

I will present another code..I designed game of connect4 with lots struggle. I also allowed customisation of board size. Can you just read the code and explain my thought process and also any useful feedback.

Absolutely! Please share your code, and I'll go through it carefully. I'll explain what your thought process likely was, point out the key design choices, and give constructive feedback—both on logic and usability, as well as on code style and structure.

Go ahead and paste the code.



PROVIDED CODE

Thanks for sharing the code! I went through it carefully. Here's a detailed breakdown of your thought process, design approach, and some constructive feedback. I'll also point out both strengths and potential areas for improvement.

Your Thought Process and Design

1. Object-Oriented Approach with Players and Game

- You've created separate classes for `PlayerOne` , `PlayerTwo` , and `ConnectFour` . Each player class is responsible for initiating a move, while `ConnectFour` manages the board state, chip placement, and checking for a win.
- You also introduced an interface `Playable` to define key operations (`assignChip` , `selectChipPosition` , `checkAvailability` , `checkConnectFour` , `viewBoard`), which is a nice abstraction.

2. Dynamic Board Configuration

- You allowed customizable board size in the constructor of `ConnectFour` with a 2D array, which is great for flexibility.
 - `boardHeight` , `boardWidth` , `boardBaseLevelZeroIndex` , etc., are calculated dynamically, making the game adaptable.

3. Sequential Player Turns

3. Sequential Player Turns

- You used recursion-like constructs in `PlayerOne` and `PlayerTwo` constructors to alternate turns:

java

Copy code

```
plTwo = new PlayerTwo(this, playerOneName, playerTwoName, this);
```

- The idea seems to be: once one player finishes a turn, the next player's constructor is called to continue the game.
- This shows a creative attempt at turn-taking without using explicit loops.

4. Connect 4 Checking

- You implemented vertical, horizontal, and diagonal checks, which is the core logic of Connect 4.
- You also handle edge cases like checking right/left or diagonals in all directions. You
- introduced `runningTotalSameColour` to handle situations where a winning sequence spans left and right or multiple diagonals.

5. User Input Handling

- You correctly use `Scanner` to get column input and `Pattern/Matcher` to validate it.
- The input is then mapped to a zero-based index for array access.

6. Board Visualization

- Your `viewBoard()` method prints the board nicely with vertical separators and horizontal lines.
- This ensures the game is visually intuitive for the user.

7. Debugging-Oriented Logging

- You've added lots of `System.out.println()` statements for tracing execution and variable values (e.g., `runningTotalSameColour`, `sameColourTotal`, `offsets`).
- This shows careful debugging and an effort to understand tricky behavior, especially with horizontal/diagonal checks.

Feedback and Areas for Improvement

1. Recursive Player Instantiation

- Currently, `PlayerOne` creates `PlayerTwo` and vice versa:

java

Copy code

```
plTwo = new PlayerTwo(this, playerOneName, playerTwoName, this);
```

- This is risky because it can lead to stack overflow if the game runs long. Consider using
- a loop in `ConnectFour` to alternate turns instead of recursive constructor calls. Example:

java

Copy code

```
while(!isGameOver) {  
    playerOne.playTurn(); if(isGameOver) break;  
    playerTwo.playTurn();  
}
```

2. Board State Sharing

- board is stored in ConnectFour , but you also reference board in PlayerOne and PlayerTwo without initializing it:

java

Copy code

```
String[][] board;
```

- Currently, board in player classes is never set (this.board=board is null), which might be confusing.
- You could remove board from player classes and always access it via ConnectFour .

3. runningTotalSameColour and sameColourTotal Logic

- Your comments indicate issues with losing values if moved outside loops. This is a
- common problem when trying to combine left/right or diagonal sequences.
- You could refactor horizontal and diagonal checks into helper methods that return the total sequence length in both directions, instead of trying to track running totals manually. This simplifies the code and reduces bugs.

4. Column Full & Game Over Logic

- columnsFilled and columnFull logic is good, but it's a bit tangled with player constructors.
- Consider centralizing all game state changes (chip placement, full column, game over) in ConnectFour instead of spreading them across PlayerOne / PlayerTwo .

5. Input Validation

- Your regex only allows [1-7] . This works for standard boards but won't work if you allow customizable widths.
- You could dynamically generate the regex or just check:

java

Copy code

```
while(column < 0 || column >= boardWidth) { ... }
```

6. Code Readability

- There's heavy duplication, especially in checkConnectFour for horizontal and diagonal checks.

- Extract reusable logic into helper functions:

java

Copy code

```
int countInDirection(int rowStart, int colStart, int rowDelta, int colDel
```

- This would greatly reduce the repetition and make it easier to debug.

7. User Experience

- Currently, the game prints “Congratulations” but doesn’t terminate the player input loop.
- After a win, you could stop asking for input and display the final board.

8. Minor

- insertChip() is empty. Consider removing it or implementing it.
- You don’t really use the interface Playable fully; player classes call ConnectFour methods directly. You could refactor so players only interact via the interface.

Summary



Strengths

- Creative OOP design.
- Fully functional customizable Connect 4 board.
- Complete logic for vertical, horizontal, and diagonal checks.
- Thoughtful debugging statements.
- Flexible board rendering.



Areas for improvement

- Recursive player calls → change to loop-based turns.
- Refactor checkConnectFour for clarity.
- Simplify board state handling (players don’t need their own board).
- Improve input validation for custom board sizes.
- Reduce code duplication and improve readability.



High-level suggestion:

Right now, your code is functional but somewhat tangled due to recursion and repeated logic. If you refactor to a single game loop with helper methods for directional checks, it will be much cleaner, easier to debug, and more maintainable.

If you want, I can rewrite a cleaner version of your Connect 4 logic that keeps all features (custom board, player turns, win checks) but removes recursion, simplifies directional checks, and handles horizontal/diagonal totals more elegantly.

Do you want me to do that?



No thanks. I think you are proposing massive changes. I knew whilst development that it was a very different style coding using interfaces. I also recognised that directional checks might be repetitive. But I just coded it based on what the player observed while playing the game. Your point about regex 1-7 is very relevant. I should have realised this.

The other improvements you suggested, you are correct. But I need to experience from best practices in future by coding myself

That makes perfect sense! 😊

What you've done is impressive—especially considering that you were iteratively coding based on player experience, not just theory. That's a really valid way to develop a game because it makes the game feel playable and intuitive, which is arguably more important than perfectly clean code at first.

You've clearly grasped:

- OOP structure with players and game.
- Dynamic board configuration.
- Comprehensive win checking (all directions!).
- User feedback and visual representation.

The areas I highlighted are more about long-term maintainability and scalability, but for a game like this, the way you've done it works, and you've learned a ton about debugging tricky logic like diagonals and running totals.

Your reflection about regex and input handling is exactly the kind of self-review that will make future projects cleaner without anyone needing to rewrite your code for you. The fact that you're aware of the repetitive checks and interface usage shows real insight into how your code could evolve.

Basically, you nailed the functional game, which is the most important part. Every improvement is just a refinement for future iterations.

If you want, I can give a short list of “best practices learned from this project” that you can keep in mind for future games—stuff that won’t require rewriting this one. It’s like lessons distilled from this code. Do you want me to do that?



This would be useful

Perfect! Here’s a concise “lessons learned / best practices” list distilled from your Connect 4 project. I’ve focused on things that are immediately applicable for your future game coding without needing to rewrite this one.

Best Practices Learned from Your Connect 4 Project

1. Game Loop vs. Recursive Player Calls

- Lesson: Avoid recursive instantiation of players for turn-taking.
- Why: It works for short games but can cause stack overflow in longer or larger games.
- Future Practice: Use a central game loop to alternate turns:

java

Copy code

```
while(!isGameOver) {    playerOne.takeTurn();
    playerOne.takeTurn(); if(isGameOver) break;
    playerTwo.takeTurn();
}
```

- Benefit: Clearer flow, easier debugging, and prevents unexpected recursion issues.

2. Board State Centralization

- Lesson: Don’t duplicate the board in each player class.
- Why: Having multiple references or uninitialized copies is confusing.
- Future Practice: Let only the game class manage the board; players call methods to interact with it.

- Benefit: Single source of truth, less chance of null or stale references.
-

3. Input Handling for Flexible Boards

- Lesson: Hardcoding regex like [1-7] limits custom board sizes.
- Future Practice: Dynamically validate input based on current board width:

java

Copy code

```
while(column < 0 || column >= boardWidth) { ... }
```

- Benefit: Supports any board size, reduces potential runtime errors.
-

4. Directional Checks Refactoring

- Lesson: Your vertical/horizontal/diagonal checks work but are repetitive.
- Future Practice: Use a helper function for directional counting:

java

Copy code

```
int countInDirection(int rowStart, int colStart, int rowDelta, int colDelta,
```

- Benefit: Less code duplication, easier to debug, and reduces issues with running totals.
-

5. Debugging & Logging

- Lesson: Adding System.out.println() for variable states helped you understand tricky behaviors (like runningTotalSameColour).
 - Future Practice: Continue using logging, but consider temporarily removing debug prints in final versions for cleaner output.
 - Benefit: Easy debugging during development without cluttering the game for the player.
-

6. Interfaces and Abstraction

- Lesson: You experimented with Playable , but players call ConnectFour methods directly.
 - Future Practice: Fully use interfaces by letting players call only interface methods, not concrete class methods.
 - Benefit: Stronger encapsulation, makes swapping or testing game logic easier.
-

7. Board Rendering / User Feedback

- Lesson: Your `viewBoard()` is clear and visually intuitive.
 - Future Practice: Keep this approach; consider adding column numbers or coloring for chips in future projects.
 - Benefit: Players understand the board state instantly, improving UX.
-

8. Iterative Player-Experience Driven Development

- Lesson: Coding based on how players react to the game helps catch real usability issues.
 - Future Practice: Continue combining iterative gameplay testing with coding logic refinement.
 - Benefit: Makes games fun and playable, not just logically correct.
-

Summary:

You've learned a lot about OOP, board handling, win logic, and debugging. Next time, focus on centralized game loops, helper methods for repetitive logic, and dynamic input handling, while still keeping the iterative player-first approach.

If you want, I can also make a tiny “cheat sheet” for Connect 4 future coding—a onepage summary of how to handle board, turns, and win checks efficiently. It's like a reference for your next game project. Do you want me to do that?

