CSC 335 Project 5: Networked Connect 4

Due: Monday, November 15, 2021, by 11:59pm

GitHub Classroom Link: https://classroom.github.com/a/p5C4BhYl

Connect 4 is a tic-tac-toe based two-player game. In the physical game, there are two colors of tokens, yellow for the first player and red for the second. Each player takes turns dropping a token into an upright board which only allows for dropping your token on top of column (as in a stack), reducing the freedom on each turn from that of traditional tic tac toe.

As the name implies, you win when you create a connected line of 4 of your tokens, either horizontally, vertically, or diagonally. Games can end in a tie if neither player makes a line of 4 connected tokens.

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

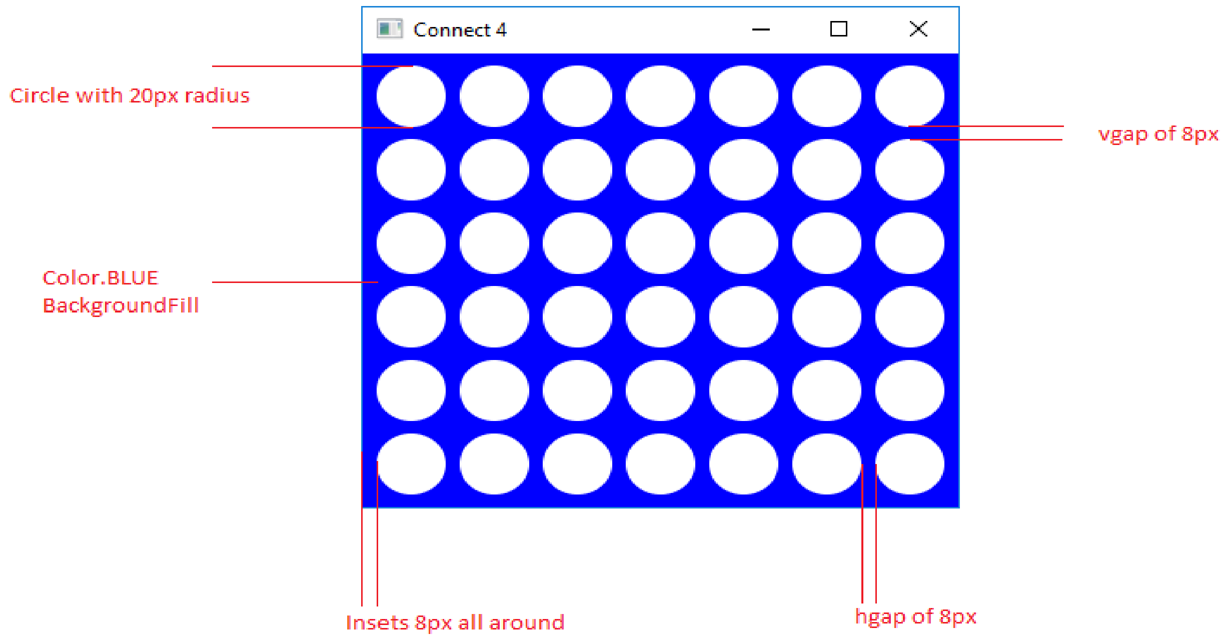
Implementation

Your job for this project is to implement the game of Connect 4 using all of the tools we have learned so far. That means you should follow the MVC design, provide full source code documentation using `javadoc`, a test suite that tests your controller to 100% statement coverage, and, of course, works.

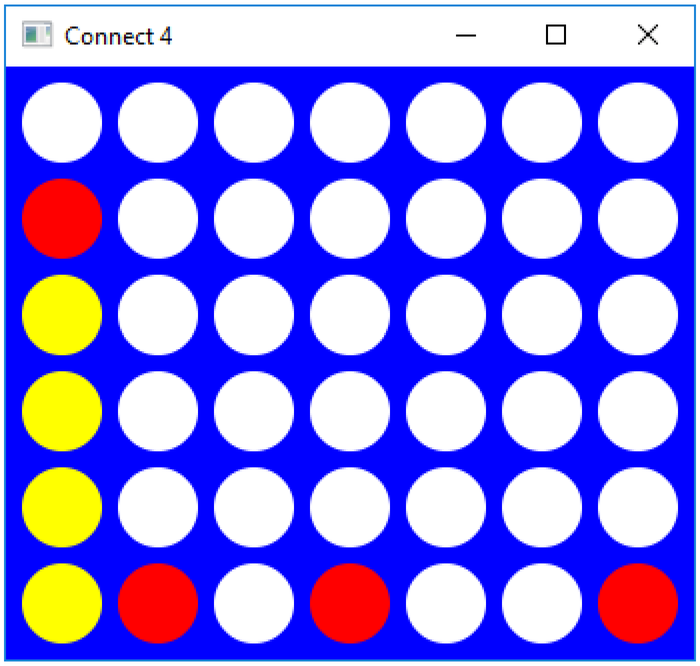
The computer "AI" will be a random player. Have it choose a legal move randomly.

We will construct a Connect4View that is a javafx.application.Application and use a GridPane to hold javafx.scene.shape.Circle objects representing our empty spaces (white) or our player and computer tokens (yellow and red).

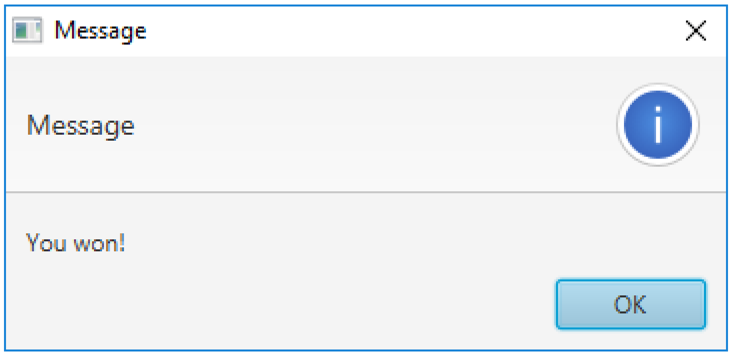
Layout the main Stage with the following constraints:



As the user and computer play, we will want to switch the appropriate circles to yellow and red:



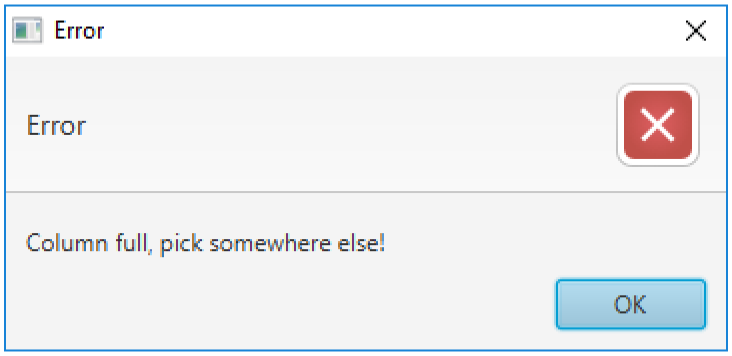
When you win or lose, display a modal (showAndWait()) Alert:



Message

You won!

If the user clicks in a full column, display this modal Alert:



Error

Column full, pick somewhere else!

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

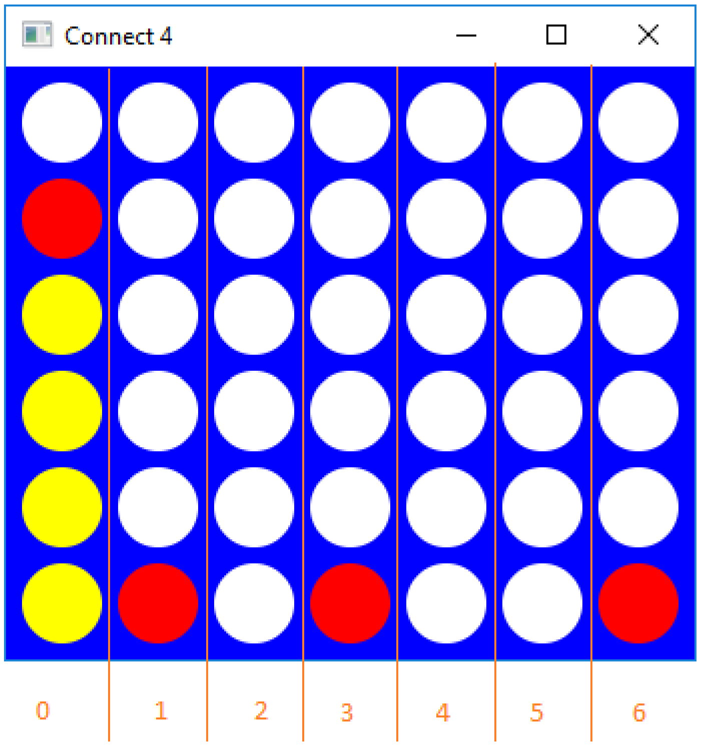
Main Board

As mentioned above, your game board will be a `GridPane` of 6 rows and 7 columns. The Inset, HGap, and Vgaps will all be 8 pixels. Each element of the board will be a Circle shape with a 20px radius and initially a white fill color. The background of the `GridPane` will be filled with blue.

Event Handling

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

The regions of the board are shown below. The dividing line is directly between the columns, halfway into the 8 pixel HGap.



MVC and Observer/Observable

When your view is notified to be redrawn (the model has changed), it should change the color of **ONLY** the disk that represents the most recent move. You can learn which disk to update by passing the row, column, and color via the `Object arg` parameter. This can be set by the model using the parameterized overload of `notifyObservers(Object arg)`. Whatever we pass to `notifyObservers()` will become the `arg` parameter of `update()`.

Make this argument an instance of `Connect4MoveMessage` class encapsulates the row, column, and color information of the move:

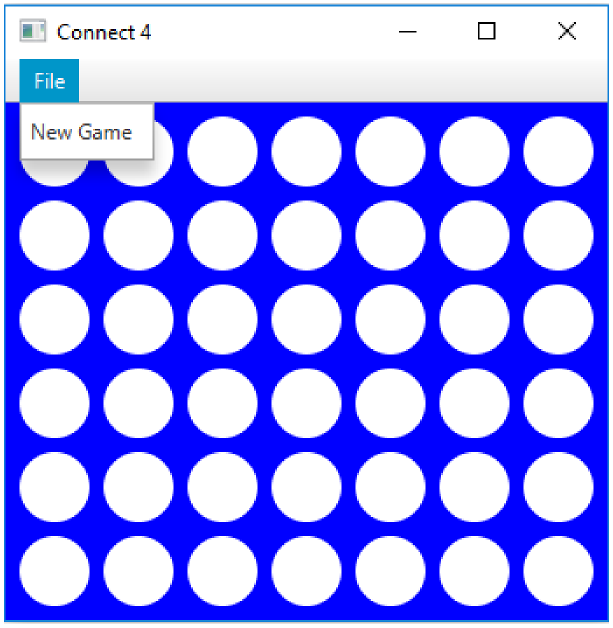
```
public class Connect4MoveMessage implements Serializable {
	public static int YELLOW = 1;
	public static int RED = 2;

	private static final long serialVersionUID = 1L;

	private int row;
	private int col;
	private int color;

	public Connect4MoveMessage(int row, int col, int color) {
		this.row = row;
		this.col = col;
		this.color = color;
	}

	public int getRow() { return row; }
	public int getColumn() { return col; }
	public int getColor() { return color; }
}
```

Construct an instance of this class in the model when a move is made, and use that information in the view when `update()` is called to change only the one circle to be the proper color.

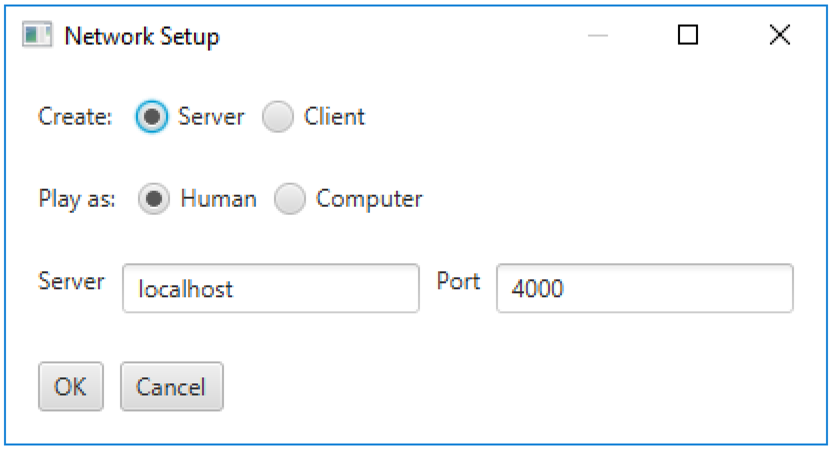
You may **NOT** add any additional fields to this class. It is important that this message be the same for every team, as we will be using this to grade.

Establishing Client and Server Roles

Add a Menu and MenuItem of “New Game”:



That when selected, produces this as a dialog box:


Network Setup

Create: [✓ Server] Client

Play as: [✓ Human] Computer

Server: localhost Port: 4000

For this dialog, create a class which extends Stage. Make it `initModality(Modality.APPLICATION_MODAL)` so that when you showAndWait() the instance, it acts as a modal dialog. Allow the user to select which role they will handle and whether they will be a human player or if your AI will be playing. When the user clicks OK or Cancel, have the class provide methods to query the object for what the user selected.

Default the server to localhost and the port to 4000.

Network Protocol

When the Server is created, have it accept() a client connection. The Server will always take the first turn. If it is a human player, the player will click and send the event to the client. Otherwise the random AI will generate its turn and send it to the client. The client will go second. This will repeat until the game is over.

Rather than send the model each turn, we will use the `Connect4MoveMessage`.

Networking Hints

Note that when we are waiting for the other player's turn, if we read from our InputStream, we will block the main thread, which will cause the GUI to become unresponsive to user input since we are preventing our events from being pulled from the event queue.

To resolve this, do two things:

1. Do not block the main thread that handles the GUI, spawn a new, temporary thread to block waiting for the other player's `Connect4MoveMessage` to be sent
2. Disable the UI via some boolean that simply prevents your click handler from letting the user take two turns in a row.

When you finally receive the other player's `Connect4MoveMessage`, you'll need to negate the boolean and update the UI. You can't do these things from the thread you spawned, only from the main thread, or it will not work properly.

To solve this, use `Platform.runLater` (https://docs.oracle.com/javase/8/javafx/api/javafx/application/Platform.html#runLater-java.lang.Runnable-) . This takes a block of code (either a lambda or a class that implements Runnable with a run method) and does not create a new thread, but rather adds an event to the event queue whose handler will simply execute the block of code in the lambda/run method you provided. It will do this on the main thread, allowing you to update the GUI and not freeze the UI.

Requirements

As part of your submission, you must provide:

- At least five classes, Connect4, Connect4View, Connect4Controller, Connect4Model, and Connect4MoveMessage. The Connect4 class will have `main`, separating it from the view. You may have as many additional classes as you need.
- A main class, Connect4, that launches your view using :

```
        Application.launch(Connect4View.class, args);
```

- A Connect4View class that uses JavaFX to display the GUI described above and is an Observer
- Complete javadoc for every class and method, using the `@author`, `@param`, `@return`, and `@throws` javadoc tags. Generate your documentation into a docs folder.
- Test cases for your controller with 100% statement coverage

Your code must follow the MVC architecture as we have described it. 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 col)` method and a `computerTurn()` method that represent the turns
  - A set of methods that determine the end of game and the winner
  - As few public methods as possible, with helper methods being private and all non-final fields being private

For the networking portion:

- You must communicate using only Connect4MoveMessages as described above
- Your program must be able to act as a server or as a client with the same main() and UI.
  - This means to test, you should be launching two instances of the game and having them connect to each other over the network.
- Your program should allow either the random AI or the user to play
- This means it should be possible for the AI to play against itself
- At least *two* feature branches, and a commit history that shows each team member's contributions.
- Make sure your submission contains a master branch with your final submission

Submission

Make sure to periodically commit and push your changes to github.

We will grade the last commit on the master branch that was pushed prior to the deadline.