The Old Maid Card Game

RAWAN AFANEH

The Old Maid Card Game

In this assignment, you are required to use Java Multithreading to simulate the old maid card game, in which each player is represented as a thread that plays the game.

- 'OldCardGame' Class:

```java
import java.util.Scanner;
public class OldCardGame {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("WELCOME TO THIS CARD GAME!");
        System.out.println("how many palyers will play:");
        Scanner sc = new Scanner(System.in);
        int numPlayers = sc.nextInt();
        while (numPlayers<2){
            System.out.println("the game must have at least 2 players!");
            System.out.println("please enter the number again:");
            numPlayers = sc.nextInt();
        }
        Game game = new Game(numPlayers);
        game.startGame();
    }
}
```

The **OldCardGame** class is the starting point of the card game application, responsible for interacting with the user to initialize the game and begin its execution.

main() Method:

The **main()** method represents the entry point of the application. It carries out the following steps:

1. Displays a welcome message to the user, indicating the start of the card game.
2. Prompts the user to enter the number of players participating in the game using the **Scanner** class.
3. Implements a loop to ensure that the number of players is valid. If the entered value is less than 2, the program displays an error message and prompts the user to enter the number of players again until a valid value is provided.
4. Creates an instance of the **Game** class, passing the valid number of players as an argument.
5. Initiates the game's execution by invoking the **startGame()** method of the **Game** instance.

- ‘Player’ Class:

```java
import java.util.ArrayList;
import java.util.List;

public class Player implements Runnable {
    private int playerId;
    private List<Card> hand;
    private Game game;
    private boolean finished;
    private static final int MOVE_DELAY = 1500;

    public Player(int playerId, Game game) {
        this.playerId = playerId;
        this.hand = new ArrayList<>();
        this.game = game;
        this.finished = false;
    }

    @Override
    public void run() {

        while (true) {

            synchronized (game.getTurnLock()) {
                if(game.getPlayers().size()==1){
                    break;
                }
                while (this != game.getCurrentPlayer()) {
                    try {
                        game.getTurnLock().wait();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return;
                    }
                }

                if (hand.isEmpty()) {
                    setFinished(true);
                    game.removePlayer(this);
                }else {
                    if(game.getPlayers().size()==1){
                        break;
                    }
                    performTurn();
                }

                game.nextTurn();

                game.getTurnLock().notifyAll();
            }
        }
    game.getThreadPool().shutdown();
        System.exit(0);
    }


    private synchronized void performTurn() {

        System.out.println("________________________________________________________");
        System.out.println("it's player "+this.getPlayerId()+"'s turn");
        System.out.println("Player has :"+this.getHand());



        Player nextPlayer = game.getNextPlayer(this);
        Card drawnCard = nextPlayer.drawRandomCard();
        if (drawnCard != null) {
            System.out.println("the Player "+this.playerId+" pick randomly card from Player
"+nextPlayer.playerId);
            addToHand(drawnCard);
            System.out.println("the drawn card is: "+ drawnCard);
        }
        System.out.println("Checking if there any new pairs....");
        this.throwMatchingPairs();
        try {
            Thread.sleep(MOVE_DELAY);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
    public void addToHand(Card card) {
        hand.add(card);
    }

    public Card drawRandomCard() {
        if (!hand.isEmpty()) {
            int randomIndex = (int) (Math.random() * hand.size());
            return hand.remove(randomIndex);
        }
        return null;
    }

    public List<Card> getHand() {
        return hand;
    }

    public void throwMatchingPairs() {
        List<Card> pairsToDiscard = new ArrayList<>();

        for (int i = 0; i < hand.size(); i++) {
            for (int j = i + 1; j < hand.size(); j++) {
                if ((hand.get(i).getRank().equals(hand.get(j).getRank() )) && (
matchingSuits(hand.get(i).getSuit(), hand.get(j).getSuit()))) {

                    pairsToDiscard.add(hand.get(i));
                    pairsToDiscard.add(hand.get(j));
                }
            }
        }

        if(pairsToDiscard.size()!=0){
            hand.removeAll(pairsToDiscard);
            System.out.println("the pairs cards removes");

        }
        else{
            System.out.println("there is no pairs");
        }

    }

    private boolean matchingSuits(CardSuit suit1, CardSuit suit2) {
        return (suit1 == CardSuit.SPADES && suit2 == CardSuit.CLUBS) ||
                (suit1 == CardSuit.CLUBS && suit2 == CardSuit.SPADES) ||
                (suit1 == CardSuit.DIAMONDS && suit2 == CardSuit.HEARTS) ||
                (suit1 == CardSuit.HEARTS && suit2 == CardSuit.DIAMONDS);
    }

    public int getPlayerId() {
        return playerId;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

    public boolean isFinished() {
        return finished;
    }

}
```

The Player class embodies the behavior and attributes of a player participating in the card game. It encapsulates the logic for a player's turn, managing their hand of cards, and interacting with the game.

Attributes (playerId, hand, game, finished, MOVE_DELAY): The class holds several attributes that define a player's state and interactions. These include the player's unique identifier (playerId), a list of cards in the player's hand (hand), a reference to the game instance (game), a flag indicating whether the player has finished their turn (finished), and a constant representing the time delay for each move (MOVE_DELAY).

Constructor: The class constructor initializes a Player instance with a unique player ID and a reference to the game. It also initializes an empty list for the player's hand and sets the finished flag to false.

run() Method (Runnable Implementation):

This method represents the execution of a player's turn within a separate thread. It operates within an infinite loop, allowing the player to take turns repeatedly until the game ends. The loop is synchronized on the turnLock object obtained from the game, ensuring orderly and synchronized player turns.

Within the loop, the method checks if the game has only one player remaining, and if so, it exits the loop to end the thread. Otherwise, it waits until it is the current player's turn by repeatedly calling wait() on the turnLock until the player's turn arrives.

Once it is the player's turn, the method checks if the player's hand is empty. If it is, the player marks themselves as finished and is removed from the game using the removePlayer() method. Otherwise, the player's turn logic is executed by calling the performTurn() method.

After completing a turn, the method notifies all waiting threads and advances to the next player's turn using the nextTurn() method.

performTurn() Method:

This method encapsulates the logic of a player's turn. It displays information about the player's turn, their current hand, and then proceeds to perform the turn's actions, including drawing a random card from the next player, adding it to the player's hand, and checking for matching pairs using the throwMatchingPairs() method.

Card Interaction Methods (addToHand(), drawRandomCard(), getHand(), throwMatchingPairs()):

These methods manage the player's interaction with their hand of cards. The addToHand() method adds a card to the player's hand, the drawRandomCard() method randomly selects and removes a card from the hand, the getHand() method retrieves the player's current hand of cards, and the throwMatchingPairs() method identifies and removes pairs of cards with matching ranks and suits.

matchingSuits() Method:

This private method checks whether two given card suits match each other, specifically checking for combinations of suits that indicate a matching pair (e.g., SPADES and CLUBS).

Other Utility Methods (getPlayerId(), setFinished(), isFinished()):

These methods provide access to the player's ID, allow modification of the finished flag, and check whether the player has finished their turn.

- ‘Game’ Class

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Game {

    private GameRule gameRules;
    private static final int MOVE_DELAY = 1500;

    public List<Player> getPlayers() {
        return players;
    }

    private List<Player> players;
    private Deck deck;


    private ExecutorService threadPool;
    private int currentPlayerIndex;
    private final Object turnLock = new Object();

    public Game(int numPlayers) {
        players = new ArrayList<>();
        deck = new Deck();
        threadPool = Executors.newFixedThreadPool(numPlayers);
        gameRules = new GameRule(this,players,deck);

        for (int i = 0; i < numPlayers; i++) {
            Player player = new Player(i, this);
            players.add(player);
        }
    }

    public void startGame() {
        gameRules.dealCards();
        startPlayerThreads();
        startFirstTurn();
        waitForGameToFinish();
        threadPool.shutdown();

    }
    private void waitForGameToFinish() {
        while (!isGameOver()) {

        }
    }
    private void startFirstTurn() {
        synchronized (turnLock) {
            currentPlayerIndex = 0;
            turnLock.notifyAll();
        }
        try {
            Thread.sleep(MOVE_DELAY);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public synchronized Player getCurrentPlayer() {
        return players.get(currentPlayerIndex);
    }

    public synchronized void removePlayer(Player player) {
        players.remove(player);
        if (currentPlayerIndex >= players.size()) {
            currentPlayerIndex = 0;
        }
    }

    public synchronized Player getNextPlayer(Player currentPlayer) {
        int nextPlayerIndex = (currentPlayerIndex + 1) % players.size();
        return players.get(nextPlayerIndex);
    }

    public synchronized void nextTurn() {
        currentPlayerIndex = (currentPlayerIndex + 1) % players.size();
    }

    public synchronized  Object getTurnLock() {
            return turnLock;
        }

        public Deck getDeck() {
            return deck;
        }

        public synchronized boolean isGameOver() {
            return gameRules.isGameOver();

        }

        private void startPlayerThreads() {
            for (Player player : players) {
                threadPool.execute(player);
            }
        }


        public synchronized ExecutorService getThreadPool() {
            return threadPool;
        }

    }
```

The **Game** class serves as the central orchestrator of the card game, coordinating players, turns, and the game's progression through its various stages.

Attributes (gameRules, players, deck, threadPool, currentPlayerIndex, turnLock): The class maintains key attributes to facilitate game control. These include references to the **GameRule** instance (**gameRules**), the list of player instances (**players**), the card deck (**deck**), a thread pool for player threads (**threadPool**), the index of the current player taking a turn (**currentPlayerIndex**), and a synchronization lock (**turnLock**) to manage orderly player turns.

Constructor: The class constructor initializes a game instance by creating players, associating a deck, and setting up the **GameRule**. Each player is assigned a unique identifier, and the **GameRule** is provided with references to the game, players, and deck.

startGame() Method:

This method initiates the card game's execution by performing the following steps:

1. Dealing cards to players using the **dealCards()** method of the **GameRule**.
2. Launching player threads using the **startPlayerThreads()** method to enable concurrent player actions.
3. Starting the first turn using the **startFirstTurn()** method to ensure that players take turns in an orderly manner.
4. Waiting for the game to finish by repeatedly checking the game's state using the **waitForGameToFinish()** method.
5. Shutting down the thread pool to terminate player threads after the game concludes.

waitForGameToFinish() Method:

This method employs a loop to wait until the game is over. It continuously checks the **isGameOver()** method of the **GameRule** to determine whether the game has reached its conclusion. Once the game is over, the loop exits, and the method returns.

startFirstTurn() Method:

This method initiates the first turn of the game by synchronizing on the **turnLock** object. It sets the **currentPlayerIndex** to 0, representing the first player, and notifies all waiting threads. This ensures that the first player can proceed with their turn while other players wait.

Turn Management Methods (getCurrentPlayer(), removePlayer(), getNextPlayer(), nextTurn()):

These methods manage the turn-based nature of the game. The **getCurrentPlayer()** method retrieves the current player based on the **currentPlayerIndex**. The **removePlayer()** method removes a player from the game, adjusting the current player index if necessary. The **getNextPlayer()** method calculates the index of the next player, enabling the game to proceed to the next turn. The **nextTurn()** method updates the **currentPlayerIndex** to switch to the next player's turn.

getTurnLock() Method: This method provides access to the **turnLock** object, allowing other classes to synchronize their actions with the turn management system.

Integration with GameRule and Player Threads: The **Game** class collaborates closely with the **GameRule** class to enforce game rules and manage player actions. It uses a thread pool to execute player threads concurrently, enabling players to take turns in parallel while adhering to synchronization mechanisms.

- ‘GameRule’ Class:
-
```java
import java.util.ArrayList;
import java.util.List;

public class GameRule {
    private Game game;
    private List<Player> players;
    private Deck deck;
    public GameRule(Game game ,List<Player> players,Deck deck) {
        this.game = game;
        this.players=players;
        this.deck=deck;
    }

    public void dealCards() {
        int totalCards = game.getDeck().getSize();
        int cardsPerPlayer = totalCards / players.size();
        int extraCards = totalCards % players.size();

        for (Player player :players) {
            int numCardsToDeal = cardsPerPlayer;


            if (extraCards > 0) {
                numCardsToDeal++;
                extraCards--;
            }

            for (int i = 0; i < numCardsToDeal; i++) {
                Card card = deck.drawCard();
                player.addToHand(card);
            }
        }

        System.out.println("Each player has at least " + cardsPerPlayer + " cards");

        for (Player player : players) {
            player.throwMatchingPairs();
        }

        System.out.println("All players threw all the matching pairs");
    }

    public synchronized boolean isGameOver() {
        List<Player> playersToRemove = new ArrayList<>();

        for (Player player : players) {
            if (!player.isFinished() && player.getHand().isEmpty()) {
                player.setFinished(true);
                playersToRemove.add(player);
            }
        }

        players.removeAll(playersToRemove);

        if (players.size() == 1) {
System.out.println("________________________________________________________________
_______");
System.out.println("________________________________________________________________
_______");
            System.out.println("PLAYER " +players.get(0).getPlayerId() +" THE FINAL
ONE WHO HAS THE JOKER! Player "+ players.get(0).getPlayerId() + " IS THE LOSER ");
System.out.println("________________________________________________________________
_______");
System.out.println("________________________________________________________________
_______");
            return true;
        }

        return false;
    }
}
```

The **GameRule** class plays a pivotal role in orchestrating the rules and progression of the card game. It encapsulates logic related to dealing cards, checking for matching pairs, and determining the end of the game.

Attributes (game, players, deck): The class holds references to the **Game** instance, the list of **Player** objects, and the **Deck** instance. These attributes establish a vital connection between the game's components, allowing the **GameRule** to interact with players and the deck.

Constructor: The class constructor initializes the **GameRule** instance with references to the game, players, and deck. This constructor injection ensures that the **GameRule** operates in the context of a specific game and its participants.

dealCards() Method:

This method handles the distribution of cards to players at the beginning of the game. It calculates the number of cards each player should receive (**cardsPerPlayer**) based on the total number of cards in the deck and the number of players. The division includes an allocation for any remaining cards (**extraCards**).

For each player, the method iterates through the deck, dealing cards up to numCardsToDeal for each player. After distributing cards, it triggers each player's throwMatchingPairs() method to discard any matching pairs from their hands.

The method concludes by displaying information about the card distribution and the removal of matching pairs.

isGameOver() Method:

This synchronized method determines whether the game has concluded. It evaluates the state of each player's hand to identify if any player has finished their cards and, consequently, should be removed from the game. If a player's hand is empty and they are not marked as finished, the method sets them as finished and adds them to the playersToRemove list.

After processing each player, the method removes all players from the players list who have been marked as finished. If only one player remains, a message is displayed indicating that this player is the final one holding the Joker, thus making them the loser of the game. In such a scenario, the method returns true to signal the end of the game. Otherwise, it returns false.

- Card, CardRank, and CardSuit Classes:

1.
```
public class Card {
    private final CardSuit suit;
    private final CardRank rank;

    public Card(CardSuit suit, CardRank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public CardSuit getSuit() {
        return suit;
    }

    public CardRank getRank() {
        return rank;
    }

    @Override
    public String toString() {
        return rank + " of " + suit;
    }
}
```

2.
```
public enum CardRank {
    ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK,
QUEEN, KING, JOKER

}
```

```
3. public enum CardSuit {
       SPADES, HEARTS, DIAMONDS, CLUBS
   }
```

The Card class forms the foundation of representing individual playing cards within the card game. This class encapsulates the attributes and behavior of a single card, allowing for seamless interaction and manipulation within the game.

- Attributes (suit and rank): The class defines two private final attributes, suit and rank, which represent the suit and rank of the card, respectively. These attributes are set during card creation and remain immutable throughout the card's lifecycle.
- Constructor: The class includes a constructor that initializes a card with a specified suit and rank. This constructor enforces the immutability of the card's attributes, preventing any modifications after creation.
- Accessor Methods (getSuit() and getRank()): The class provides public accessor methods to retrieve the suit and rank of the card. These methods encapsulate the card's attributes, allowing external classes to access the card's properties without direct manipulation.
- toString() Method Override: The toString() method is overridden to provide a customized string representation of the card. This representation includes the rank and suit of the card, enhancing human-readable output when printing card information.

2. CardRank Enum:

The CardRank enum defines the possible ranks that a playing card can have. This enum contributes to code clarity by providing a clear and consistent representation of card ranks.

- Enum Values: The enum includes values representing standard card ranks, ranging from ACE to KING, along with an additional value, JOKER. These values are semantically meaningful and provide a comprehensive representation of card ranks.

3. CardSuit Enum:

The CardSuit enum enumerates the various suits that a playing card can belong to. Similar to the CardRank enum, this enum enhances code readability and organization by defining a consistent set of card suits.

- Enum Values: The enum consists of four suit values: SPADES, HEARTS, DIAMONDS, and CLUBS. These values reflect the standard suits found in a deck of playing cards and enable a structured representation of card suits.

Integration and Usage:

The Card, CardRank, and CardSuit classes collectively work together to create a comprehensive representation of playing cards. The Card class leverages the CardRank and CardSuit enums to capture a card's specific attributes, such as its rank and suit. This integrated approach simplifies card creation, manipulation, and display throughout the card game.

In summary, the Card, CardRank, and CardSuit classes exemplify fundamental object-oriented design principles, including encapsulation, abstraction, and consistency. Their modular structure contributes to a well-organized codebase and facilitates the development of a fully functional and interactive card game.

a. Object-Oriented Design in Your Code:

The implementation of the card game exhibits strong adherence to fundamental object-oriented design principles, reflecting a well-structured and organized codebase. The following aspects highlight the application of object-oriented design:

Abstraction and Encapsulation:

The code employs encapsulation to encapsulate complex functionalities within dedicated classes, enhancing code modularity and readability. For instance, the Card, CardRank, and CardSuit classes encapsulate the representation and attributes of playing cards, providing a high-level interface for card-related operations while hiding intricate details.

Inheritance (Enums):

Inheritance is effectively employed through the use of enum types. Specifically, the CardRank and CardSuit enums serve as subclasses of the Enum class, allowing for a clear representation of possible card ranks and suits. This hierarchical structure promotes code organization, consistency, and ease of maintenance.

Composition (Game Composition):

The Game class showcases composition by aggregating various components, such as Player, Deck, and GameRule. This composition fosters a modular design approach, facilitating the separation of concerns and enabling efficient development and modification of individual components.

Polymorphism (Overriding):

Polymorphism is demonstrated through method overriding, enhancing code flexibility and extensibility. For instance, the Card class overrides the toString() method to provide a customized string representation of card attributes. This enables consistent and customizable behavior across various card instances.

Single Responsibility Principle (SRP):

The code adheres to the Single Responsibility Principle, with each class fulfilling a distinct role and responsibility. A notable example is the Player class, which effectively encapsulates player-specific actions, ensuring a focused and maintainable design.

High Cohesion and Low Coupling:

The code showcases high cohesion by grouping related functionalities within classes, leading to a clear and concise structure. Simultaneously, low coupling is maintained, reducing interdependencies between classes and promoting code reusability and ease of modification.

b. Thread Synchronization Mechanisms:

The code employs robust thread synchronization mechanisms to ensure seamless interaction between concurrent player threads and maintain a consistent game state. Key synchronization mechanisms include:

synchronized Blocks:

The code uses synchronized blocks to synchronize access to critical sections of code. An exemplar is the run() method within the Player class, where synchronized blocks ensure orderly player turns, preventing concurrent execution and maintaining game integrity.

wait() and notifyAll():

The wait() and notifyAll() methods serve as essential tools for thread coordination. The wait() method allows player threads to voluntarily pause, ensuring they await their turn before proceeding. Conversely, the notifyAll() method signals waiting threads, allowing the next player to take their turn.

turnLock Object:

The turnLock object plays a pivotal role in facilitating thread synchronization. By leveraging this synchronization primitive, the code guarantees sequential player turns, mitigating potential race conditions and conflicts arising from concurrent execution.

c. Defending Against Clean Code Principles (Uncle Bob):

The code exhibits a commendable adherence to Clean Code principles; however, a few areas warrant consideration for further alignment:

Method Length:

While the code maintains a well-organized structure, some methods, particularly the run() method in the Player class, could benefit from being subdivided into smaller, focused methods. This approach enhances code readability and comprehension by isolating specific actions within distinct methods.

Variable Naming:

To promote code clarity, ensure that variable names explicitly convey their purpose and usage. For instance, opting for more descriptive names, such as playersToRemove, fosters improved code readability and comprehension.

Dependency Inversion Principle (DIP):

Constructor Injection: Dependencies are passed into classes through their constructors. For instance, the Game class receives instances of GameRule, Player, and Deck through constructor injection. This approach promotes loose coupling between classes and facilitates easier testing, maintenance, and future changes.

By following DIP, the application's components are decoupled, making the codebase more resilient to changes and ensuring a higher level of flexibility and reusability.

Single Responsibility Principle (SRP):

This principle is evident in the design of the card game application's classes:

Card class: It is responsible for representing a playing card, encapsulating its suit, rank, and behavior. The class's sole purpose is to provide methods for retrieving card properties and displaying card information.

Deck class: This class handles the management of a deck of cards, including initialization, shuffling, and drawing. Its responsibility is confined to the management of card-related operations.

Player class: The Player class represents an individual player and encapsulates methods for managing the player's hand, performing turns, and identifying matching pairs. It focuses solely on player-related actions within the game.

GameRule class: This class defines the game rules, including dealing cards and determining the game's end. It is dedicated to managing game-specific logic and rules.

Game class: The Game class orchestrates the overall game flow, managing players, turns, and thread synchronization. Its primary responsibility is to facilitate the gameplay and coordinate interactions.

Open/Closed Principle (OCP):

This principle is evident in the design of the card game application's card-related classes:

CardRank and CardSuit enums: These enumerations define the possible card ranks and suits. Adding new card ranks or suits can be achieved without altering existing code. The enums are open for extension through the addition of new enum constants, without requiring changes to existing behavior.

Liskov Substitution Principle (LSP):

The card game application demonstrates LSP in the following way:

CardRank and CardSuit enums: Instances of these enums can be seamlessly used interchangeably with their parent enums, promoting polymorphism. For instance, any method that expects a CardRank or CardSuit can work with any valid enum value without causing errors.