# Informatics 1: Object Oriented Programming

## Assignment 1 - Connect Four

**The University of Edinburgh**
David Symons (d.symons@ed.ac.uk)

## Overview

This assignment is about practising the programming concepts taught during the first few weeks of the course. To hone and demonstrate your skills, you will complete a series of steps to implement the board game *Connect Four* (https://en.wikipedia.org/wiki/Connect_Four) in Java. A program structure is provided with some incomplete methods for you to fill in. The following three sections describe the steps you need to complete for a basic, intermediate and advanced solution.

Marks will be awarded out of 100 and count for 25% of your overall grade for Inf1B. Completion time may vary depending on your experience and the grade you are aiming for. Of the 200 hours the course is designed to take, 10 have been allotted for this assignment. This is the estimated time for completing the basic and intermediate tasks. Marks above 69% are expected to be rare and require you to tackle the advanced section, which goes beyond the course in terms of content and time.

Before you start, please read the following sections below:

- *Restrictions*
- *Good Scholarly Practice*
- *Marking Criteria*
- *Submission, late submission & extensions*

This assignment must be **submitted on CodeGrade by 16:00 on Friday February 18th**.

## Setup

This section describes how to download the provided code, create a new project and run the game.

Download the provided code

- Go to LEARN → Assignments → Assignment 1 (where you also found these instructions).
- Download *ConnectFourStarter.zip* and unpack the *src* and *saves* folders it contains.

Create a new project

- Create a new, empty Java project called “ConnectFour” in IntelliJ (or your preferred IDE).
- Copy the *src* and *saves* folders into your project directory (replacing the default *src* folder).

Explore the packages and classes

- Do not panic! Most of the classes have already been completed for you. Providing code is the only way of letting you work on a larger, more interesting project before you’ve learned all required concepts. You do not have to understand how everything works.
- You will only be editing *Model.java* and *TextView.java* in the *game* package. Depending on how far you get, you will also be working on one or more classes in the *players* package.

Run the code

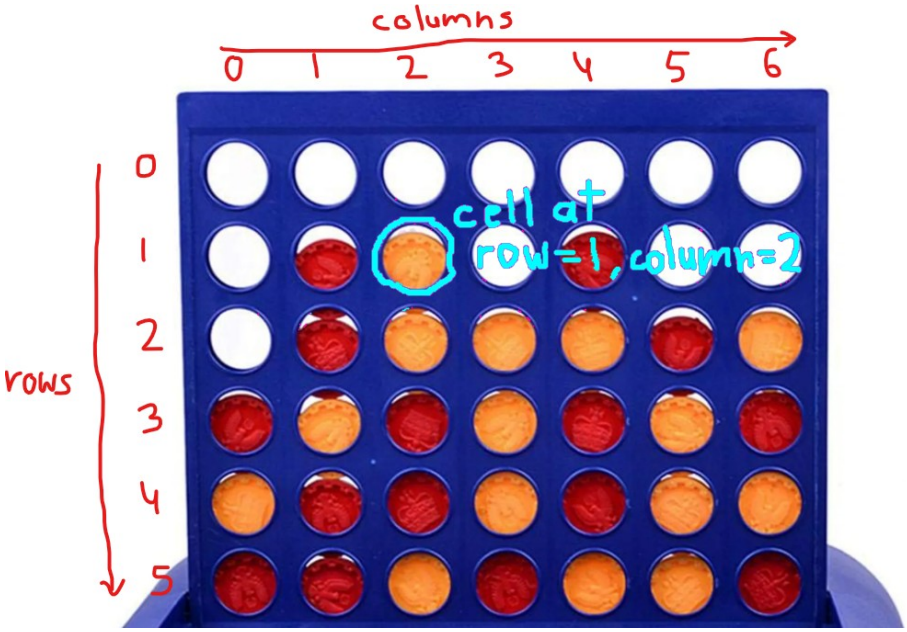
- Open *ConnectFour.java* (in the *main* package) and press your IDE’s run button. You should be greeted with a welcome message and a menu. Selecting option 1 starts a new game, but until you have completed some steps in the basic section, this will enter an infinite loop from which you have to quit manually. This animated gif shows correct setup and how to terminate the loop.

## Basic requirements

This section describes the features required for a basic implementation of Connect Four. A good solution will allow you to pass, without attempting any of the intermediate or advanced features. The suggested sequence of steps is intended to guide you towards creating a playable game.

Step 1: Model the state of the board [Relevant file: Model.java]

Think about how to represent the state of the board. Add the appropriate field(s) to *Model.java* and initialise these variables in its *initNewGame* method. This is called every time a new game is started and should create an empty board. For now, you can ignore the *settings* parameter and assume a standard board size with 6 rows and 7 columns. The top row is row 0, with row numbers increasing downwards. The leftmost column is column 0, with column numbers increasing towards the right (see illustration below). Next, implement the *getPieceIn* method to allow access to the state of the board. It takes row and column indices as its parameters and should return the owner of the piece in that cell of the board i.e. 1 for player 1 and 2 for player 2. Return 0 to indicate an empty cell.

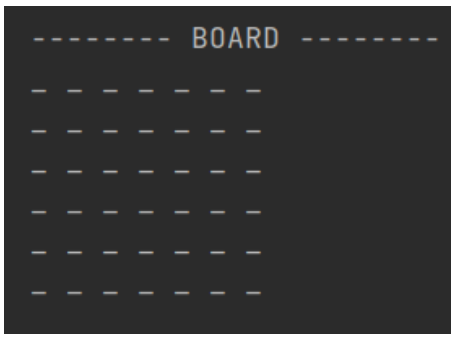



Step 2: Allow the user to select moves [Relevant file: HumanPlayer.java]

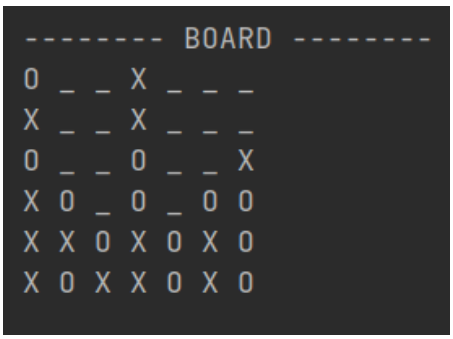
You can already select different players from the main menu (option 4), but all of these players will just concede the game on their first move. The goal is now to replace this behaviour with something useful. To allow the user to make a move, implement the *chooseMove* method of *HumanPlayer.java* (in the *players* package). This is called at the start of a player's turn and should return the column index into which the player wants to insert their next piece (or -1 if they really wish to concede). All you need to do is print out a message asking the user to enter a move and return what they typed in (use *InputUtil.readIntFromUser* for this). When you start a game, the program now waits until the user has entered a move and pressed enter. After entering a few moves, you will realise that you are still stuck in an infinite loop though. That is because we are not yet reacting to those moves. We will return to this issue a few steps later. Note that *HumanPlayer* is selected by default for both players so you do not need to change the players in the menu every time you start the program.

Step 3: Display the board [Relevant file: TextView.java]

Open *TextView.java* and complete the *displayBoard* method to print a text representation of the board to standard output. Use characters 'X' and 'O' to represent pieces belonging to players 1 and 2 respectively and use '_' (underscore) for empty slots. You can add a space between each column if you like, but please do not add any other decorations. You can test this is working by starting a new game. An empty board should be displayed, see left image below.




Empty board




Board after several moves

Step 4: Store the moves in the model [Relevant file: Model.java]
Although the model receives these moves you enter (as the parameter to its *makeMove* method), it has not yet been set up to put the pieces onto the board. As long as the moves you make are not stored in memory, the board will remain empty. Correct this now, making sure each piece is inserted in the correct row and column. Use the *getActivePlayer* method to determine the owner of the piece. You should now be able to place some pieces on the board when you play, but since player 1 starts the game and the active player is never changed, all pieces will be marked as 'X'.

Step 5: Taking turns [Relevant file: Model.java]
The model also needs to keep track of whose turn it is. Currently, its *getActivePlayer* method always returns 1. After putting the first piece on the board, player 2 should become active and *getActivePlayer* should return 2. After that it's back to player 1 and so on. Think about how to represent whose turn it is and when to switch. In particular, getters (methods that return a value) should never change anything so that you can call them as often as you like. Make sure player 1 always goes first when a new game starts. Players should now be able to take turns and, after a few moves, the state of your board should be similar to that shown in the screenshot above on the right.

Step 6: Game over [Relevant file: Model.java]
The game should end either when the board is full or when a player concedes (by entering -1 as their move). The model's *getGameStatus* method is responsible for returning a code that indicates if the game is still in progress or, if not, how it ended. A zero means the game is ongoing, a 1 means victory for player 1, a 2 victory for player 2 and 3 indicates a tie. To make programs easier to read, we often store such codes as special constants. That is why instead of returning 0, *getGameStatus* returns IModel.GAME_STATUS_ONGOING. Change this to return IModel.GAME_STATUS_TIE when the board has been filled up. If player 2 concedes, return GAME_STATUS_WIN_1 and GAME_STATUS_WIN_2 if player 1 concedes. Hint: Think about what causes the game status to change and where this should be recorded. Detecting the winner (first player to get four in a row) is an intermediate feature and not required for a basic solution.

## Intermediate features

Once you are satisfied that you have all basic features working, you can enhance your program by adding the functionality described here. This will allow you to obtain a mark of up to 69% provided that your code for all the basics is working well.

Variable game settings [Relevant files: TextView.java & Model.java]
When the user selects "Change game settings" from the menu, the *requestGameSettings* method in *TextView.java* is called. Edit this method to allow the user to enter a board height (number of rows) and width (number of columns). You should also be able to choose the required streak length, so you can play Connect X. It is important to ask for the settings in this order (rows, columns, streak). The user should press enter after typing each value (use *InputUtil.readIntFromUser* for this).
Now, make any changes required for the model to support the variable game settings. The chosen settings are passed to the *initNewGame* method and can be accessed using *settings.nrRows* etc.

Input validation [Relevant files: Model.java & TextView.java]
So far, we have trusted the user to enter only sensible values. If they do not, they can crash the program or make the game impossible to win. Input validation involves detecting and preventing such issues.
Start by changing the model's *isMoveValid* method to return *false* when the passed in move is invalid (and *true* otherwise). Players will then automatically be asked to re-enter rejected moves. The *requestGameSettings* method should repeat the request for each setting until an acceptable value has been entered. The board should have no less than 3 rows or columns and no more than 10 (see constants for this in IModel).

Implement your first AI player [Relevant file: RoundRobinPlayer.java]
Complete the *prepareForGameStart* and *chooseMove* methods in *RoundRobinPlayer.java* in order to create your first AI player. This player's first move (after a game is started/loaded) is always to play the leftmost column that is not full. The column index then increases by 1 with each time, until we reach the last column and return to column 0. A typical sequence of moves would therefore be: 0, 1, 2, 3, 4, 5, 6, 0, 1, … The only exception is when a column is full. In this case, skip ahead until a valid column is found. Loading games is an advanced feature that you do not have to implement here. However, please do not assume the board will always be empty when *prepareForGameStart* is called. Your code should be compatible with variable game settings.
To test your AI player, choose option 4 from the game menu and then select one or both players to be the RoundRobinPlayer (option 2). All combinations of players are possible, including AI vs AI.

Automatic win detection [Relevant file: Model.java]
Write code to detect when the game is won. The *getGameStatus* method in the model should then return 1 or 2 to indicate the winner. Again, this should work with variable game settings.

## Advanced features
Tasks described in this section are much more difficult and only required for a distinction.

Load saved game states from file [Relevant file: Model.java]
Implement a mechanism by which you can load a game state from a text file and resume play. This involves implementing *initSavedGame* in the model. All save files are assumed to be in the "saves" folder in the project hierarchy. The user is only prompted for the name of the file to be loaded (this has already been implemented). Each file starts with 4 lines containing one integer denoting the number of rows, columns, streak length and the active player (in that order from top to bottom). The board state follows, with one line for each row of the board (top to bottom). Each line is a string with as many characters as there are columns. Zeroes represent empty cells and filled cells are represented by 1 or 2 depending on the player who owns the piece. You can assume all save files are formatted correctly and that saved games are not yet over (there is no winner and the board is not full). See the "Save.txt" and "AnnotatedSaveFile.txt" in the "saves" folder.

Implement the WinDetectingPlayer [Relevant files: WinDetectingPlayer.java & maybe Model.java]
If the player can win on their current turn, a winning move must be selected. Otherwise, choose any valid move that does not let the opponent win next turn. If no such move exists, concede. This player must be able to cope with variable game settings and games being loaded from file.

*Implement the CompetitivePlayer* [Relevant files: CompetitivePlayer.java & maybe Model.java]
Create the strongest AI player you can come up with. It will compete against a pretty strong AI of our own. Points are awarded depending on the score over a number of games. Games in which your AI submits invalid moves or tampers with the game state will be forfeited (i.e. counted as a loss). This player must be able to cope with variable game settings and games being loaded from file.

Warning: It may be tempting to pitch your AI against that of a fellow student, but exchanging your code (or even the .class files) would go against the the University's regulations for assessed work. What you could do, is send each other the moves your AI makes and input the moves you receive using the HumanPlayer (similar to correspondence chess).

## Restrictions

Please note that you will only be able to submit the files you have been asked to change! That means you should not edit any other files, as your solution must be compatible with the supplied versions of those files. Going against this can cause the automated tests to fail and will result in you losing the associated marks!

You are permitted to use the Java API, but you must not use any other/external libraries. You cannot upload libraries anyway, but you should also not copy library code into your the files you upload.

For your own learning, we recommend you do not use any functional language constructs such as lambdas or streams. Those were taught in first semester, but this course is about imperative programming. This is not enforced for assignment 1, but will be for assignment 3.

## Good Scholarly Practice

Please remember the good scholarly practice requirements of the University regarding work for credit. See: https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct
This also has links to the relevant University pages.

All code you submit must be your own! CodeGrade will detect copied code.

## Marking Criteria

Marks will be assigned in accordance with the University's Common Marking Scheme (CMS).
See: https://web.inf.ed.ac.uk/infweb/student-services/ito/students/common-marking-scheme

The basic requirements correspond to the description of a pass grade (up to 49%).
Intermediate features are required to demonstrated the mastery required for a 2$^{nd}$ class (up to 69%).
You will need to tackle the advanced section for a distinction (up to 100%).

Please note that attempting the more difficult features only removes grade ceilings. It does not guarantee you a mark in the corresponding range. You still need to fulfil all requirements for the lower categories as well!

## Submission, late submission & extensions

Submission instructions will be provided separately on LEARN → Assignments.

You can submit as often as you like until the deadline and will receive feedback from some of the automated tests, which you can use to improve. Submit often to avoid last minute upload issues!

This assignment must be **submitted on CodeGrade by 16:00 on Friday February 18$^{th}$**.

Information regarding late submission and extensions is also under LEARN → Assignments.