The Old Maid Card Game

A report

Done by: Aya Almallah

Instructors:

Motasim Aldiab

Fahed Jubair

Contents

1 Introduction

This report details the implementation of the Old Maid Card Game using Java Multithreading. during Atypon's Training in May 2023. The objective of the assignment was to create a simulation of the game where players, represented as threads, engage in the game autonomously. The report discusses the utilization of Java's multithreading capabilities, synchronization mechanisms, and the OOP design employed in the implementation.

2 Object-Oriented Design and Classes

The implementation of the Old Maid card game revolves around a well-structured set of classes, each responsible for specific aspects of the game's mechanics, player interactions, and overall gameplay. Here, we delve into the key classes that form the foundation of the implementation:

2.1 Enums

The three distinct enums—CardSuit, CardValue and CardColor—effortlessly encapsulate the fundamental characteristics of playing cards, forming the foundation of their representation within the implementation.

- CardColor Enum: This enum encapsulates the colors of playing cards, categorizing them into RED and BLACK based on their suits.
- CardSuit Enum: Enumerating the possible suits of playing cards, this enum defines the four traditional suits—SPADES, CLUBS, HEARTS, and DIAMONDS—as well as the special Joker.
- CardValue Enum: Mirroring the values of playing cards, this enum encompasses the full range of card values from ACE to KING, inclusive. Like the other enums, it also accommodates the Joker value. The enum offers the fromNumber method to convert numeric indices to corresponding card values, enabling convenient card instantiation.

2.2 Card Class

The Card class serves as the foundation for representing playing cards in the game. This class holds the essential traits of a card and integrates seamlessly with other parts of the game. In the Card class, the following methods contribute to its role:

- isMatchingPair(Card other): This method critically determines if two cards constitute a matching pair. By evaluating the values and colors of both cards, this method encapsulates the logic required to identify matching pairs.
- toString(): The toString() method creates a human-readable card representation. It's used for displaying cards conveniently and aiding in debugging, making card information easily understandable.

By enclosing fundamental attributes like suit, value, and color within the Card class, the implementation achieves a higher level of abstraction. This encapsulation fosters a clean separation of concerns and enhances code organization, grouping card-related attributes and behaviors logically.

2.3 Deck Class

The Deck class forms the backbone of card management, driving vital game operations. It covers:

- initializeDeck(): Constructs the deck with cards, including the Joker. Cards are added to set the scene.
- shuffleDeck(): Shuffles cards using Collections.shuffle(), ensuring unpredictability for dynamic gameplay.
- isEmpty(): Informs if the deck lacks cards or has potential draws by checking its size.
- drawCard(): Provides a card by taking the last from the deck, even handling an empty deck.

These deck functions reside in the Deck class, adhering to the "Single Responsibility Principle." This design boosts order and manageability, ensuring each class fulfills its role. The Deck class's methods supply essential tools for deck status, drawing cards, and maintaining a shuffled deck ready for play.

2.4 Player Class

The Player class embodies the core of individual players in the Old Maid card game. Acting as thread-based entities, it wraps player-specific actions and interactions. The class's methods play a central role in guiding the player's involvement throughout the gameplay:

- addCardToHand(Card card): This method adds a card to the player's hand. By appending the card to their hand's list, it takes care of constructing and managing their hand.
- isFinished(): The isFinished() method checks if a player has discarded all their cards, marking the end of their game. By sizing up their hand, it signals their progress.
- showHand(): Responsible for showing the player's cards, the showHand() method ensures their hand is visible. It cycles through the cards, displaying each one's suit and value.
- discardMatchingPairs(): This method handles spotting and removing matching pairs of cards from the player's hand. By systematically comparing cards, it efficiently removes pairs while keeping the hand updated.
- takeTurn(): At the heart of the player's turn lies the takeTurn() method. It draws a random card from the next player, incorporating it into the current player's hand. This step includes verifying potential matching pairs and initiating their removal.
- notifyGame(): Employing synchronization, the notifyGame() method signals the game controller to advance to the next turn. By notifying the turn lock object, it guarantees the seamless progression of the turn-based gameplay.
- run(): Serving as the player's thread entry point, the run() method captures their entire gameplay sequence. It orchestrates turns, interactions with fellow players, and overall progression. The method navigates synchronization, turn transitions, and communication with other player threads.

The Player class encapsulates these methods, capturing player actions like card handling, turn-taking, and interaction with the game's environment. These intricately crafted methods truly embody the dynamic spirit of the Old Maid card game, enhancing engagement and delivering an authentic gameplay experience.

2.5 Game Class

The Game class holds the core of the Old Maid card game, knitting together the game's flow and player interactions. Its collection of methods works hand in hand to weave a synchronized and engaging experience:

- initializePlayers(int numPlayers): This method sets up player threads and their connections. It assigns each player their next player and the shared turn lock, creating the backbone of turn-based play.
- dealCards(): The dealCards() method manages card distribution. It gives each player cards from the deck, ensuring an even spread and exhausting the deck evenly.
- removeMatchingPairsForAllPlayers(): In this method, players remove pairs from their hands. This ensures they begin with only unmatched cards, setting the initial gameplay state.
- removePlayersWithEmptyHands(): Here, players with empty hands are removed. It's a sign that they've discarded all their cards and are no longer part of the game.
- startingPlayerThreads(): This method starts all player threads, setting them in motion. By triggering each player's start() method, the gameplay gets going.
- stopGame(): The stopGame() method gracefully halts the game. It tells player threads to stop, ensuring proper termination and resource release.
- run(): Serving as the game's thread entry, the run() method encapsulates the entire gameplay. It oversees card distribution, player turns, and the game's outcome. By using synchronization and multithreading, this method delivers a seamless and captivating gaming experience.

The Game class's methods work in harmony, weaving together the intricate fabric of the Old Maid card game. By encapsulating game mechanics, turn management, and player interaction, this class embodies the game's essence and demonstrates the potency of object-oriented design in crafting an immersive gaming adventure.

2.6 Main Class

At the front of user interaction and program flow, the Main class takes the spotlight as the gateway to the Old Maid card game. This class drives user participation, game kickoff, and the wrap-up of gameplay:

- main(String[] args): The main method choreographs program execution. It prompts users to type in the number of players via the console, starts the Game class with the specified player count, and oversees game advancement.

By capturing user input and initiating the game, the Main class offers an approachable interface for the Old Maid card game. Its role, propelled by execution, encourages interaction and ensures a smooth shift from user input to an active game realm.

3 Thread synchronization mechanisms

In the Old Maid card game, effective thread synchronization mechanisms are crucial to ensuring that multiple players can interact, take turns, and perform actions concurrently without introducing race conditions or conflicts. The simulation employs synchronization constructs to orchestrate the interactions between the Game class and the individual Player threads.

1. **synchronized Blocks and Methods**
    - Both the Game and Player classes utilize the synchronized keyword to define critical sections where only one thread can execute at a time. This prevents multiple threads from modifying shared resources simultaneously.
    - In the Game class, synchronized blocks are used to manage access to the turnLock object. This ensures that player threads wait their turn before executing their actions, and the main game thread waits for player threads to finish their turns.
    - In the Player class, synchronized blocks are used to coordinate player turns and interactions. The takeTurn() method synchronizes access to the nextPlayer object, enabling controlled selection of cards from other players.
2. **wait() and notify() Methods**
    - The wait() and notify() methods play a pivotal role in coordinating actions between threads. In the Player class, the run() method employs this.wait() to pause a player thread until it's notified by the game to take its turn.
    - The main game thread uses turnLock.wait() to pause itself, allowing player threads to complete their turns before proceeding. The main thread then uses player.notify() to signal to a player that it's their turn to play.

3. **Ensuring Sequential Turns**

    - The combination of wait() and notify() methods, along with synchronized blocks, ensures that player turns are executed in a sequential manner, avoiding concurrency issues.
    - The coordination guarantees that players take turns one after the other, preventing scenarios where multiple players attempt to take actions simultaneously.

4. **Main Interaction with Game Thread**

    - In the main thread of the program, the game.join() call is used to ensure that the main thread waits for the Game thread to finish before proceeding.
    - This synchronization mechanism guarantees that the entire game simulation, including player interactions and turns, is completed before the program terminates.

5. **Preventing Race Conditions**

    - The synchronization mechanisms used in the simulation effectively prevent race conditions, ensuring that player actions like card selection and hand management are performed in a synchronized manner.
    - For instance, the synchronization in the takeTurn() method of the Player class ensures that no two players can access the same card from another player's hand concurrently.

Conclusion

The robust thread synchronization mechanisms integrated into the Old Maid card game demonstrate the careful design and implementation of multithreaded interactions. By employing synchronized blocks, wait() and notify() methods, and fostering controlled access to shared resources, the simulation ensures that players' actions are synchronized, coordinated, and executed without conflicts. This approach maintains the integrity of the game's logic and guarantees a seamless and controlled gameplay experience within a multithreaded environment.

4 Clean Code Principles

The implementation of the Old Maid card game aligns with the clean code principles advocated by Robert C. Martin, commonly referred to as Uncle Bob. These principles emphasize the creation of code that is organized, maintainable, and readable, fostering a conducive environment for development and future enhancements. Let's explore how these principles manifest in different facets of the codebase:

Readability and Simplicity:

- **Descriptive Naming:** Meaningful names for classes, methods, and variables enhance code readability, making it easier for developers to understand the purpose and functionality of each element.
- **Structured Layout:** Consistent indentation and formatting contribute to a clean and organized appearance, reducing visual clutter and aiding in code comprehension.
- **Concise Methods:** Methods are designed to be concise, focusing on performing a single task. This approach enhances readability by isolating specific functionality within well-defined methods.

Code Reusability:

- **Abstraction and Inheritance:** The use of inheritance, particularly with the **Player** class, allows for the creation of various player types that share common attributes and behaviors. This promotes code reusability by abstracting shared functionality.

Single Responsibility Principle (SRP):

- **Focused Classes:** Each class adheres to the SRP, encapsulating a single responsibility. For example, the **Card** class is solely concerned with defining card attributes and behaviors, while the **Deck** class manages deck-related operations. This organization simplifies code maintenance and debugging.

Open/Closed Principle (OCP):

- **Extensible Enums:** Enums representing card attributes (**CardColor**, **CardSuit**, **CardValue**) provide a foundation for adding new values or suits without altering existing code. This exemplifies the OCP by allowing extension without modification.

Incorporating these principles into the Old Maid card game implementation contributes to a codebase that is not only readable and simple but also designed for reusability and maintainability. Adhering to SRP and OCP fosters a modular architecture that can accommodate changes and enhancements with minimal impact on existing code.