CSC 335 Project 5: Reversi

Due: Monday, April 12, 2021, by 11:59pm

GitHub Classroom Link:

https://classroom.github.com/g/LVsdtf5x

Project Description

For this project (and the associated lab on branching and merging in git), we will be working in teams of two. By **2:00pm on Thursday, April 1, 2021**, everyone should have signed up for a team on this Google Spreadsheet: https://docs.google.com/spreadsheets/d/1-0UNIu9MTNfjvYiXNMxdpUegjMNom50peVTqqnw9GQ0/edit?usp=sharing

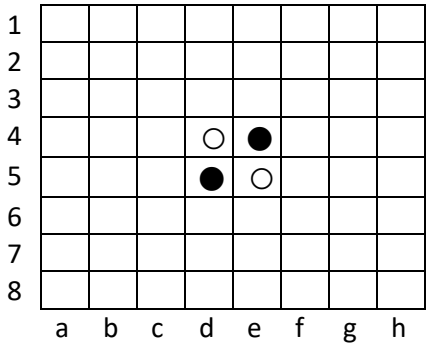
In the associated lab, form a team on GitHub. The first person to use the link above will make the team. Name it P5-NetIDfirstperson-NetIDsecondperson. The second person can then select the team from the list. The team for the lab is the same as the team for this project.

You **MUST** work with someone for the project. If you do not sign up for a team, you will be assigned randomly another person in the class as your partner. Picking someone means that you get to start sooner and can pick someone in a similar time zone if that is important to you.

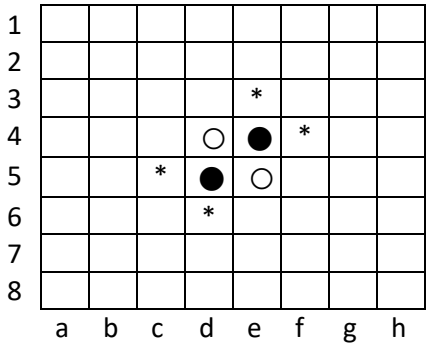
Note that this is an empty repository. Please create a new Eclipse project in the repository and use that to build and test your program.

Description

Reversi (often called by the trademarked name Othello) is two-player game played on an 8x8 grid. In the physical game, there are tokens with a different color on each side. The color white represents the first player and the color black represents the second. The board starts out with four pieces, 2 from each player in the center of the board like so:

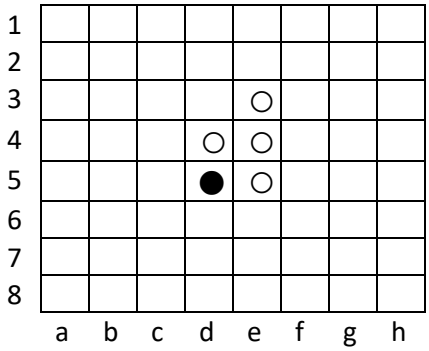


Each player takes turns placing a token with their color face up onto the board and has the requirement that their token be placed so that it forms at least one straight (horizontal, vertical, or diagonal) occupied line between the new piece and another of their colored pieces, with one or more contiguous opposite-color pieces in between. From the above starting configuration, the legal moves for white are indicated with an asterisk below:

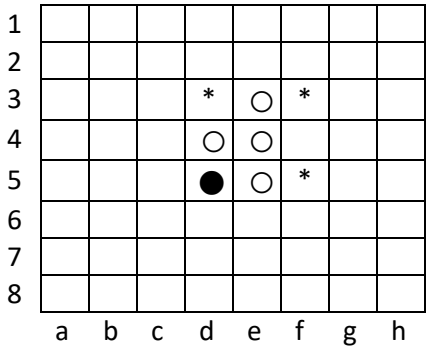


For example, if the white player places a token at e3, the new white token is in line vertically with another white piece at e5 white a black piece in between (at e4). The move d3 would not be legal, because there are no black pieces between d3 and d4, and there is no white piece at f5.

When a legal move is played, all opposite-colored pieces along the horizontal, vertical, and diagonal lines that end with the closest same-colored piece are "captured" and flipped over (in the physical game) to be your color. So, for our move e3, the board would then become:



The current score is the number of pieces of each color. After this move, white has a score of 4 and black has a score of 1. It is now black's turn, and their legal moves are:



The game continues until the board is full or there are no legal moves for either player. If there is no legal move for one player, the other player plays consecutive turns, until the game completes.

The winner is determined by which player has more tokens at the end of the game. A tie is possible.

For more information on the game, you can see https://en.wikipedia.org/wiki/Reversi

The board will be 8 rows high by 8 columns wide.

The computer will be a (relatively) weak player, whose choice of move is to pick from the legal moves the one that maximizes their score (considering only the current move). If there is more than one such move, simply select one of the best moves randomly.

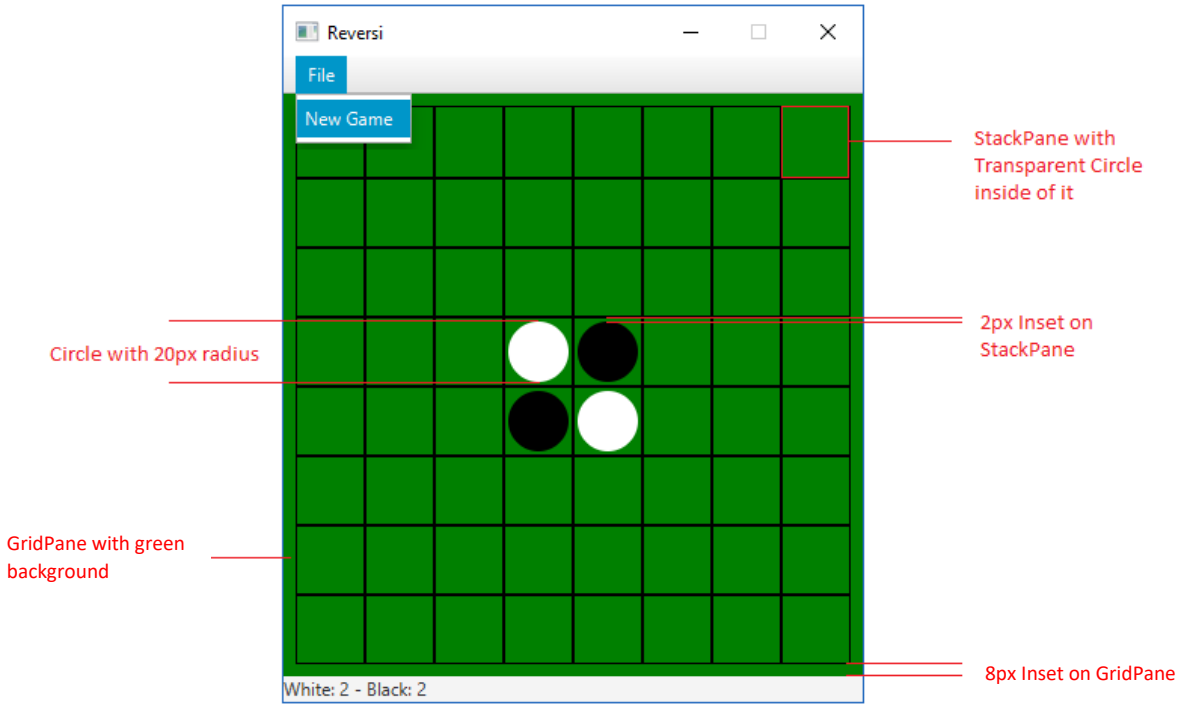
Implementation

Your job for this project is to implement the game of Reversi using all of the tools we have learned so far. That means you should follow the MVC design, use UML to plan and document your class structure, provide full source code documentation using `javadoc`, a complete test suite that tests your model and controller to 100% branch coverage, and, of course, works.

GUI

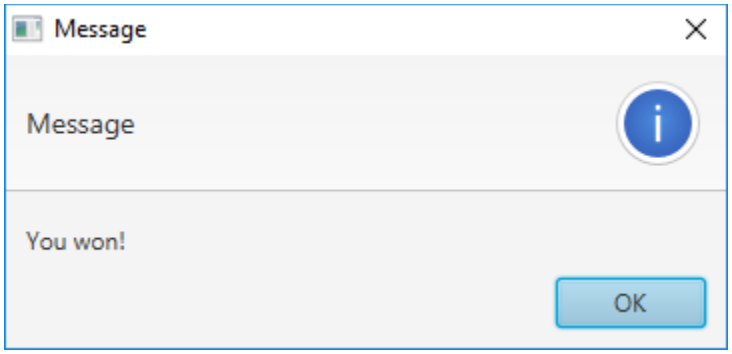
We will construct a new `ReversiView` that is a `javafx.application.Application` and use a `GridPane` to hold `javafx.scene.shape.Circle` objects representing our empty spaces (transparent) or our player and computer tokens (white and black).

Lay out the main Stage with the following constraints:



As the user and computer play, we will want to switch the appropriate circles to white and black:

When you win or lose, display a modal (showAndWait()) Alert:



If the user clicks in a square that is not a legal move, ignore the click.

After the game is over, do not allow any further moves.

Implementation

Main Board

As mentioned above, your game board will be a `GridPane` of 8 rows and 8 columns. The Inset will be 8 pixels. Each element of the board will be a `StackPane` which contains a `Circle` shape with a 20px radius and initially a transparent fill color. The background of the `GridPane` will be filled with green. Each `StackPane` will have a thin black border around it.

Beneath the `TilePane`, use a `Label` to display the current score.

Event Handling

You will add a `MouseClick` handler to the pane. You can click anywhere to place your piece. That means you'll need to get the X and Y coordinate of the click event and turn it into a row and column number. You will then call into your controller to make the desired move.

MVC and Observer/Observable

When the controller changes the model, we'd like to change the view. The model can do this for us, by notifying us when the model changes if our view is an Observer and the model is an Observable.

Have your model class extend `java.util.Observable` and have your view class implement `java.util.Observer`. As a requirement to the `Observer` interface, you will need to implement the `update(Observable o, Object arg)` method in the View.

Make a `ReversiBoard` class to encapsulate the board independent of the underlying Model. Construct an instance of this class in the model when a move is made and pass that to `update()` via the `Object arg` parameter. That argument is set by the Observable model calling `setChanged();` and `notifyObservers(Object arg)`. Whatever we pass to `notifyObservers()` will become the `arg` parameter of `update()`.

Note: As of Java 9, Observer and Observable have been deprecated. The reasons for this deprecation aren't germane to us, and there is no real "better choice". So we will stick with it as it is useful. If you're

using Java 10 or later and want to have Eclipse stop yelling at you, you can use `@SuppressWarnings("deprecation")` above your Model class.

Saving and Loading

The `ReversiBoard` class (and only the `ReversiBoard` class) should be Serializable.

When the program loads, look for a file named "save_game.dat". If it exists, load it into your model and show its state on the view. If it doesn't exist, assume it is a new game and construct a new default `ReversiModel`.

When the windows closes (`setOnCloseRequest`), handle the `WindowClosing` event and save the current game to "save_game.dat" by writing out the serialized `ReversiBoard` class. Do NOT use anything other than `ObjectInputStream` and `ObjectOutputStream` to implement loading and saving.

Add a menu (pictured above) to the top pane of the application. Your `MenuBar` should contain a "File" `Menu` with one `MenuItem` for "New Game". When an action is performed on the `MenuItem`, construct a new model, update the view, and delete the "save_game.dat" file if it exists.

When the game is won or lost, delete "save_game.dat" if it exists.

Requirements

- A two-member team on the spreadseet by class time **2:00pm on Thursday, April 1, 2021**

As part of your submission, you must provide:

- At least five classes, Reversi, ReversiView, ReversiController, ReversiModel, and ReversiBoard. The Reversi class will have `main`, separating it from the view. You may have as many additional classes as you need.
- Complete javadoc for every class and method, using the `@author`, `@param`, `@return`, and `@throws` javadoc tags. Generate your documentation into a doc folder.
- A complete UML diagram for your design, drawn using http://draw.io and the xml file committed as part of your repository
- Test cases for your model and controller with 100% branch coverage
- A main class, that launches your view using :

```
Application.launch(ReversiView.class, args);
```

- A ReversiView class that uses JavaFX to display the GUI described above and is an Observer
- A ReversiModel class that extends Observable and notifies its observers when the model changes (a move is made)
- A ReversiBoard class that encapsulates the board from the model to pass to the view and is Serializable
- A menu for starting a new game
- The ability to save the game when an in-progress window is exited
- The automatic loading of a saved game on the next run
- At least two feature branches, and a commit history that shows each team member's contributions.
- A master branch that contains your final submission

- As always, reasonable comments and style should be followed, with test cases to convince yourself that your modifications work

Your code must follow the MVC architecture as we have described it in class. That means:

- No input or output code except in the View
- A model that represents the state of the game but guards access through public methods
- A controller that allows the view to interact indirectly with the model, providing the abstracted operations of your game
    - Including a `humanTurn(int row, int col)` method and a `computerTurn()` method that represent the turns
    - A set of methods that determine the end of game and the winner
    - Some way to access the board to be able to display it as part of the view
    - As few public methods as possible, with helper methods being private and all non-final fields being private
- Your UML diagram should be used to plan out the program and will be a Section Lab grade taken as part of your final submission

Submission

As always, the last pushed commit on the master branch prior to the due date will be graded