Due
July 15, 2016, before 11:59 pm

Notes
- Please follow both the "Programming Standards" and "Assignment Guidelines" for all work you submit. Programming standards 1-25 are in effect.
- Hand-in will be via the UMLearn Dropbox facility. Make sure you **leave enough time** before the deadline to ensure your hand-in works properly. The assignments are submitted using UMLearn's time (not the time on your computer). Assignments that are late will be marked according to the late policy in the ROASS.
- All assignments in this course carry an equal weight.

Assignment 3: Connect

Description
In this assignment, you will write Java classes to enable a user to play a game of "Connect", where players alternate dropping coloured playing pieces into a grid. (The game is sold under the copyrighted name "Connect 4".) The goal of the game is to get four of your pieces in a line (horizontal, diagonal or vertical) before your opponent does. Here's a video that describes the game in 19 seconds: https://youtu.be/ylZBRUJi3UQ. In your game, you will use a standard 7x6 board.

In your game, a human player will play against the computer. The human player always plays first. The human player will be able to play again after a match is finished and play as many matches as they want.

You will be responsible for implementing the game logic (controlling turns, determining if someone has one the game, etc.) as well as two or more game AIs (automated players) to play against the human.

Details
You are responsible for designing the game logic and the game AI for Connect. The game logic is responsible for maintaining the current status of the board (where each players' pieces are) as well as determining if a player has won. On the other hand, displaying the board and prompting the human player for input is the responsibility of the GUI. You are not responsible for developing the GUI: it will be given to you.

You are responsible for implementing one or more different game AIs (automated players). How the AIs work is up to you, but you should ideally develop multiple different approaches to playing the game. You don't need to program an AI that wins all the time but your AIs should at least be ranked (in order of increasing difficulty) and the more sophisticated approaches should involve some logical move choice by the game AI. The reward for developing a very sophisticated game AI is the possible admiration of your classmates, and veritable minutes of entertaining game play.

Bonus Marks (15%)

In this assignment, you are required to implement at lest one game AI. Bonus marks will be given if you develop at least one additional game AI and allow the user to select the chosen difficulty (using the `promptForOpponentDifficulty` method described below). The second AI must show an increased level of difficulty over your basic AI.

Game Logic

The logic of your game AI should proceed roughly as follows:

- The game starts, and the human player is prompted for the difficulty of opponent they want.
- The human player plays first, and turns alternate between the human player and the game AI.
- If one player wins, the match stops and the winner is announced. The match may also end in a draw if the board is completely filled.
- After the match ends, the human player is then prompted to see if they want another match. The board is reset and the human is prompted for the difficulty of the opponent they want.

The tools for accomplishing all of these tasks are described below. The three major parts of the project are the Game GUI, the Game Logic and the AI. The functioning of each of these three parts is dictated by three interfaces.

Game GUI: There is an interface for the game display and for getting user input. The GUI will satisfy the GameDisplay interface:

```
public interface GameDisplay {
     public void gameOver(Status PlayerNumber);
     public void updateBoard(Status[][] board);
     public int promptForOpponentDifficulty(int maxDifficulty);
}
```

Your game logic class will call these methods. The methods work as follows:

- `gameOver(Status PlayerNumber)`: this method should be called to inform the GUI that the game is over. If either player wins, the Status should reflect that, while if the game is a draw, the Status should be "neither".
- `updateBoard(Status[][] board)`: this method should be called whenever a new move has been made and the board needs to be updated. Each position in the grid should be given by a status (i.e., which player occupies the position, or "neither" if the spot is empty). Note that the GUI behavior is **undefined** when an updateBoard call is made with a nonsensical board (i.e., one with no gravity, wrong number of player tokens, etc.).
- `promptForOpponentDifficulty(int maxDifficulty)`: this method should be called once at the beginning of each match. The method will be given the maximum difficulty of the AI (for example if you have two AI levels, the argument

will be a `2`). The method will return a value between 1 and maxDifficulty, indicating the user's choice of how hard the game AI should be. Your controller will need to call this method **only** if you choose to attempt for the bonus marks, and implement additional AI units with varying difficulty.

See below for details of the Status enumerated type.

Game Logic: You are responsible for implementing the back end that manages the game logic. Your game logic will call the methods of the GameDisplay interface. It must also satisfy the ConnectController interface:

```
public interface ConnectController {
     public boolean addPiece(int col);
     public void reset();
}
```

The methods work as follows:

- `addPiece(int col)`: the method informs the controller that a piece has been played in the column 'col'. This represents the move by the human player. Note that the AI will **NOT** call this method (see below for the method makeMove(), which asks the AI which move they want to make). Note also that the method does not tell the game logic **where** the piece ultimately rests in the column: that's the responsibility of the Game Logic to determine.
- The method should return false only when the move is invalid (i.e., when col is already full). If the method returns false, the board should be unaffected by the call (i.e., if a bad move is made, the board remains unchanged).
- `reset()`: this method resets the game. It will be called when the player requests a new game.

While the game logic will call methods from the GameDisplay interface, the GUI (provided) will call methods from the ConnectController interface.

Game AI: You are also responsible for developing one or more game AIs (see option for bonus marks, above). These AIs must satisfy the ConnectPlayer interface:

```
public interface ConnectPlayer {
     public int makeMove(int lastCol);
}
```

The method makeMove should take the last move made by the (human) player and return the next move made by the AI. The method will only be called when a play is possible (i.e., there is at least one valid move the AI can make) and the AI must make a valid play. Your AI does not need to worry about what it should do if there's no valid moves. Since the AI always plays second, the lastCol value will always be valid.

Note that the method only supplies the ***last*** move made by the human player, not all previous plays or the current state of the board. If the AI wants to save the previous moves, or the current state, it will have to do so on its own.

Linking Classes and Starting the Game: To link all classes, you need two things:

1. Your Game Logic class should have a constructor that accepts a single parameter of type `GameDisplay` (so that it can call the `GameDisplay` interface's methods, like `gameOver`, etc.)

2. You need to implement a Factory class, so that the `GameDisplay` does not need to know the name of your class that implements the `ConnectController` interface. Create this class:

```
public class ControllerFactory {
     public static ConnectController makeController(GameDisplay
     gd) {
          return new XXX(gd);
     }
}
```

Here `XXX` is the name of your class that implements the `ConnectController` interface. This name is up to you. Note however, that this Factory class **MUST** be called `ControllerFactory` and the method **MUST** be called `makeController`, otherwise your code will not work.

To start the game, create a class with a main method. The main method simply needs to call the constructor of the provided GUI class.

```
public class Main {
     public static void main (String[] args) {
          new YYY();
     }
}
```

Here YYY is the name of the class that I will provide to you for the GUI.

Status Enum: Statuses in the game will be represented by the Status Enum:

```
public enum Status {
     ONE, TWO, NEITHER;
}
```

This value could represent which player occupies a space on the board (i.e., in `updateBoard`), or who has won a game (i.e., in `gameOver`).

Data Structures

In this assignment **ONLY** you are permitted to use a 2D array**.** No linked structures are likely to be necessary in the game logic. However, any data structures constructed elsewhere in the game, other than 2D arrays to store the board, must be implemented by you.

Hand-in

Submit all your source code for all classes you implement. You do not need to provide classes and enumerated types that we have provided (GUI, Status). You also do not need to submit the interfaces that are described in the assignment. They will be added to your code when compiling and running your submission.

You should also submit a README.TXT file that describes exactly how to compile and run your code from the command line. The markers will be using these instructions exactly and if your code does not compile directly, you will lose marks.

You will be given a preliminary (text-based) interface for programming. You will be given the GUI interface a few days before the deadline. The markers will be testing your code with the GUI.

You **MUST** submit all of your files in a zip file. Submit all files on UM Learn Dropbox.