operations are being disabled, you **must** restart Eclipse immediately. Failure to do so could cause problems with your project. We cannot make any allowances in grading for problems caused by your failure to restart Eclipse in this situation. If you ever have this problem and can't get the message to go away, even by restarting Eclipse, come to the instructor's office hours rather than ignoring the problem.

9. The course project grading policy document describes how projects will be graded, and in particular the criteria that will be used for grading your programming style. Be sure to read this information **carefully**.

10. The project grading policy handout describes the late policy for projects and homeworks. **Do not wait** until the last minute to finish and submit your project. It is strongly suggested that you finish and submit the project **at least** one day before the deadline, to allow time to reread the assignment and ensure you have not missed anything that could cause you to lose credit. If you have, this will give you time to correct it. This will also give you time to get any last–minute unexpected problems fixed.

11. The submission procedure will be the same as described in Homework #1.

    Note: do not use the submit server's web submission procedure to submit a jarfile or zip archive. If you have trouble submitting assignments using the procedure described in the first homework you should get the problem resolved so you can use the procedure to submit directly from Eclipse.

12. After you submit you must **log into the submit server** and verify that your submission compiled and worked correctly there!

13. Recall that the course project grading policy says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project grading policy for full details.

6 Academic integrity statement

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.

The academic integrity requirements also apply to any student tests for programming assignments, which must be **your own original work**. Copying the public tests and turning them in as your student tests would be plagiarism, and sharing student tests in any way is prohibited.