**INTRODUCTION**

A **Simple Memory Game** (also known as "Concentration" or "Matching Pairs") is a fun and interactive game that challenges players to match pairs of cards by flipping them over. Typically, the cards are arranged face down, and the player flips two cards at a time. If the cards match, they remain face up; if they don't, they are flipped back face down. The goal of the game is to match all pairs in as few moves as possible.

Here's a basic **Java** implementation of a simple memory game that you can enhance over time. This version is a text-based, console game for simplicity. You can modify it to have a graphical user interface (GUI) later using libraries like Swing or JavaFX.

**Step-by-Step Guide**

1. **Card Representation:** We'll represent each card with a character or integer. The game will randomly shuffle these cards and lay them out.
2. **User Input:** The player will input two card positions at a time to flip and check for matches.
3. **Game Loop:** The game will continue until all pairs are matched.

**PROJECT REQUIREMENT**

A **Simple Memory Game** (also known as "Concentration" or "Matching Pairs") is a fun and interactive game that challenges players to match pairs of cards by flipping them over. Typically, the cards are arranged face down, and the player flips two cards at a time. If the cards match, they remain face up; if they don't, they are flipped back face down. The goal of the game is to match all pairs in as few moves as possible.

Here's a basic **Java** implementation of a simple memory game that you can enhance over time. This version is a text-based, console game for simplicity. You can modify it to have a graphical user interface (GUI) later using libraries like Swing or JavaFX.

**Step-by-Step Guide**

1. **Card Representation:** We'll represent each card with a character or integer. The game will randomly shuffle these cards and lay them out.
2. **User Input:** The player will input two card positions at a time to flip and check for matches.
3. **Game Loop:** The game will continue until all pairs are matched.

**1. Functional Requirements:**

These are the core functionalities that the game should support.

**1.1 Game Setup:**

* **Grid Size:** The game should have a fixed grid size (e.g., 4x4, 6x6) where cards are placed face down initially.
* **Cards:** Each card should have a pair, and all pairs should be shuffled randomly.
  + For example, with a 4x4 grid (16 cards), there will be 8 pairs of cards, and each card should be represented by a character or symbol (e.g., 'A', 'B', 'C', etc.).

**1.2 Game Flow:**

* **Card Reveal:** The player should be able to select two cards at a time by entering their row and column (e.g., 0 1 for the first card).
* **Match Check:** If the two selected cards match, they should remain face up; if not, they should flip back face down.
* **Turn Handling:** After each turn, the game should check whether the two cards match or not and update the board accordingly.
* **Game End Condition:** The game ends when all pairs of cards are found and the board is completely revealed.

**1.3 User Interaction:**

* **User Input:** The player will input coordinates (row and column) to select cards, typically in the format row col.
  + The game should validate that the user input is within the bounds of the grid (e.g., 0-3 for a 4x4 grid).
  + If a card is already revealed, the player should be prompted to select different cards.

**1.4 Game Feedback:**

* **Feedback on Success/Failure:** After each selection of two cards, the game should provide feedback such as:
  + "It's a match!"
  + "No match! Try again."
* **Display Board:** After each turn, the updated board should be displayed, showing the state of the cards (whether face up or face down).

**1.5 Game Reset/Restart:**

* After completing a game, the user should be given the option to restart the game with a new shuffled deck of cards.

**2. Non-Functional Requirements:**

These are the quality attributes or constraints that should be considered while developing the game.

**2.1 Performance:**

* **Efficiency:** The game should run efficiently, with fast feedback and updates (even with larger board sizes like 6x6 or 8x8).
* **Responsive Input:** User input should be processed immediately after being entered (with appropriate validation).

**2.2 Usability:**

* **User Interface:** The text-based user interface (CLI) should be clear and easy to navigate.
  + The board should be easy to read and understand, with cards clearly labeled as face-down (?) or showing the correct symbol when revealed.
  + The prompt for entering the card positions should be simple, e.g., "Enter row and column for the first card."

**2.3 Maintainability:**

* **Modular Code:** The game logic should be well-organized, and methods should be modular for easy maintenance and extension.
* **Code Comments:** Proper comments and documentation should be added throughout the code to explain the logic, especially for key functions like board setup, input validation, and checking for matches.

**2.4 Scalability:**

* **Grid Size Adjustability:** The grid size should be configurable, meaning the game can work with various board sizes (e.g., 4x4, 6x6, 8x8).
* **Card Handling:** Ensure the game logic can handle different numbers of cards, both in terms of logic and layout.

**2.5 Reliability:**

* **Error Handling:** Proper validation for user inputs (e.g., checking if the entered coordinates are valid or not) and handling edge cases like entering out-of-bounds coordinates.
* **Game State Persistence:** If the game is restarted, the game state should be reset correctly without errors.

**3. Optional Features for Extension (Future Enhancements):**

Here are some additional features that can be added later to enhance the game:

**3.1 User Interface (UI):**

* **Graphical Interface:** Implement a GUI using JavaFX or Swing, where users can click on cards rather than entering row and column coordinates manually.
* **Animations:** Add animations to flip the cards and show matching pairs.

**3.2 Difficulty Levels:**

* **Custom Grid Sizes:** Allow the player to choose different grid sizes (e.g., 4x4, 6x6, 8x8, etc.).
* **Time Limit:** Add a timer for each turn, or track the total time taken to complete the game.

**3.3 Score System:**

* **Moves Counter:** Count and display the number of moves or turns taken to complete the game.
* **High Scores:** Store the best scores based on the fewest moves or time and display a leaderboard.

**3.4 Sound and Effects:**

* Add sound effects for card flipping, matching pairs, and game completion (if implementing a GUI).

**3.5 Multiplayer:**

* **Turn-Based Multiplayer:** Implement a turn-based multiplayer mode where two players take turns to match cards.
* **Online Mode:** For advanced projects, you could introduce networking to allow multiplayer games over the internet.

**4. Technical Specifications:**

**4.1 Programming Language:**

* Java 8 or higher (for more advanced features like lambdas or streams, although not necessary for the basic game).

**4.2 Libraries:**

* **Standard Java Libraries:**
  + java.util.Scanner: For reading user input.
  + java.util.ArrayList, java.util.List: For managing the cards and their pairs.
  + java.util.Collections: For shuffling the cards.
  + java.util.Random: If you want to randomly generate card pairings (though Collections.shuffle is generally used).

**4.3 Game Environment:**

* **IDE:** Any IDE such as IntelliJ IDEA, Eclipse, or NetBeans for development.
* **JVM:** Java Development Kit (JDK 8 or above.

**5. Milestones and Timeline:**

Here's a suggested breakdown of the tasks and milestones for the project:

1. **Initial Setup (1-2 Days):**
   * Set up the project structure.
   * Implement basic board setup and card generation.
2. **Core Game Logic (3-5 Days):**
   * Implement the main game loop.
   * Handle user input and card revealing.
   * Check for matches and update board state.
3. **User Interface (2-3 Days):**
   * Implement board display.
   * Provide user feedback after each turn.
4. **Testing and Debugging (2-3 Days):**
   * Test the game thoroughly for bugs and unexpected behaviors (e.g., out-of-bound inputs, game crashes).
   * Optimize the code if necessary.
5. **Optional Features (5+ Days):**
   * If you plan to implement additional features such as a scoring system, difficulty levels, or a GUI, allocate extra time for that.

**6. Deliverables:**

At the end of the project, the following deliverables should be provided:

* **Source Code:** Well-commented, modular Java code.
* **Game Documentation:** Instructions on how to play the game, expected inputs, and features.
* **Test Results:** Any unit tests or integration tests you performed, especially for input validation and game logic.

**UML DIAGRAM**

|  |
| --- |
| **MemoryGame** |
| - BOARD\_SIZE: int  - board: String[][]  - revealed: boolean[][]  - pairsFound: int |  - values: String[] |
| + main(args: String[]): void  - initializeBoard(): void  - printBoard(): void  - isValidMove(row: int, col: int): boolean |

Top of Form

**SOURCE CODE**

import java.util.\*;

public class MemoryGame {

    private static final int BOARD\_SIZE = 4;  // 4x4 grid (16 cards)

    private static String[][] board;          // 2D array for the card values

    private static boolean[][] revealed;     // 2D array to track revealed cards

    private static int pairsFound = 0;        // Count of pairs found

    private static String[] values = {"A", "B", "C", "D", "E", "F", "G", "H"}; // Card values

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        board = new String[BOARD\_SIZE][BOARD\_SIZE];

        revealed = new boolean[BOARD\_SIZE][BOARD\_SIZE];

        // Initialize and shuffle the board

        initializeBoard();

        // Main game loop

        while (pairsFound < (BOARD\_SIZE \* BOARD\_SIZE) / 2) {

            printBoard();

            System.out.println("Enter coordinates of two cards to flip (row and column between 0 and 3):");

            // Get first card

            System.out.print("First card (row column): ");

            int row1 = scanner.nextInt();

            int col1 = scanner.nextInt();

            // Get second card

            System.out.print("Second card (row column): ");

            int row2 = scanner.nextInt();

            int col2 = scanner.nextInt();

            // Flip the cards

            if (isValidMove(row1, col1) && isValidMove(row2, col2)) {

                if (board[row1][col1].equals(board[row2][col2])) {

                    revealed[row1][col1] = true;

                    revealed[row2][col2] = true;

                    pairsFound++;

                    System.out.println("Match found!");

                } else {

                    System.out.println("No match! Try again.");

                }

            } else {

                System.out.println("Invalid move. Try again.");

            }

            // Check if the game is over

            if (pairsFound == (BOARD\_SIZE \* BOARD\_SIZE) / 2) {

                printBoard();

                System.out.println("Congratulations! You've matched all pairs!");

            }

        }

        scanner.close();

    }

    // Initialize the board with shuffled pairs of values

    private static void initializeBoard() {

        List<String> cards = new ArrayList<>(Arrays.asList(values));

        cards.addAll(Arrays.asList(values)); // Duplicate values for pairs

        Collections.shuffle(cards);  // Shuffle to randomize the cards

        int cardIndex = 0;

        for (int i = 0; i < BOARD\_SIZE; i++) {

            for (int j = 0; j < BOARD\_SIZE; j++) {

                board[i][j] = cards.get(cardIndex++);

                revealed[i][j] = false;  // Initially no cards are revealed

            }

        }

    }

    // Print the board state, showing only revealed cards

    private static void printBoard() {

        System.out.println("\nBoard:");

        for (int i = 0; i < BOARD\_SIZE; i++) {

            for (int j = 0; j < BOARD\_SIZE; j++) {

                if (revealed[i][j]) {

                    System.out.print(board[i][j] + " ");

                } else {

                    System.out.print("? ");

                }

            }

            System.out.println();

        }

    }

    // Check if the selected card position is valid

    private static boolean isValidMove(int row, int col) {

        return row >= 0 && row < BOARD\_SIZE && col >= 0 && col < BOARD\_SIZE && !revealed[row][col];

    }

}

/\*

output

Board:

? ? ? ?

? ? ? ?

? ? ? ?

? ? ? ?

Enter coordinates of two cards to flip (row and column between 0 and 3):

First card (row column): 0 0

Second card (row column): 1 0

No match! Try again.

Board:

? ? ? ?

? ? ? ?

? ? ? ?

? ? ? ?

Enter coordinates of two cards to flip (row and column between 0 and 3):

First card (row column): 0 0

Second card (row column): 0 1

Match found!

Board:

A ? ? ?

A ? ? ?

? ? ? ?

? ? ? ?

Enter coordinates of two cards to flip (row and column between 0 and 3):

Congratulations! You've matched all pairs!

Board:

A A B B

C C D D

E E F F

G G H H

\*/

**EXPLANATION OF CODE**

**Memory Game Explanation and Review**

The provided code implements a **Memory Game** in Java with a 4x4 grid of cards. The goal of the game is to match pairs of cards. Below is a detailed breakdown and review of how the code works, along with suggestions for potential improvements.

**Overview of the Code**

1. **Game Setup:**
   * **Grid:** A 4x4 grid is used, so there are 16 cards in total. This grid size is specified by the constant BOARD\_SIZE.
   * **Card Values:** The cards are represented by a set of letters (A to H). Each value is duplicated to form pairs (e.g., two 'A's, two 'B's, etc.).
   * **Shuffling:** The cards are shuffled randomly using Collections.shuffle(), ensuring that each game will have a different card arrangement.
2. **Gameplay Loop:**
   * The game repeatedly asks the user to select two cards (by entering row and column numbers).
   * If the cards match, they are revealed (marked as true in the revealed array), and the number of matched pairs is incremented.
   * The game continues until all pairs are found.
3. **Board Display:**
   * The printBoard() method is responsible for displaying the current state of the board. Revealed cards are shown with their respective values, and unrevealed cards are displayed as ?.
4. **Validation:**
   * **Input Validation:** The game checks if the selected coordinates are valid (within bounds and the card is not already revealed) using the isValidMove() method.
   * **Match Check:** After flipping two cards, the game checks if the selected cards match and gives feedback.
5. **Game End:**
   * The game congratulates the player once all pairs are found.

**Code Review and Suggestions for Improvement**

1. **Card Values and Shuffling:**
   * Currently, the card values are hard-coded into the values array. If you want to extend the game to support more grid sizes (e.g., 6x6), the card set needs to be dynamically adjusted.
   * **Improvement:** You could dynamically generate the card values based on the board size. For example, if you have a 6x6 grid (36 cards), you would need 18 unique values.

**CONCLUSION**

The provided code is a solid foundation for a memory game in Java. It implements the core game mechanics of shuffling cards, revealing them, and checking for matches. By applying the suggested improvements (such as input validation, delay between turns, and game restart options), the game can become more robust, user-friendly, and scalable for larger grids or additional features.

**Array Usage: Key Insights**

1. **2D Arrays for Grid Representation:** The board and revealed 2D arrays are central to the game, where the board holds the values of the cards and revealed tracks their state (face-up or face-down).
2. **Dynamic Shuffling with Lists:** The cards list is dynamically populated and shuffled using a combination of arrays and a List. This makes the board setup random each time the game is played.
3. **User Interaction with Arrays:** The user interacts with the arrays indirectly through the input coordinates (row and column). These inputs are used to update the state of the game by comparing and modifying the arrays.
4. **Efficient Memory Use:** Arrays provide an efficient and straightforward way to store and manipulate the game state. They allow easy access to card values and visibility states, making the game logic simple to implement and maintain.

In this **Memory Game** implementation, arrays are essential to managing the state of the game, including the board's contents and the visibility of cards. The game uses both **1D** and **2D arrays** to manage the layout and behavior of the cards. By leveraging arrays for the board and card state, the game remains organized and scalable, allowing easy changes to grid size or card values. The combination of arrays allows the game to handle dynamic behavior like shuffling, revealing, and matching cards efficiently. With some minor enhancements (such as input validation, dynamic grid size, and error handling), this basic array-based structure can be easily expanded to support more complex features.

Top of Form