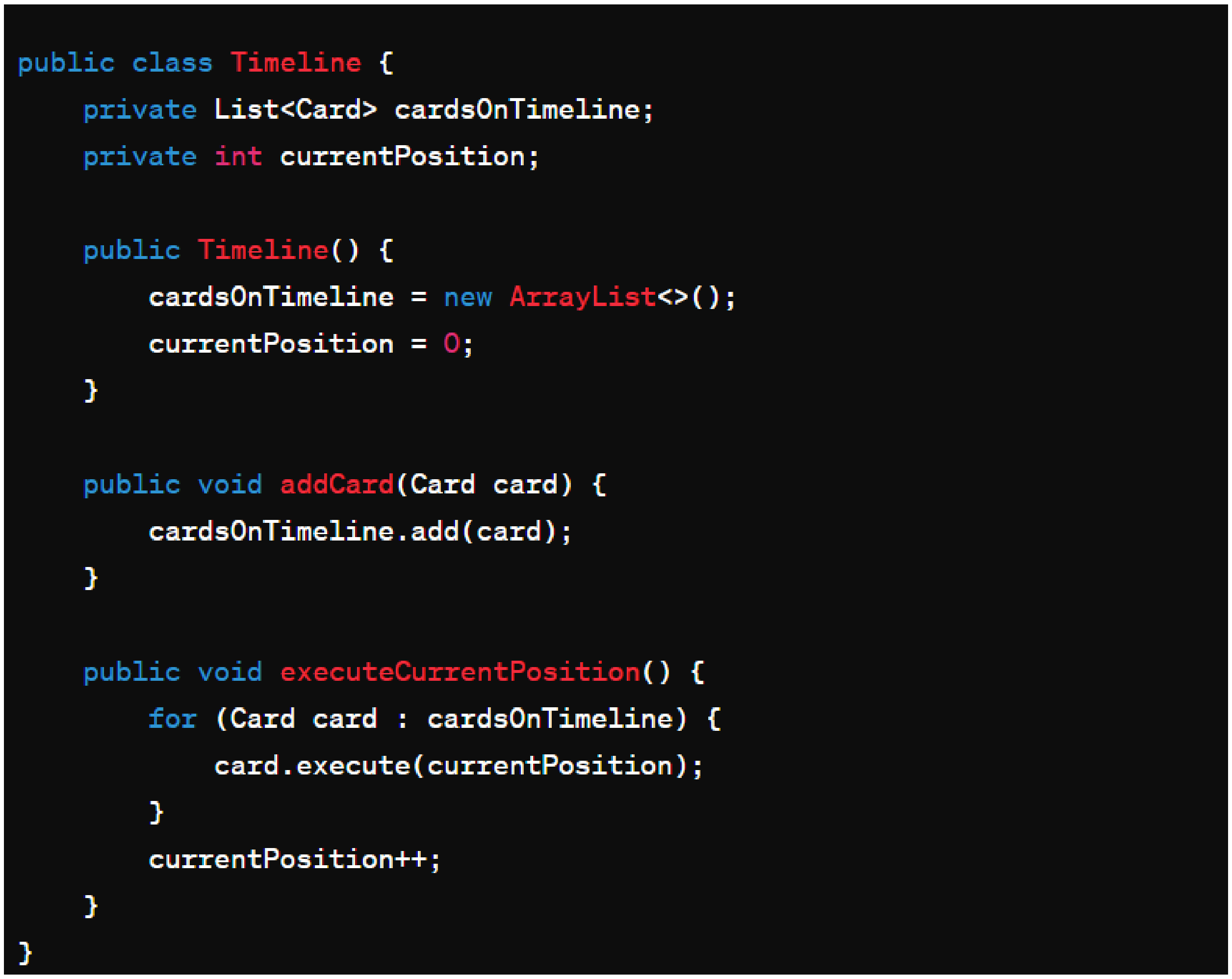
OOP Project Phase1

Aida Karimzadeh 400101797

1.

For implementing timeline in our project, we can create a simple class for it.

Here is a sample of its code:



```java
public class Timeline {
    private List<Card> cardsOnTimeline;
    private int currentPosition;

    public Timeline() {
        cardsOnTimeline = new ArrayList<>();
        currentPosition = 0;
    }

    public void addCard(Card card) {
        cardsOnTimeline.add(card);
    }

    public void executeCurrentPosition() {
        for (Card card : cardsOnTimeline) {
            card.execute(currentPosition);
        }
        currentPosition++;
    }
}
```

In this class, we maintain a list of cards which are on the timeline. Then we define a method; addCard() with which we can add a card to the timeline.

Then we can write another method; executeCurrentPosition() that executes all cards on the timeline.

In this class, we have Card objects that are needed to have an 'execute(int position)' method that defines what happens when the card is executed at a given position on the timeline.

2.

Damage/Healing Cards:

1. **Shadow Bolt:**
    - Type: Damage
    - Description: Hurls a bolt of dark energy at the target, causing shadow damage.
    - Ability: Inflicts 40 shadow damage to the target enemy.
2. **Holy Light:**
    - Type: Healing
    - Description: Radiates healing energy, restoring health to allies with divine light.
    - Ability: Heals the target ally for 50 health points.
3. **Venomous Bite:**
    - Type: Damage
    - Description: Injects a potent venom into the target, causing poison damage over time.
    - Ability: Deals 30 initial damage and 15 poison damage per turn for 3 turns.
4. **Rejuvenating Spring:**
    - Type: Healing
    - Description: Creates a spring of rejuvenating water, restoring vitality to allies.
    - Ability: Heals all allies within range for 20 health points per turn for 3 turns.
5. **Ice Shard:**
    - Type: Damage
    - Description: Launches sharp shards of ice at the enemy, causing frost damage.

- Ability: Inflicts 35 frost damage and slows the target's movement speed by 50% for 2 turns.

6. **Regenerative Aura:**
   - Type: Healing
   - Description: Emits a soothing aura, accelerating natural healing processes for allies.
   - Ability: Restores 15 health per turn to all allies within range for 5 turns.
7. **Searing Blaze:**
   - Type: Damage
   - Description: Ignites the target with a blazing inferno, causing fire damage.
   - Ability: Deals 50 fire damage to the target enemy.
8. **Divine Shield:**
   - Type: Healing
   - Description: Conjures a protective shield of divine energy, absorbing damage.
   - Ability: Grants a shield that absorbs 60% of incoming damage for 2 turns.
9. **Toxic Cloud:**
   - Type: Damage
   - Description: Releases a noxious cloud of poison, damaging all enemies in the area.
   - Ability: Deals 20 poison damage to all enemies within range per turn for 3 turns.
10. **Life Drain:**
    - Type: Healing
    - Description: Drains the life force from the enemy, transferring it to the caster.
    - Ability: Absorbs 50% of the damage dealt to the target enemy and restores the caster's health by the same amount.
11. Fireball:
    - Type: Damage
    - Description: Launches a fiery projectile at the target, dealing moderate damage.
    - Ability: Inflicts 50 damage to the target enemy.
12. Healing Aura:
    - Type: Healing
    - Description: Creates a soothing aura around allies, restoring their health.
    - Ability: Heals all friendly units within range for 30 health.
13. Poisonous Dagger:
    - Type: Damage

- Description: Inflicts a toxic wound on the target, causing damage over time.
- Ability: Deals 20 initial damage and an additional 10 damage per turn for 3 turns.

14. Regeneration:
    - Type: Healing
    - Description: Accelerates natural healing processes, gradually restoring health.
    - Ability: Restores 10 health per turn for 5 turns to the caster.
15. Thunder Strike:
    - Type: Damage
    - Description: Calls forth a bolt of lightning to strike the enemy, dealing massive damage.
    - Ability: Inflicts 100 damage to the target enemy.

Spell Cards:

1. Teleportation:
    - Type: Spell
    - Description: Instantly transports the caster to a designated location on the battlefield.
    - Ability: Allows the caster to teleport to any visible location on the battlefield.
2. Barrier:
    - Type: Spell
    - Description: Creates a protective barrier around the caster, reducing incoming damage.
    - Ability: Grants a shield that absorbs 50% of incoming damage for 3 turns.
3. Haste:
    - Type: Spell
    - Description: Temporarily enhances the caster's speed and agility, allowing for quick actions.
    - Ability: Increases caster's movement speed and attack speed by 50% for 3 turns.
4. Silence:
    - Type: Spell
    - Description: Silences the target, preventing them from casting spells or using abilities.
    - Ability: Silences the target for 2 turns, rendering them unable to cast spells or use abilities.
5. Meteor Shower:

- Type: Spell
- Description: Summons a rain of fiery meteors to bombard the battlefield, damaging all enemies.
- Ability: Deals 50 damage to all enemies on the battlefield.

3.

To implement the overall gameplay, we need to create several classes.

First, we have a Card Class. This is the base class for all types of cards. Subclasses include: DamageCard, HealingCard, SpellCard.

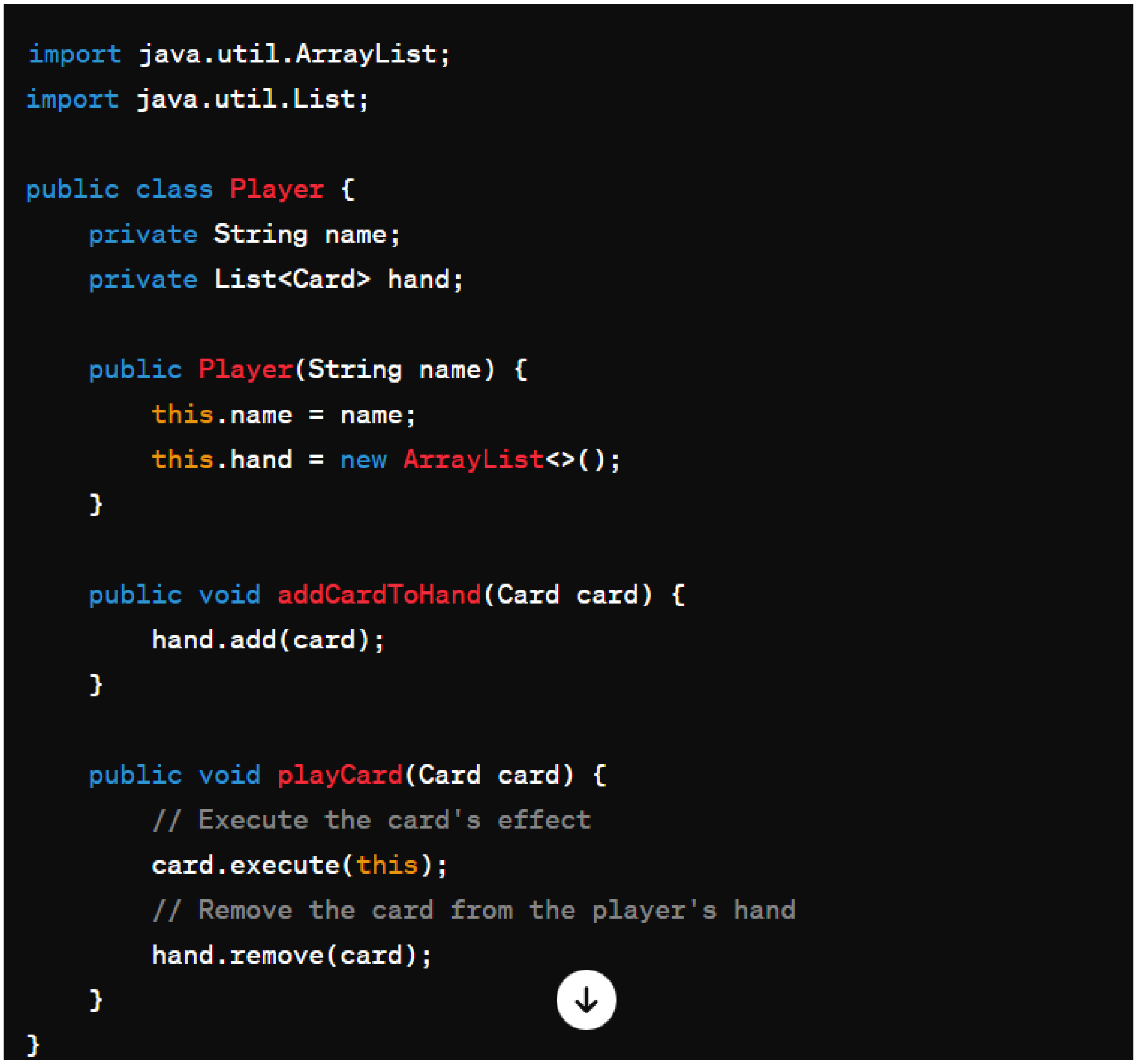
Methods: Executes the effect of the card on the given player.

```
public abstract class Card {
    protected String name;
    protected String description;

    public Card(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public abstract void execute(Player player);
}
```

Then we have a DamageCard, HealingCard, SpellCard Classes which inherit from Card Class.

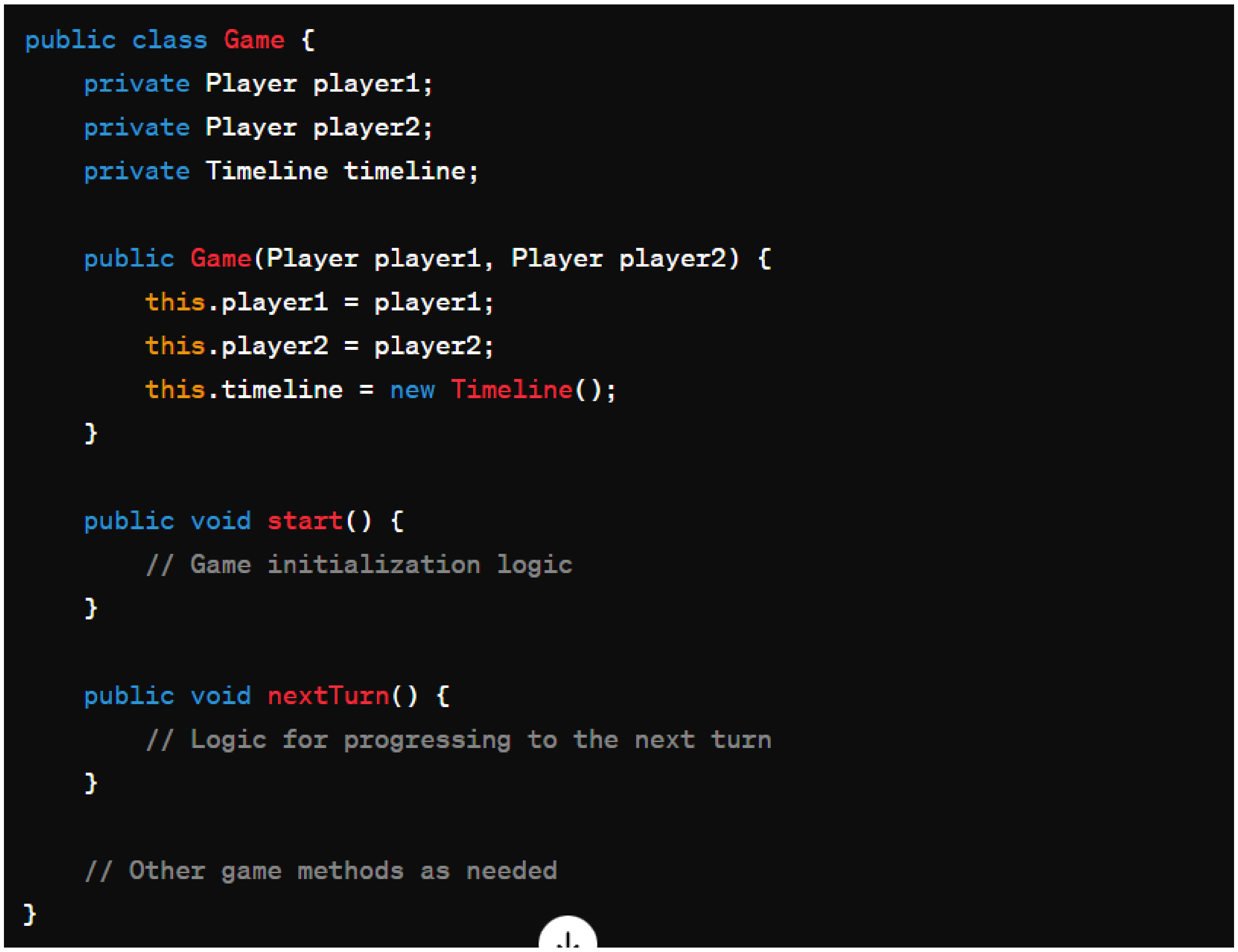
The other class is Player Class which can create objects of players, and it cintains a list of cards the player currently possesses.

Like this:



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

public class Player {
    private String name;
    private List<Card> hand;

    public Player(String name) {
        this.name = name;
        this.hand = new ArrayList<>();
    }

    public void addCardToHand(Card card) {
        hand.add(card);
    }

    public void playCard(Card card) {
        // Execute the card's effect
        card.execute(this);
        // Remove the card from the player's hand
        hand.remove(card);
    }
}
```

Then we implement a Game Class which controls the overall flow of the game, manages player turns, card drawing, and card executions as follow:



```java
public class Game {
    private Player player1;
    private Player player2;
    private Timeline timeline;

    public Game(Player player1, Player player2) {
        this.player1 = player1;
        this.player2 = player2;
        this.timeline = new Timeline();
    }

    public void start() {
        // Game initialization logic
    }

    public void nextTurn() {
        // Logic for progressing to the next turn
    }

    // Other game methods as needed
}
```

4.

For saving information of card upgrades and … in a database we need to set up a database schema to store the information.

For example for storing information of a player including her name or id and… we can structure this table:

Column Name	Data Type	Description
player_id	INT	Primary key, unique identifier for each player
name	VARCHAR	Name of the player
…	…	Other player attributes

Or for storing information of each card in the game we can struct this tabe:

Column Name	Data Type	Description
card_id	INT	Primary key, unique identifier for each card
name	VARCHAR	Name of the card
description	TEXT	Description of the card
type	VARCHAR	Type of the card (e.g., Damage, Healing, Spell)
...	...	Other card attributes

With this schema we can do the following operations to manage information:

- **Create:** When a player obtains a new card or upgrades an existing card, insert a new row into the PlayerCards table with the player's ID, the card's ID, and any upgrade information.
- **Read:** Retrieve player information, card information, or player-card relationships as needed to display in the game interface.
- **Update:** Modify player information, card information, or player-card relationships when players upgrade their cards or perform other actions that affect their data.
- **Delete:** Remove player information, card information, or player-card relationships when players delete their accounts or perform other actions that require data removal.

5.

Players can upgrade their cards using resources earned through the game like experienced points.

For example, we can define resources:

-players earn gold and gems through winning matches. Then we can define a mechanism for the players to spend their resources to upgrade their cards.

Then we can define rewards for the winners:

-the winner earn resources that they can use to purchase card upgrades and unlock new cards.

Then we define penalty for losers:

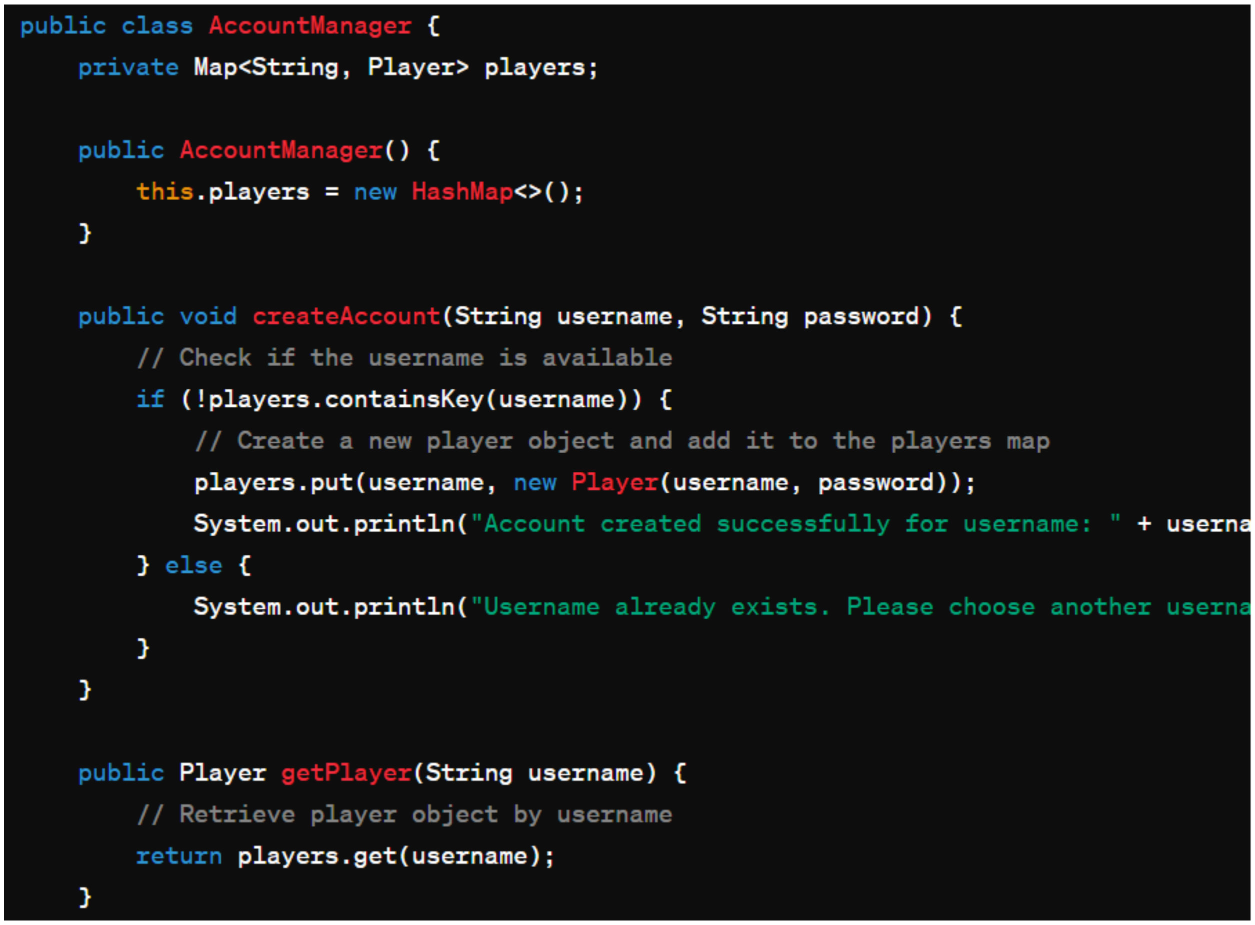
-for example, the loser earns less cards than the winner in the next rounds.

6.

To handle player identification and account creation we represent a class for individual players that define some properties as username and password:

```java
public class Player {
    private String username;
    private String password;
    // Other player attributes (e.g., level, experience, currency)

    public Player(String username, String password) {
        this.username = username;
        this.password = password;
        // Initialize other attributes
    }

    // Getter and setter methods for player attributes

    public boolean authenticate(String inputPassword) {
        // Check if the provided password matches the player's password
        return password.equals(inputPassword);
    }
}
```

Then we define an AccountManager Class to store player data and create new accounts. This class manages player's account by storing them in a map (username as key and Player object as value). It provides methods to create new accounts and authenticate players.



```
public class AccountManager {
    private Map<String, Player> players;

    public AccountManager() {
        this.players = new HashMap<>();
    }

    public void createAccount(String username, String password) {
        // Check if the username is available
        if (!players.containsKey(username)) {
            // Create a new player object and add it to the players map
            players.put(username, new Player(username, password));
            System.out.println("Account created successfully for username: " + userna
        } else {
            System.out.println("Username already exists. Please choose another userna
        }
    }

    public Player getPlayer(String username) {
        // Retrieve player object by username
        return players.get(username);
    }
```

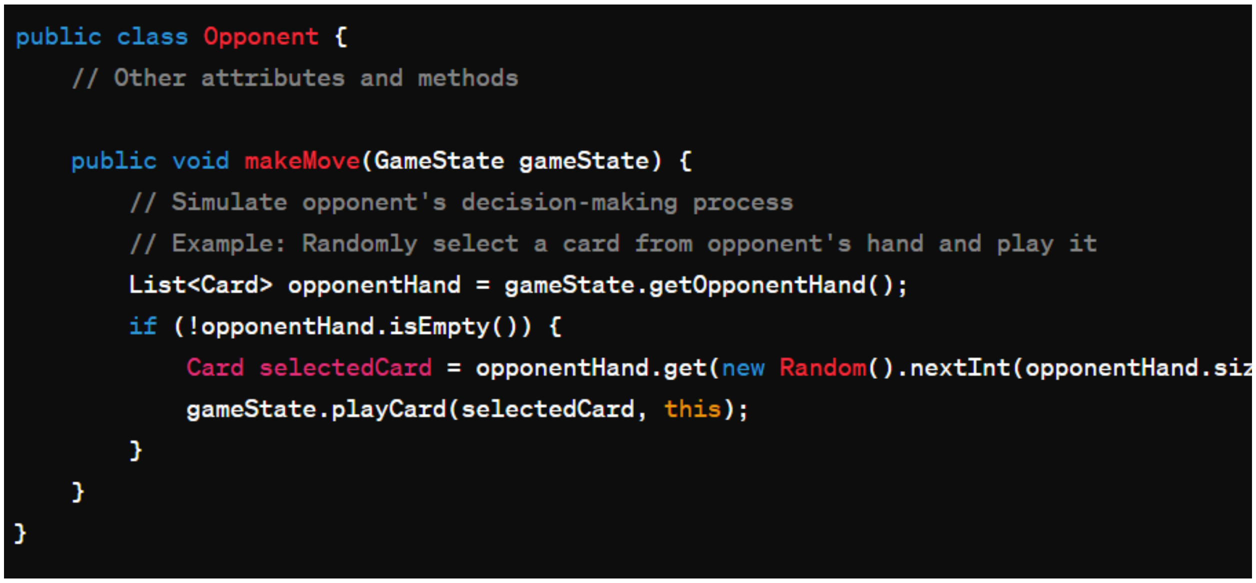
7.

To design an AI opponent, we need to design algorithms and methods.

First, we define a class called Opponent. Then we have to design an algorithm for decision-making. We set some rules that the opponent follows to make decision during gameplay. We can handle this, by some machine learning technics.

Second, we implement methods to simulate the opponent's moves during game play.

We define a method in Opponent class called makeMove. For example the opponent randomly selects a card from its hand and play it.


```java
public class Opponent {
    // Other attributes and methods

    public void makeMove(GameState gameState) {
        // Simulate opponent's decision-making process
        // Example: Randomly select a card from opponent's hand and play it
        List<Card> opponentHand = gameState.getOpponentHand();
        if (!opponentHand.isEmpty()) {
            Card selectedCard = opponentHand.get(new Random().nextInt(opponentHand.siz
            gameState.playCard(selectedCard, this);
        }
    }
}
```