Old Maid Assignment

Made by: Samer AlHamdan

Instructors:

Motasim Aldiab

Fahed Jubair

1. Introduction **3**

2. The Assignment **3**

3. OOP Design and Classes **4**

3.1 Enums 4

3.2 Cards 5

3.3 Deck 6

3.4 Player 7

3.5 GameManager 9

3.6 Main 10

4. Thread Synchronization Mechanism **12**

5. Code Cleanness **13**

1. Introduction

This report offers a comprehensive analysis and optimization strategy for the Old Maid Card Game using Java multithreading and object-oriented programming concepts. during Atypon's Training in May 2023. The game is played automatically by the computer without any human players involved. Each player in the game is run as a separate thread, and the main thread takes on the role of the game manager, responsible for starting and managing the game.

2. The Assignment

The Old Maid Card Game assignment aims to implement the classic card game using Java multithreading and OOP principles. The game involves multiple players, and each player is represented as a separate thread in the program. The main objective is to automatically simulate the game's flow, with each player taking turns to play the game until a loser is determined.

Game Rules:

1. The game is played with a standard 52-card deck, along with one additional card, the Joker.
2. Players are dealt cards equally at the beginning of the game.
3. The goal is to form matching pairs of cards with the same value and color.
4. The Joker has no match and cannot form a pair with any other card.
5. At the beginning of the game, players remove any matching pairs from their initial hand.
6. Each player takes turns to pick a random card from the next player, potentially forming a pair and discarding it.
7. The game continues until one player is left with the Joker, who is considered the loser.

Approach:

1. The game is played automatically without human intervention.
2. Each player is a separate thread, and the game manager acts as the main thread.
3. The turn-based gameplay is simulated using wait and notify mechanisms between player threads.

3. OOP Design and Classes

The project follows a well-organized Object-Oriented Programming (OOP) design, utilizing classes and enums to represent various components of the game. The main classes and their responsibilities are as follows:

3.1 Enums

The three enum classes (`Value`, `Type`, and `Color`) add valuable organization and representation of the possible values and attributes related to playing cards in the Old Maid Card Game project. Let's explore the value they add to the Object-Oriented Programming (OOP) design of the project:

1. `Value` enum:
    - Represents the possible values that a card can have, such as TWO, THREE, FOUR, etc.
    - Helps to encapsulate and categorize the different card values.
    - Provides a clear and expressive way to refer to card values throughout the codebase, improving code readability and maintainability.
    - Enables type safety, ensuring that only valid card values are used in the program.
2. `Type` enum:
    - Represents the card types or suits, including SPADES, HEARTS, CLUBS, and DIAMONDS.
    - Helps to categorize and distinguish cards based on their suits.
    - Allows the code to use descriptive names for card suits, enhancing code clarity and reducing the risk of errors due to misspellings or incorrect values.
3. `Color` enum:
    - Represents the colors of the cards, i.e., RED and BLACK.
    - Enables grouping and easy identification of card colors, which is useful when determining matching pairs in the game.
    - Provides a concise and meaningful representation of the card colors, enhancing code readability.

By using enums, the code gains the benefits of well-structured, constant-like representations for card values, suits, and colors. This adheres to good OOP practices by encapsulating related data into classes, promoting reusability, and reducing the risk of error-prone string or integer representations. Enums also help in enforcing type safety and improving code maintainability, making it easier to understand and modify the codebase in the future. Overall, the enum classes contribute to a more structured, organized, and robust design for the Old Maid Card Game.

3.2 Cards

The `Card` class is used to represent a playing card in the Old Maid Card Game. Each instance of the `Card` class corresponds to a single card with a specific type (suit), value, and color.

1. **Properties:**
    - `type` (Type enum): Represents the type or suit of the card.
    - `value` (Value enum): Represents the value of the card.
    - `color` (Color enum): Represents the color of the card. It is calculated based on the card's type.
2. **Constructor:**
    - The constructor takes `type` and `value` as parameters and initializes the corresponding properties of the card.
    - Additionally, it calculates the `color` of the card based on the `type` . If the card's type is CLUBS or SPADES, the color is set to BLACK. If the type is DIAMONDS or HEARTS, the color is set to RED.
3. **Methods:**
    - `getValue()` : This method returns the value of the card.
    - `getColor` : This method returns the color of the card.
4. **toString() Override:**
    - The `toString()` method is overridden to provide a custom string representation of the card.
    - The string representation includes the card's `value` followed by its `type`.

5. **OOP Principles:**
    - **Encapsulation:** The properties `type` , `value` , and `color` are encapsulated within the class, ensuring data integrity and controlling access through methods.
    - **Abstraction:** The class abstracts the concept of a playing card, providing only essential information such as type, value, and color.
    - **Inheritance:** The `Type` , `Value` , and `Color` enums are used, and they share a common inheritance from the base `Enum` class. This design promotes code reuse and consistency among different parts of the program.
    - **Polymorphism:** The `toString()` method is overridden, providing a polymorphic behavior to return a specific string representation of the card.
    - **Composition:** The `Card` class uses composition to associate the `Type` and `Value` enums, representing the properties of the card.

3.3 Deck

The `Deck` class is used to represent a deck of playing cards in the Old Maid Card Game. It contains a collection of `Card` objects and provides functionality to create, shuffle, and draw cards from the deck. The deck is shuffled at the beginning of the game to randomize the order of cards.

1. **Properties of the Deck class:**
    - `cards` (List of Card): Represents the collection of cards in the deck.
2. **Constructor:**
    - The constructor of the `Deck` class initializes the deck by creating instances of `Card` for each combination of `Type` and `Value` , excluding the Joker cards.
    - It populates the `cards` list with all the standard playing cards (52 cards) and adds one Joker card to the deck.
3. **Methods:**
    - `shuffle()`: This method shuffles the order of the cards in the deck using `Collections.shuffle()` to ensure randomness.
    - `isEmpty()`: This method checks if the deck is empty, i.e., it has no more cards left to draw.
    - `drawCard()`: This method removes and returns the last card from the deck (top of the deck). It simulates drawing a card from the deck.

4. **OOP Design:**
    - **Encapsulation:** The `Deck` class encapsulates the collection of cards within the class by making the `cards` list private. It provides controlled access to the deck's state through its public methods, preventing direct manipulation of the deck's internal state from outside the class.
    - **Abstraction:** The class abstracts away the details of how the deck is implemented and manipulated. Clients of the `Deck` class interact with it using its public methods, without needing to know the internal workings of the deck's representation.
    - **Modularity:** The `Deck` class handles deck-related operations independently. It is a self-contained module that can be reused in different parts of the game or even in other applications that require a standard deck of cards. The class provides a simple interface for managing decks, promoting code reusability and maintainability.

3.4 Player

The `Player` class represents a player in the Old Maid Card Game. Each player is a separate thread and has a hand of cards. The class handles player-specific actions, such as taking turns, picking cards from other players, removing matching pairs from the hand, and determining if the player has finished the game.

Properties:

- `playerName` (String): The name of the player, constructed as "Player {playerNumber}".
- `playerHand` (List of Card): A private list that holds the collection of `Card` objects representing the cards in the player's hand.
- `isFinished` (boolean): A boolean flag indicating whether the player has finished his cards or not.
- `nextPlayer` (Player): A reference to the next player in the game.
- `turnLock` (Object): An object used for synchronization to coordinate the turns between players.

Methods:

- `Player(int playerNumber)`: Constructor that initializes a player with a given player number. It sets the player name, initializes the player's hand, and sets the `isFinished` flag to false.
- `isFinished()` : Returns the `isFinished` flag, indicating if the player has finished the game.
- `setFinished(boolean finished)` : Sets the `isFinished` flag to the given value, indicating whether the player has finished the game or not.
- `setLock(Object turnLock)`: Sets the `turnLock` object to be used for synchronization during the game.
- `setNextPlayer(Player nextPlayer)`: Sets the `nextPlayer` reference to the next player in the game.
- `addCardToHand(Card card)`: Adds a card to the player's hand.
- `getHand()`: Returns a copy of the player's hand.
- `getSize()`: Returns the number of cards in the player's hand.
- `getPlayerName()`: Returns the name of the player.
- `removeMatchingPairs()`: Removes any matching pairs of cards from the player's hand.
- `printHand()`: Prints the player's current hand.
- `takeTurn()`: Represents the player's turn in the game. The player picks a random card from the next player, adds it to their hand, and removes any matching pairs.
- `checkIfFinished()`: Checks if the player has finished the game (hand is empty) and updates the `isFinished` flag accordingly.
- `run()`: The `run()` method that gets executed when the player thread starts. It represents the player's actions during the game, including taking turns, removing matching pairs, and notifying the next player to take their turn.

OOP Principles:

- **Encapsulation:** The `Player` class encapsulates the player's state and behavior. It keeps the player's hand private and provides methods to interact with it, preventing direct access from outside the class. This ensures controlled access and manipulation of player-specific information.
- **Abstraction:** The class abstracts away the complexity of handling a player in the game. It provides high-level methods like `takeTurn()` and `printHand()` to represent player actions without exposing internal details to the outside world.

- **Inheritance:** The `Player` class extends the `Thread` class to represent a separate thread for each player. This allows each player to run concurrently, simulating the players playing the game simultaneously.
- **Polymorphism:** The `Player` class overrides the `run()` method from the `Thread` class to provide custom logic for the player's actions during the game. By doing so, it takes advantage of the polymorphic behavior of Java threads, allowing each player's actions to be executed independently.
- **Single Responsibility Principle (SRP):** The `Player` class has a clear single responsibility: to represent a player in the game. It handles player-specific actions and does not have additional responsibilities that could lead to high coupling or code complexity.

3.5 GameManager

The `GameManager` class is the core of the Old Maid Card Game. It manages the flow of the game, handles player actions, and determines the winner. It extends the `Thread` class, allowing it to run as a separate thread in the game.

Properties:

- `Players` (List of Player): A list that holds all the players participating in the game.
- `deck` (Deck): An instance of the `Deck` class, representing the deck of cards used in the game.
- `turnLock` (Object): An object used for synchronization to coordinate the turns between players.

Methods:

- `GameManager(int numPlayers)`: Constructor that initializes the game manager with a given number of players. It creates the players, sets their turn lock, and initializes the deck.
- `dealCards()`: Deals the cards from the deck to the players, ensuring that all players have an equal number of cards. It also removes matching pairs from each player's hand at the beginning of the game.
- `startingPlayerThreads()`: Starts the threads for each player, allowing them to play the game concurrently.

- `run()`: The `run()` method that gets executed when the `GameManager` thread starts. It manages the game flow, ensuring that players take turns and the game continues until only one player is left.

OOP Principles:

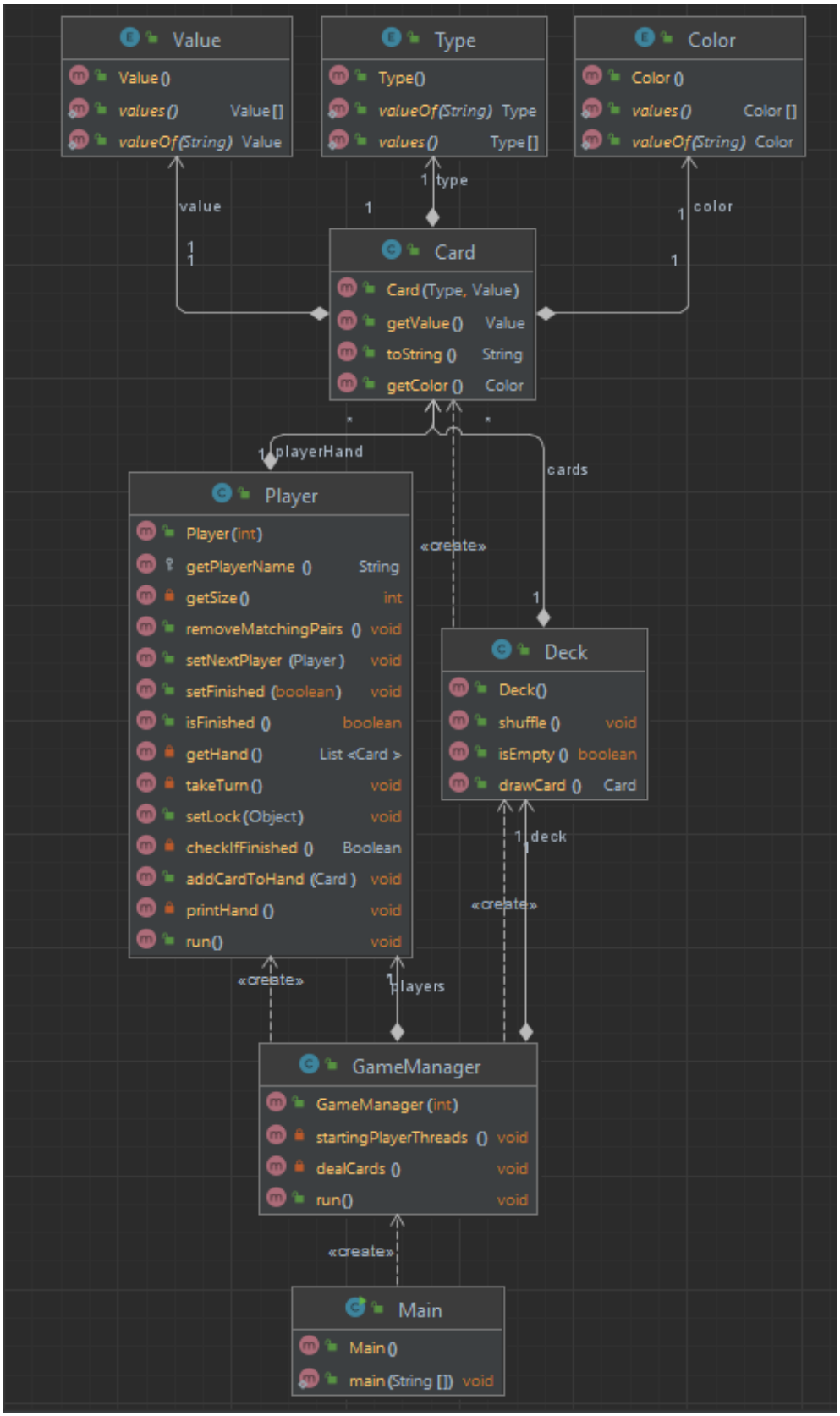
- **Encapsulation:** The `GameManager` class encapsulates the game's state and behavior, including players, deck, and turn synchronization. It provides methods to start the game, deal cards, and manage the game flow, while hiding the internal implementation details from the outside.
- **Abstraction:** The class abstracts away the complexity of managing the game. It provides high-level methods like `dealCards()` and `startingPlayerThreads()` to represent game actions without exposing internal details.
- **Inheritance:** The `GameManager` class extends the `Thread` class to represent the game manager as a separate thread. This allows the game to run concurrently with player threads, simulating the gameplay in real-time.
- **Polymorphism:** The `GameManager` class takes advantage of the polymorphic behavior of Java threads. Each player runs concurrently in their separate thread, and the game manager uses synchronization mechanisms (e.g., `wait()` and `notify()`) to coordinate their turns.
- **Single Responsibility Principle (SRP):** The `GameManager` class has a single responsibility: to manage the game. It handles game-specific actions and does not have unrelated responsibilities, promoting better code organization and readability.

3.6 Main

The `Main` class serves as the entry point of the Old Maid Card Game. It prompts the user to enter the number of players, creates an instance of the `GameManager`, starts the game, and waits for it to finish. Once the game is over, it prints "Game Over!" and terminates the program.

The `Main` class demonstrates good OOP design principles by keeping the main logic separate from other classes, providing a clean and simple entry point for the game, and adhering to the principles of encapsulation and abstraction.

Uml diagram for the Code:



4. Thread Synchronization Mechanism

In my code, thread synchronization mechanisms are employed to ensure that different threads (representing players) interact with shared resources (e.g., the game deck) in a coordinated and controlled manner. The primary synchronization mechanisms used in the code are `wait()` and `notify()`, along with the use of an `Object` (turnLock) as a synchronization point.

1. **wait() and notify() in Player class:**
    - In the `Player` class, the `wait()` and `notify()` methods are used to control the turn-based flow of the game.
    - The `run()` method inside the `Player` class represents the player's thread behavior.
    - When a player has finished their turn, they call `wait()`, which temporarily releases the lock on the current `Player` object and makes the thread wait for further notifications.
    - The `wait()` method is wrapped in a `synchronized` block to ensure that no other threads interfere with the player's turn and that the player does not proceed until they receive a notification.

2. **notify() in GameManager class:**
    - In the `GameManager` class, the `notify()` method is used to signal the waiting player to continue their turn.
    - The `run()` method inside the `GameManager` class represents the main game loop, which is responsible for coordinating player turns.
    - When a player has completed their turn, the `GameManager` notifies the waiting player to proceed by calling `notify()` on the corresponding player object.
    - The `notify()` method is also wrapped in a `synchronized` block to ensure proper synchronization and to prevent potential race conditions.

3. **turnLock (Object):**
    - The `turnLock` is an `Object` used as a synchronization point between the `GameManager` and `Player` objects.
    - The `turnLock` is passed to each `Player` during their initialization in the `GameManager` class, ensuring that all players have access to the same lock.
    - It is used to synchronize the player's turn and wait for the notification from the `GameManager` class when it's their turn to play.
    - The `synchronized` blocks on the player and the `turnLock` ensure that only one player proceeds at a time and that the game progresses smoothly.

4. **Synchronization in `GameManager` class:**
    - In the `GameManager` class, synchronization is used to ensure that players take their turns one at a time and avoid race conditions when accessing shared resources like the deck and player lists.
    - The `synchronized` block is used to obtain a lock on the player when it's their turn, preventing other players from executing their turns simultaneously.

In summary, the combination of `wait()`, `notify()`, and the `turnLock` object enables the implementation of turn-based game flow. The use of synchronization mechanisms ensures that players execute their turns in a controlled order, preventing conflicts and providing a coordinated gameplay experience.

5. Code Cleanness

My code showcases effective application of clean code principles, making it well-structured and readable. It follows standard coding practices, ensuring that the code is not only functional but also maintainable and scalable. Let's explore the clean code principles present in my implementation:

1. **Meaningful Naming:** I have utilized meaningful and self-explanatory names for classes, methods, and variables For example, `Player`, `Card`, `Deck`, and `GameManager`. This makes it easier for anyone reading the code to understand its purpose without diving deep into the implementation details.
2. **Encapsulation:**The `Player`, `Deck`, and `GameManager` classes have well-defined methods, and their internal state is encapsulated properly. This ensures that the internal details of these classes are hidden, and they can be used as black boxes by other parts of the code.
3. **Single Responsibility Principle (SRP):**Each class seems to have a single responsibility, which is a good practice. For instance, the `Player` class is responsible for managing a player's hand and taking turns, while the `Deck` class handles card management.
4. **Use of Enums:** By employing enums for representing card attributes, I have organized constant values effectively. This enhances code readability and prevents the use of ambiguous or hard-coded values.
5. **Avoidance of Magic Numbers:** My code refrains from using magic numbers directly, as I have used named constants for parameters and other values. This not only improves readability but also allows for easy modification if needed.

6. **Code Formatting:** My code is well-formatted and consistently indented, making it aesthetically pleasing and easier to follow.
7. **Thread Safety Consideration:** I have demonstrated awareness of thread safety by employing synchronization mechanisms such as `wait()` and `notify()` to manage player turns. This ensures the smooth functioning of the game in a multi-threaded environment.
8. **Object Reusability:** The implementation of separate classes for `Player`, `Deck`, and `GameManager` promotes reusability and modularity. Each class can be utilized independently, allowing for easy extension or modification without affecting the rest of the codebase.

In conclusion, my code exemplifies the application of clean code principles, resulting in a well-organized and maintainable implementation. The meaningful naming, encapsulation, and modularity contribute to code readability and clarity. Furthermore, the consideration of thread safety ensures smooth execution in multi-threaded scenarios. With the current clean code practices in place, my code is set to be easily understood, extended, and enhanced.