Informatics 1: Object Oriented Programming

Assignment 1 - Connect Four

The University of Edinburgh
David Symons (dsymons@exseed.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 basic program structure is provided in the form of incomplete Java classes. The following three sections describe the features you need to implement for a basic, intermediate and advanced solution. Your submission will be accompanied by a short technical report in which you describe how your program works and explain the reasons behind the design decisions you made.

Marks will be awarded out of 100 and count for 20% of your overall grade for Inf1B. Each 20 credit course is designed to involve around 200 hours of work. Of those, 10 hours have been allotted for this assignment. Completion time may vary depending on your level of experience and the grade you are aiming for. While you are encouraged to aim high, attempting the more difficult features is not a requirement and grades in the upper bands are expected to be rare. Details on marking criteria can be found in the eponymous section below.

Before you start, please read carefully the sections on *Restrictions* and *Good Scholarly Practice*. The final section gives instructions for how to submit your work, which is **due by 16:00 on Friday 12th February**.

Basic requirements

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

Step 1: Creating a new project with the provided code

- Create a new project called "ConnectFour-<UUN>" in IntelliJ (or your preferred IDE).
- Download the provided code (srcAssignment1.zip) from LEARN.
- Unpack the .java files into the *src* folder of your project e.g.: ConnectFour-s1234567/src/
  Please do not put the code into a package such as: ConnectFour-s1234567/src/main/

Step 2: Familiarise yourself with the provided code

- Study the provided code to get an idea of how the classes fit together. You will not have to change *ConnectFour.java* or *InputUtil.java*. They are there for your convenience, but you will need to understand how to use them.
- Pay particular attention to which class references an instance of which other class. This, together with the comments in the provided code, will help you understand the intended flow of control.
- Start forming a plan for how to fill in the missing parts of the program. You can revise this as you go, but having a high-level plan for how different parts fit together will lead to a cleaner design with less restructuring and debugging required.

Step 3: Model the state of the game

Think about how to represent the state of the board (use the default size of 6 rows and 7 columns) and whose turn it is. Then add the appropriate fields to *Model.java* and initialise the variables to represent the initial state of the game (an empty board, player 1 goes first).

Step 4: Display the board

Open *TextView.java* and complete the *displayBoard* method. Think about how you want to display the pieces belonging to the different players and make sure the board is printed the correct way up. You should already be able to run the program to check the output.

Step 5: Allow pieces to be played

Implement the *makeMove* method in *Model.java*. The provided code already calls this, so you should see a piece being inserted into column number 1. Make sure the piece is inserted in the correct position.

Step 6: Implement the game loop

Go to *Controller.java* and use methods provided by the *view* and the *model* to implement the flow of the game. There is some code here that hints at how the model and view should communicate. You will need to rearrange and add to this until the two players and take turns to play their pieces. The model must be updated after every move and the new state of the board must be printed out.

Step 7: Game over

The game should end either when the board is full or when either of the players concedes. Think about different ways of handling input to allow a player to give up. Detecting the winner is an intermediate feature.

Step 8: Complete any missing features

Make sure user the user knows whose turn it is and add any missing features to tidy up the game. Test it and make sure you catch some of the invalid inputs that can crash the game.

Step 9: Write a short report (important!)

Explain your solution and design decisions in a short report covering the following aspects:

- Give a brief, top-level description of the program’s major components and how they fit together.
- How did you represent the board and why did you choose that representation?
- How does the program keep track of which player goes next and how are players represented?
- How did you test and debug your application?
- Which problems did you encounter and how did you solve them?
- Are there any remaining issues? If so, do you have any ideas about how they could be solved?

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%.

Allow the user to start a new game (easy)

When the current game is over, ask the user(s) if they wish to play again. You will have to reset the game state and wipe the board.

Variable game settings (moderate)

At the start of a new game, you can enter the desired dimensions of the board (number of rows and columns). You should also be able to choose how many pieces you have to get in a row to win the game, so you can play Connect X. Make sure the game can be won.

Enhanced input validation (moderate)
Now that the game settings are variable, you will have to be very careful when validating user input. It should not be possible to crash the game or make invalid moves.

Automatic win detection (moderate-hard)
Write code to detect when the game is won and declare the winning player.

Report
State which of the intermediate features you have implemented in a separate section of your report. Then proceed to explaining your design. Describe any difficulties you encountered (whether you overcame them or not). If there are remaining issues, give some ideas for how they may be fixed.

Advanced features

Possible enhancements are described here. These are only required for a distinction. You may tackle any, all or suggest your own (provided the complexity is on par with the suggested tasks). Your report must clearly state which extensions you have attempted and briefly explain your approach (again use a separate section). A partial solution will give you some credit. In particular, you can gain credit by explaining any problems or limitations. Ideas for how to fix remaining issues should also be noted even if you do not have time to implement them.

Play against the computer (variable difficulty)
Implement an NPC (non-player character) that the user can choose to play against (human vs human must still be possible). This task can be as hard as you like. Start with a simple policy by which the computer makes arbitrary or random (but valid!) moves. For an improved AI you could:

- Always select a winning move if there is one
- Avoid making moves that allow the opponent to win on their next turn (if possible)
- ...

Save and load the game (hard)
Implement a mechanism by which you can write the state of the game to a file. This file can be loaded at a later stage to resume the game where you left off.

Research a design pattern
The provided code is based on a design pattern known as Model-View-Controller (MVC). Read up on this (or another pattern) and include a short summary of it in your report. This should explain how the pattern is used and what it is good for. The following link is a possible starting point for your research: https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

Restrictions

To demonstrate that you can apply the concepts you were taught, your implementation is restricted to using the basic features of the Java programming language. More specifically, the following rules are to be observed:

- Do not use any functional language constructs such as lambdas or streams.
- Do not use any collection classes such as lists or maps.
- Do not use any third party libraries.
- Keep all classes in the default package, i.e. no package. This makes marking a lot easier.

You are permitted to use the Java API (with the exception of data structures) and indeed, it is good practice to reuse existing code rather than reinventing the wheel. It is, however, unlikely you will need to do so for this assignment unless you attempt advanced features.

Good Scholarly Practice

As with all assessed work, you must fulfil the University requirements for good scholarly practice. Details can be found here: https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

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 2nd class (up to 69%).
You will need to tackle at least one extension to get a distinction (up to 100%).

Please note that attempting the more difficult features only removes grade capping. It does not guarantee you a mark in the corresponding range. You still need to fulfil all requirements for the lower categories. The quality of your code and report are also very important. Specifically, the following criteria will be taken into account:

Code

As a (relative) beginner, your first priority will be to write functionally correct code. However, to prevent bad habits from developing, you should pay attention to code quality from the very start. The following briefly explains desirable qualities:

- Structure
    - A good design should split the task in more manageable chunks (think divide and conquer) e.g. by a meaningful division of the code into classes (the provided code already does this).
    - Modularity and reusability: Classes and methods perform well defined tasks. They can be seen as building blocks for solving many different problems, not just the task at hand. Long methods are a symptom of a too problem-specific design.
    - Extensibility: New features can be added without redesigning the whole program.
- Robustness
    - The program is able to handle errors and invalid/unexpected inputs without crashing.
    - Correct use of access level modifiers (public/private) and static to restrict exposure.
- Technical/language proficiency
    - Use the appropriate features of your programming language. This means not restricting yourself to building everything using a few keywords when there is a more elegant solution. It also means not using a needlessly complex mechanism for solving a simple task.
- Readability (self-documenting code)
    - Use descriptive but concise names for variables, methods and classes.
    - Class names start with a capital letter, method and variable names with a lower case letter.
    - AllNamesAreWrittenInCamelCase
    - Exception: Static constants are written in SHOUT_CASE.
- Commenting
    - Quantity: Comments should be used where they are useful and nowhere else. Comment as much as you feel is needed for someone else to be able to understand your code – or to make sure you yourself can understand it when you come back to it a year later. Do not clutter really simple bits of code (that are self-documenting) with needless comments.
    - Quality: Comments should provide additional insight into what is happening in the code. They should not just re-state what the code is doing.

Documentation (i.e. your report)

Having to write a report may seem tedious, but it serves two important purposes. Documenting your thoughts and justifying why you have chosen a particular design deepens your own understanding. It also allows you to demonstrating knowledge and ensures that you can still get some credit for that even if something goes wrong with your code.

Below are (some of the) qualities a good report should have. They may not all be relevant to this assignment, but are intended to give you an idea of what is important in technical writing. Please do not worry about your level of English. This is not a (natural) language course and as long as you can make yourself understood, you will not lose marks.

- Content
  - Answer questions directly without beating about the bush.
  - Do not just write down everything you know about the topic, stick to what what was asked.
  - Provide an insight into your decision making. Your code shows what you did, your report should explain why you did it. For example, explain why you chose one design over another e.g. by laying out the alternatives and their advantages/disadvantages.
- Structure
  - Break the content down into well labelled sections.
  - If you are making an argument, ensure you organise your points into a logical sequence.
  - Try to make the text flow by avoiding frequent and abrupt changes of topic.
  - Finish with an honest evaluation of your work.
- Writing style
  - Should be clear and concise. Make the report as short as possible without cutting essential details.
  - Avoid repetition: Make your point once and make it well, but avoid re-stating it.

Submission

This assignment is **due at 16:00 on Friday 12th February**.

Please follow the following steps to submit your work:

1. Save your report as a PDF.
2. Put all .java files (including the provided ones) directly into a single folder called “src”.
   It should have no sub-folders and use no packages.
3. Create a single .zip file containing your report and code folder.
4. For instructions on uploading the .zip file, see the Assignment 1 folder on the LEARN page.

Resubmission

You can submit more than once up until the submission deadline. Your last submission before the deadline will be marked. Once the deadline has passed you can no longer change your submission.

Late submission

If you have not uploaded anything before the deadline, you can submit up to 7 days late. Be careful to submit the correct version, as resubmission is not possible. A lateness penalty of 5% per day will apply unless you have an extension and/or learning adjustment. The full policy can be found here: https://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests