ATYPON

Instructors:
Fahed Jubair
Motasim Aldiab

done by:

Hamza Hassan

The Old Maid Card Game

ABSTRACT

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

Introduction

In this report, we present the design and implementation of an automated Old Maid card game using Java multithreading and object-oriented programming (OOP) principles. The objective of the project was to create a simulation of the game with multiple players, each represented as a separate thread. The game follows specific rules for card matching and discarding, and the players take turns in a synchronized manner.
This report will provide insights into the object-oriented design approach, the thread synchronization mechanisms employed, and how the code aligns with clean code principles.

Object-oriented design (OOP)

The game contain four classes :

1. Card Class:
   - Represents cards and their attributes (suit and value).
   - Contains a method (isMatchingPair()) to check if two cards form a matching pair.
   - Used within the Deck class to create cards.

2. Deck Class:
   - use singleton design pattern to create deck

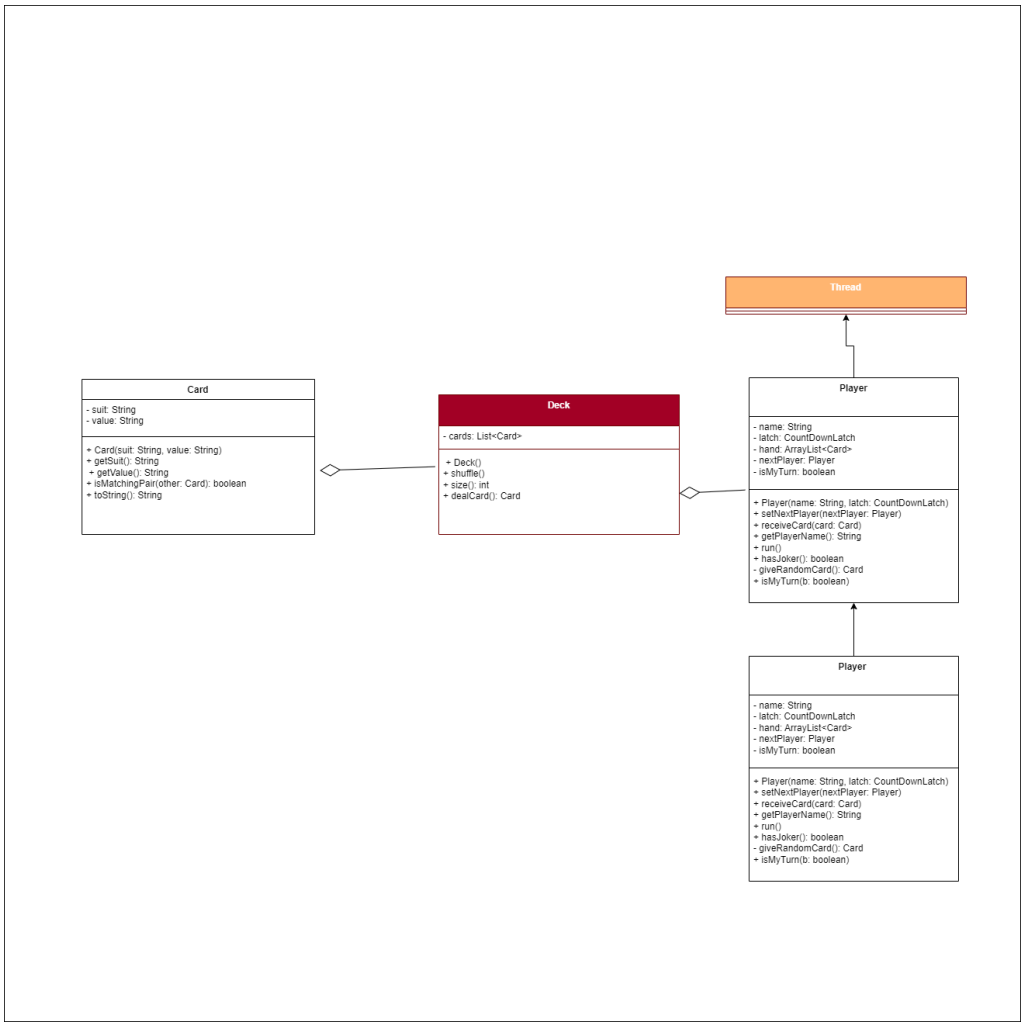
```
private Deck() {
    cards = new ArrayList<>();
    // Add standard 52 cards to the deck
    String[] suits = {"Spades", "Clubs", "Diamonds", "Hearts"};
    String[] values = {"2", "3", "4", "5", "6", "7", "8", "9",
    for (String suit : suits) {
        for (String value : values) {
            cards.add(new Card(suit, value));
        }
    }
    // Add the Joker
    cards.add(new Card( suit: "Joker",  value: "Joker"));
    System.out.println(cards.size());
}

1 usage
public static Deck getInstance() {
    if (instance == null) {
        instance = new Deck();
        instance.shuffle();
    }
    return instance;
}
```

   - Creates and initializes the deck by populating it with cards
   - has Aggregation with Card class
   - Provides methods to shuffle and deal cards

3. Player Class:
   - Extends the Thread class, representing a player in the game as a separate thread.
   - Has an association with the Card class, as it holds a collection of Card objects in its hand.
   - Employs synchronization mechanisms to control turn-based gameplay and card exchange.

4. OldMaidGame Class:
   - Contains the main method.
   - Creates instances of Player and Deck classes.
   - Ensures that players' threads are started and managed.



Multithreading mechanism

when the game start and the user enter the number of players in game, the game involves several players, each represented by a separate Player thread added to the array list of players

```
ArrayList<Player> players = new ArrayList<>();
for (int i = 1; i <= numPlayers; i++) {
    players.add(new Player( name: "Player " + i, latch));
}

for (int i = 0; i < numPlayers; i++) {
    players.get(i).setNextPlayer(players.get((i + 1) % numPlayers));
}
```

then setting up the circular order of player turns using the setNextPlayer() method. The loop iterates through each player and sets the reference of the current player to the next player in the list. this will help to synchronize the turn in game

now can distribute the card and start all threads (players)

1. Player Class and Threads:
   The Player class extends the Thread class, making it a separate thread of execution. Each player has a name, a set of cards in their hand, a reference to the next player, and a `CountDownLatch` instance to synchronize the end of the game.

2. Synchronization using Locks:
   The code uses a shared lock object to establish a critical section where threads synchronize their actions to avoid conflicts. This ensures that only one player can execute the synchronized block at a time, preventing race conditions

```
public void run() {

        synchronized (lock)
        {

            while (!isGameOver) {
                try {
                while (!isMyTurn) {
                    lock.wait();
                }
```

The player threads need to take turns to play the game. The isMyTurn boolean flag indicates whether it's a player's turn to play. if is not the player turn's, it enters a while (!isMyTurn) loop and waits for its turn using the lock.wait() statement. this mean all theads that don't have turn yet will stay in wait(),This effectively suspends the thread's execution until it's notified by another player (by invoking lock.notifyAll())

```
    }
// Thread.sleep(500);
    for (int i = 0; i < hand.size(); i++) {
    Card currentCard = hand.get(i);
    for (int j = i + 1; j < hand.size(); j++) {
        Card nextCard = hand.get(j);
        if (currentCard.isMatchingPair(nextCard)) {
            System.out.println(name + " discarded " + currentCard + " and " + ne
            hand.remove(j);
            hand.remove(i);
            dropedCardCount += 2;
            break;
        }
    }
}
isMyTurn = false;
nextPlayer.isMyTurn = true; // Move the turn to the next player
```

After a player finishes their turn, the value of "isMyTurn" for that player will change to false, and for the next player, it will change to true. Subsequently, a notification will be sent to all threads (players). Only the player with "isMyTurn" set to true will be able to exit the loop, while the others will remain in the "lock.wait()" state.

However, prior to using "notifyAll," the player will select a card at random and pass it to the next player. Following this card exchange, the player will then issue a notification. In case a player no longer possesses any cards, they will simply notify all players.

```
Card cardToReceive = giveRandomCard();
if (cardToReceive != null) {
    System.out.println(name+" will give card : "+cardToReceive);
    nextPlayer.receiveCard(cardToReceive);


}else{
    lock.notifyAll();
}
```

```
public void receiveCard(Card card) throws InterruptedException {
    synchronized (lock) {
        hand.add(card);
        System.out.println(name +"will take card "+card.toString());

        lock.notifyAll();
    }
}
```

in this way we will ensuring efficient use of the CPU

3. Turn-based Gameplay:

The main gameplay logic revolves around player turns. Each player thread runs a loop that continues until the game is over.

When a player wins and no longer has any cards, they remain in the game as a link between players. take card from the previous player and give it to the next player until the game is over.

```
public void run() {

    synchronized (lock)
    {
        while (!isGameOver) {
            try {
            while (!isMyTurn) {
                lock.wait();
            }
                for (int i = 0; i < hand.size(); i++) {
                Card currentCard = hand.get(i);
                for (int j = i + 1; j < hand.size(); j++) {
                    Card nextCard = hand.get(j);
                    if (currentCard.isMatchingPair(nextCard)) {
                        System.out.println(name + " discarded " + currentCard + " and " + nextCard);
                        hand.remove(j);
                        hand.remove(i);
                        dropedCardCount += 2;
                        break;
                    }
                }
            }
            isMyTurn = false;
            nextPlayer.isMyTurn = true; // Move the turn to the next player
                if (hand.size()==1 && hasJoker()  && Player.dropedCardCount ==52) {
                    System.out.println("-----------------------------------------------------");
                    System.out.println("the total card discarded is: "+Player.getDropedCardCount());
                    System.out.println(name+" still has the Joker");
                    isGameOver = true;
                    break;
                }

            Card cardToReceive = giveRandomCard();
            if (cardToReceive != null) {
                System.out.println(name+" will give card : "+cardToReceive);
                nextPlayer.receiveCard(cardToReceive);
            }else{
                lock.notifyAll();
            }
```

4.Game Over Conditions:

The game is end:

when there one player has only one card remaining in their hand and that card is a Joker, and all 52 cards have been discarded the game will ends.

```
isMyTurn = false;
nextPlayer.isMyTurn = true; // Move the turn to the next player
    if (hand.size()==1 && hasJoker()  && Player.dropedCardCount ==52) {
        System.out.println("-------------------------------------------------------");
        System.out.println("the total card discarded is: "+Player.getDropedCardCount());
        System.out.println(name+" still has the Joker");
        isGameOver = true;
        break;
    }
```

that's mean the loser will be the first thread leave the game

here we need to way to delay the tread to avoid terminate the game before all threads join There is two ways to do it :

1) Thread.sleep()

```
Thread.sleep( millis: 2000);
Player.getTheResult();
for (Player player : players) {
    try {
        player.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
System.out.println("--- all done ---\n");
```

2)barrier : (CountDownLatch)

```
CountDownLatch latch = new CountDownLatch(numPlayers);
```

```
latch.countDown();
```

```
//Thread.sleep(2000);
try {
    latch.await(); // This will block until the count reaches zero
} catch (InterruptedException e) {
    e.printStackTrace();
}
Player.getTheResult();
for (Player player : players) {
    try {
        player.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
System.out.println("--- all done ---\n");
```

The CountDownLatch (latch) is used to wait for all player threads to finish executing before determining the game results. The countDown() method is called on the latch at the end of each player's run() method. The main thread that started the game waits for this latch to reach zero, indicating that all player threads have completed.
I used barrier because they give faster performance

5.the Results:
The getTheResult method prints the winners and loser.
6. join:
join.

clean code principles (Uncle Bob)

these principles can be applied to the code:

1. Descriptive Naming:
- The class names Card, Deck, and Player are descriptive and reflect their roles in the game.
- Variable names like suit, value, numPlayers, isGameOver, etc., are reasonably clear.

2. Functions and Methods:
- The Card class has a single-responsibility constructor that initializes the card's suit and value. The isMatchingPair method checks for matching pairs of cards.
- The Deck class has well-defined methods like shuffle, size, and dealCard, which follow single-responsibility principle.
- The Player class handles individual player behavior and game logic.

3. Comments:
- The code contains some comments that explain certain sections.

4. Avoiding Duplication:

- avoided duplication well, as similar actions are put into methods

5. Single Responsibility Principle:
- The classes seem to have clear responsibilities:
  - Card: Represents a playing card.
  - Deck: Manages the deck of cards and provides methods for shuffling and dealing.
  - Player: Represents a player, handles card manipulation, and participates in the game loop.

6. Avoiding Global Variables:
- The `instance` variable in the `Deck` class is a singleton instance, but it's not used outside the class, so it doesn't pose a global variable concern.

7. Formatting and Indentation:
- The code follow a consistent indentation style, which is important for readability.

8. Error Handling:
- The code could benefit from improved error handling, particularly around handling potential exceptions that may arise during card dealing, player turns, or synchronization.

The end