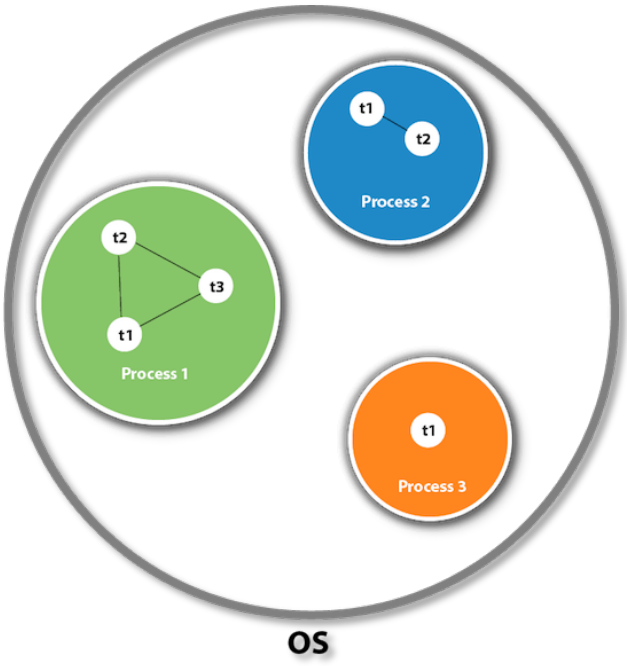
ATYPON

Old Maid Game Assignment Report

Fares Qawasmi

Dr. Fahed Jubair



Abstract:

This report presents a Java Multithreading implementation of the Old Maid card game, where players are represented as separate threads. The objective is to discard all cards by forming matching pairs, while the player holding the Joker at the end loses. To ensure efficient use of CPU resources, the implementation employs the wait/notify mechanism to put player threads in the "wait" state when it is not their turn to play. This approach allows the simulation to proceed correctly while minimizing unnecessary CPU utilization. The assignment emphasizes OOP design in addition to Java Multithreading, resulting in an engaging and efficient simulation of the Old Maid card game. The code basically sets up player threads, deals cards, and handles player interactions using the wait/notify mechanism, and the main thread oversees the game and reports the results at the end.

Object-Oriented Design:

◊ Encapsulation: Think of a class as a capsule that contains both data and methods. Encapsulation ensures that the internal workings of the capsule are hidden from the outside world, and only specific methods are accessible. In my code, the *Card* class encapsulates the details of a card, and the *Deck* class hides the complexities of managing a deck of cards. Encapsulating data and behavior promote information hiding and prevents direct access to the internal state, providing a more secure and manageable way to interact with objects.

```
2 usages
public class Deck {
    6 usages
    private List<Card> cards;
    1 usage
    public Deck() {
        cards = new LinkedList<>();
        initializeDeck();
    }
    1 usage
    private void initializeDeck() {
        String[] suits = {"♠", "♥", "♣", "♦"};
        String[] values = {"2", "3", "4", "5",
                           "6", "7", "8", "9", "10",
                           "Jack", "Queen", "King", "Ace"};

        for (String suit : suits) {
            for (String value : values) {
                Card card = new Card(value, suit,  isJoker: false);
                cards.add(card);
            }
        }
        cards.add(new Card( value: "Joker",  suit: "",  isJoker: true));
    }
    1 usage
    public void shuffle() { Collections.shuffle(cards); }
    1 usage
    public List<Card> deal(int numCards) {
        List<Card> dealtCards = new ArrayList<>();
        for (int i = 0; i < numCards && !cards.isEmpty(); i++) {
            dealtCards.add(cards.remove( index: 0));
        }
        return dealtCards;
    }
}
```

```
public class Card {
    3 usages
    private String value;
    3 usages
    private String suit;
    3 usages
    private boolean isJoker;
    2 usages
    public Card(String value, String suit, boolean isJoker) {
        this.value = value;
        this.suit = suit;
        this.isJoker = isJoker;
    }
    2 usages
    public String getValue() {
        return value;
    }
    2 usages
    public String getSuit() {
        return suit;
    }
    3 usages
    public boolean isJoker() {
        return isJoker;
    }
    @Override
    public String toString() {
        if (isJoker) {
            return "Joker";
        }
        return value + " of " + getSymbolForSuit(suit);
    }
    1 usage
    private String getSymbolForSuit(String suit) {
        switch (suit) {
            case "Spades":
                return "♠";
            case "Hearts":
                return "♥";
            case "Clubs":
                return "♣";
            case "Diamonds":
                return "♦";
            default:
                return suit;
        }
    }
```

◊ Abstraction: Imagine abstracting away the complexities of an object and presenting a simplified view. For example, when you play a card game, you don't need to know the intricate details of a card's design; you only care about its value and suit. In my code, each class represents a real-world concept (like a card, deck, or player), providing a high-level interface to interact with them, abstracting away the underlying implementation.

◊ Inheritance: the *Player* class inherits the properties of the Thread class, gaining the ability to run as a separate *thread*.

◊ Single Responsibility Principle (SRP): SRP states that a class should have a single responsibility and reason to change. Each of the classes focuses on a specific task:

- Card class: Represents a card and provides information about it.
- Deck class: Manages the deck of cards, including shuffling and dealing.
- Player class: Represents a player, handles the player's hand, and interactions during the game.
- OldMaidGame class: Orchestrates the game logic, initializes players, and manages the game flow.

Thread Synchronization Mechanisms:

Thread synchronization is crucial in a multithreaded environment to ensure that threads interact in an orderly and coordinated manner, preventing race conditions, and ensuring correct behavior. In my code, thread synchronization is achieved using Java's wait/notify mechanism, allowing players to take turns and interact correctly during the Old Maid game.

◊ The *Player* class represents each player in the game and extends the Thread class. In the run() method of the *Player* class, each player thread takes its turn by waiting for their chance to play using the wait() method.

- When a player's turn is over, they notify the next player to take their turn using the notify() method. This ensures taking turns in a sequential manner, rather than all players playing simultaneously.

```
@Override
public void run() {
    int index = players.indexOf(this);
    Player currentPlayer;

    while (running && !isGameOver) {
        if (currentPlayerIndex.get() == index) {
            currentPlayer = players.get(index);

            synchronized (currentPlayer) {
                if (hand.size() == 0) {
                    break;
                }

                int nextPlayerIndex = (index + 1) % players.size();
                Player nextPlayer = players.get(nextPlayerIndex);

                int randomCardIndex = -1;
                if (nextPlayer.getHandSize() > 0) {
                    randomCardIndex = new Random().nextInt(nextPlayer.getHandSize());
                }

                if (randomCardIndex != -1) {
                    Card randomCard = nextPlayer.hand.remove(randomCardIndex);
                    currentPlayer.receiveCard(randomCard);

                    currentPlayer.discardPair();

                    System.out.println(currentPlayer.name + " took one card from " + nextPlayer.getPName());
                    System.out.println("Card taken: " + randomCard);
                    printCurrentHand();
                }
                currentPlayerIndex.set((currentPlayerIndex.get() + 1) % players.size());
                currentPlayer.notify();
            }
        }
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (this) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

◊ The *OldMaidGame* class acts as the orchestrator of the game. After creating and starting player threads, the main thread waits for all player threads to finish before printing the game results using the join() method. This synchronization ensures that the main thread doesn't proceed to print the results until all player threads have completed their turns and the game is over.

```
public class OldMaidGame {
    4 usages
    private static int playerCount;
    17 usages
    private static List<Player> players;

    public static void main(String[] args) {
        Deck deck = new Deck();
        deck.shuffle();

        players = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter Number of Players: ");
        playerCount = scanner.nextInt();
        scanner.nextLine();

        for (int i = 1; i <= playerCount; i++) {
            System.out.print("Enter name for Player " + i + ": ");
            String playerName = scanner.nextLine();
            players.add(new Player(playerName));
        }

        int cardsPerPlayer = 53 / playerCount;
        for (Player player : players) {
            List<Card> dealtCards = deck.deal(cardsPerPlayer);
            for (Card card : dealtCards) {
                player.receiveCard(card);
            }
        }

        Collections.shuffle(players);

        Player.initializePlayers(players);

        for (Player player : players) {
            player.start();
        }
```

```
for (Player player : players) {
    try {
        player.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

◊ The currentPlayerIndex is used to keep track of the index of the current player during the game. The variable is shared among multiple player threads and the main thread. Using AtomicInteger ensures that the currentPlayerIndex is updated atomically, and any changes made to it are immediately visible to other threads, preventing race conditions and maintaining correct behavior during the Old Maid game. It is initially set to -1 for a specific purpose: to represent that no player has started their turn yet. By setting it to -1, the game ensures that the first player thread (Player 0) will be the first one to start the game.

```
4 usages
private static AtomicInteger currentPlayerIndex = new AtomicInteger( initialValue: -1);
```

Clean Code Principles (Uncle Bob):

◊ Single Responsibility Principle (SRP): The code structure exhibits a focus on single responsibility, with classes designed to handle specific tasks such as managing the game state, implementing game rules, or controlling the flow of the game. *Card* class manages individual cards, *Deck* class handles the deck of cards, *Player* class represents the players in the form of threads, and *OldMaidGame* class coordinates the game. Each class focuses on its specific tasks, promoting maintainability.

◊ Encapsulation: The code demonstrates encapsulation by using private access modifiers for class members and providing public getter methods to access necessary information. This approach hides implementation details, allowing for easier future modifications without affecting the external interface.

◊ Readability and Naming: The code uses descriptive and meaningful variable and method names, contributing to its readability. The getHandSize(), receiveCard(), discardPair(), and printCurrentHand() methods in the *Player* class are well-named and provide clarity to their purposes.

◊ Code Formatting: The code follows a consistent and organized format, using proper indentation and spacing. This enhances readability and reduces cognitive load when understanding the flow of the program.

Overview:

- ◊ Card Class: The *Card* class represents a playing card and encapsulates its properties such as value, suit, and if it is a Joker. The class appropriately uses private data members and public getter methods to provide controlled access to the card information.

- ◊ Deck Class: The *Deck* class represents a deck of playing cards and initializes the deck, shuffles it, and deals cards to players. It also employs encapsulation by keeping the list of cards private and providing methods to perform operations on the deck.

- ◊ Player Class: The *Player* class represents a player in the game, with a name and a hand of cards. It extends the Thread class, allowing each player to run as a separate thread. The class appropriately uses private data members and getter methods to access the player's information. It also contains methods to receive cards, check for matching pairs, and discard pairs from the player's hand.

- ◊ OldMaidGame Class: The *OldMaidGame* class orchestrates the game, initializes players, and manages the game loop until a winner is determined. It follows the object-oriented design principle of separation of concerns by handling different aspects of the game in separate methods.