**NAME : RAGHUL N**

**COURSE : DADS – RP29 Batch**

**Milestone - 1**

**Solitaire Card Game**

**Project Description**

**1.Aim of the Project**

The primary objective of this project is to design and implement a functional Solitaire card game leveraging Object-Oriented Programming (OOP) principles. The project aims to demonstrate how OOP concepts can be applied to effectively model and manage the game's components, structure, and behavior. Specifically, the project will focus on:

* **Encapsulation**: The project uses classes (Card, Deck, and Solitaire) to encapsulate game functionality and separate concerns.
* **Modularity**: By organizing the game into objects (cards, deck, columns, and foundation piles), the code is modular and easier to maintain.
* **Abstraction**: Complex operations like shuffling, dealing, and moving cards are abstracted into simple methods, hiding underlying details.

**2. Business Problem or Problem Statement:**

**Problem Statement:**

The aim of this project is to address the challenge of implementing a classic Solitaire card game in a structured, modular, and reusable way using Python’s Object-Oriented Programming (OOP) concepts. The specific business problem or educational objective is to:

**Simulate a Popular Game Using OOP**: Create a digital version of Solitaire that mimics the real-world gameplay experience while adhering to standard game rules and mechanics. The project should leverage OOP principles to make the codebase modular, maintainable, and easily extensible for future features or variations of the game.

**Provide a Learning Tool for OOP Concepts**: Many developers struggle with understanding how to apply abstract OOP concepts such as encapsulation, inheritance, and polymorphism to real-world projects. This project aims to serve as a practical learning tool that demonstrates how OOP principles can be effectively used to solve complex problems, such as game development.

**Improve User Engagement and Accessibility**: The game provides a platform for users to engage in a solitaire game digitally, making it accessible on any system with Python installed. This helps users enjoy a card game experience without the need for physical cards, creating entertainment and mental exercise.

**Key Challenges:**

* Implementing core Solitaire game rules.
* Using OOP to keep the code modular and reusable.
* Providing an interactive user experience with minimal user input.

**3. Project Description:**

This project creates a simple version of the **Solitaire** card game using **Python** and **Object-Oriented Programming (OOP)** principles. The game features a text-based interface where players can move cards between columns, place cards into foundation piles, and draw cards from a shuffled deck.

#### ****Key Points:****

* **Card and Deck Classes**: Models individual cards and a shuffled 52-card deck.
* **Columns and Foundations**: Seven columns for gameplay and four foundation piles for sorting cards.
* **Interactive Game Loop**: Players can move cards, draw new cards, and interact with the game using a console-based interface.
* **OOP Design**: Organized code structure for easy extension and maintenance.

**4.Functionalities**

**Card Management**:

* **Card Creation**: Create individual cards with suit and value attributes.
* **Card Representation**: Provides a string representation of a card (e.g., "Ace of Hearts").

**Deck Operations**:

* **Deck Initialization**: Creates a standard 52-card deck with all suits and values.
* **Shuffling**: Randomizes the order of cards in the deck.
* **Dealing Cards**: Allows cards to be dealt from the top of the deck.

**Game Setup**:

* **Initial Deal**: Distributes cards to seven columns for the initial game state.
* **Column and Foundation Setup**: Initializes seven columns and four foundation piles.

**Game State Display**:

* **Display Columns**: Shows the current state of all columns, including the cards they contain.
* **Display Foundation**: Shows the current state of all foundation piles.

**Card Movement**:

* **Move Between Columns**: Allows moving a card from one column to another.
* **Move to Foundation**: Moves a card from a column to one of the foundation piles.

****User Interaction**:**

* **Game Menu**: Provides options to move cards, draw a new card, or quit the game.
* **Input Handling**: Accepts user choices for moving cards and displaying game states.

**Game Loop**:

* **Continuous Play**: Keeps the game running until the player decides to quit.
* **Interactive Gameplay**: Updates the game state based on user actions and displays the results.

**5.Input Versatility with Error Handling and Exception Handling:**

In the Solitaire project, **input versatility** is key to creating a smooth and user-friendly experience, which is supported by robust **error handling** and **exception handling** mechanisms. The game prompts users to make various inputs, such as selecting columns to move cards between, choosing foundation piles, or drawing new cards. These inputs are validated to ensure that they are within expected ranges. For example, when moving a card, the program checks whether the user has selected a valid column number (between 1 and 7) or foundation pile (between 1 and 4).

To ensure that invalid inputs don’t disrupt the game, the project uses Python’s try-except blocks. These catch exceptions that might occur during gameplay, such as entering non-integer values, selecting an empty column, or attempting to move a card from a column that does not exist. For instance, if a user enters a column number outside of the valid range or an incorrect type of input (e.g., a letter instead of a number), the system catches the error and displays an appropriate message to the user, guiding them to enter valid data.

**6.Code Implementation:**

The code implementation for the Solitaire project is structured to leverage Object-Oriented Programming (OOP) principles to create a simple, yet functional, version of the classic Solitaire card game. Here’s a brief overview of the implementation:

**1.Card Class:**

class Card:

def \_\_init\_\_(self, suit, value):

self.suit = suit

self.value = value

def \_\_repr\_\_(self):

return f"{self.value} of {self.suit}"

* **Purpose**: Models an individual playing card with attributes for its suit (Hearts, Diamonds, Clubs, Spades) and value (Ace through King).
* **Constructor (**\_\_init\_\_**)**: Initializes a card with specified suit and value.
* **String Representation (**\_\_repr\_\_**)**: Provides a readable format for displaying the card.

**2.Deck Class:**

class Deck:

def \_\_init\_\_(self):

self.cards = []

self.suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']

self.values = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']

for suit in self.suits:

for value in self.values:

self.cards.append(Card(suit, value))

def shuffle(self):

random.shuffle(self.cards)

def deal\_card(self):

return self.cards.pop()

* **Purpose**: Manages a standard deck of 52 cards.
* **Constructor (**\_\_init\_\_**)**: Creates and populates the deck with all possible cards.
* **Shuffle Method**: Randomizes the order of cards.
* **Deal Card Method**: Removes and returns the top card from the deck.

**3.Solitaire Class:**

class Solitaire:

def \_\_init\_\_(self):

self.deck = Deck()

self.deck.shuffle()

self.columns = [[] for \_ in range(7)]

self.foundation = [[] for \_ in range(4)]

for i in range(7):

for \_ in range(i + 1):

self.columns[i].append(self.deck.deal\_card())

def display\_columns(self):

for i, column in enumerate(self.columns):

print(f"Column {i + 1}: {column}")

def display\_foundation(self):

for i, pile in enumerate(self.foundation):

print(f"Foundation {i + 1}: {pile}")

def move\_card(self, column1, column2):

card = self.columns[column1 - 1].pop()

self.columns[column2 - 1].append(card)

def move\_to\_foundation(self, column, foundation):

card = self.columns[column - 1].pop()

self.foundation[foundation - 1].append(card)

* **Purpose**: Encapsulates the game logic and state for Solitaire.
* **Constructor (**\_\_init\_\_**)**: Initializes the game with a shuffled deck, seven columns, and four foundation piles. Deals cards into the columns.
* **Display Methods**: display\_columns and display\_foundation show the current state of the columns and foundation piles.
* **Move Card Methods**: move\_card and move\_to\_foundation handle moving cards between columns and from columns to foundation piles.

**4.Game Loop:**

game = Solitaire()

game.display\_columns()

game.display\_foundation()

while True:

print("\n1. Move card between columns")

print("2. Move card to foundation")

print("3. Draw new card")

print("4. Quit")

choice = input("Enter your choice: ")

if choice == "1":

column1 = int(input("Enter source column: "))

column2 = int(input("Enter destination column: "))

game.move\_card(column1, column2)

elif choice == "2":

column = int(input("Enter source column: "))

foundation = int(input("Enter foundation pile: "))

game.move\_to\_foundation(column, foundation)

elif choice == "3":

card = game.deck.deal\_card()

print(f"You drew: {card}")

elif choice == "4":

break

else:

print("Invalid choice. Please try again.")

game.display\_columns()

game.display\_foundation()

* **Purpose**: Manages the game flow and user interactions.
* **Game Initialization**: Creates an instance of Solitaire and displays the initial game state.
* **Game Loop**: Continuously prompts the user for actions, processes input to move cards or draw new ones, and updates the display accordingly.
* **Error Handling**: Provides feedback for invalid choices and ensures that user actions are handled correctly.

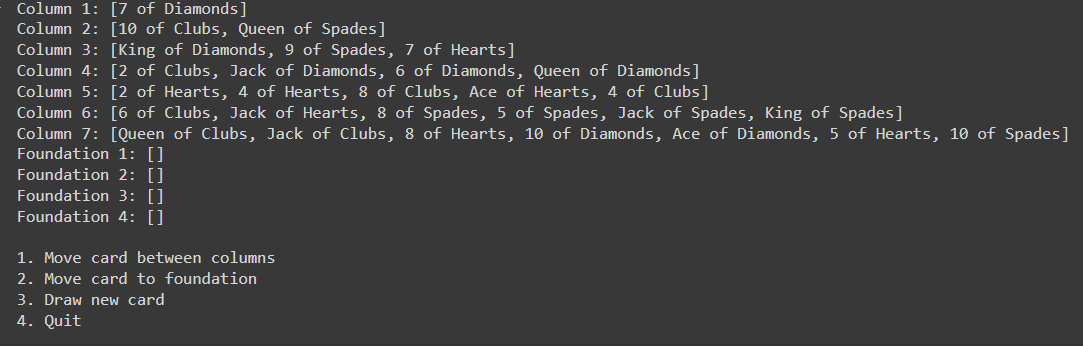
**7. Results and Outcomes:**

The Solitaire card game project successfully implements the core functionalities of a classic Solitaire game using Python and Object-Oriented Programming (OOP) principles. The game allows for smooth card management, movement between columns, and organizing cards into foundation piles, while providing clear feedback and handling invalid inputs efficiently.

**Key Results**:

* **Game Setup**: A deck of 52 cards is shuffled and dealt across 7 columns and 4 foundation piles.
* **Card Movement**: Players can move cards between columns and into foundation piles, following Solitaire rules.
* **User Interaction**: The game provides a text-based interface that handles user input, offering choices like card movement and drawing new cards.
* **Error Handling**: The system gracefully handles errors, such as invalid column selections or non-integer inputs, preventing game crashes and providing corrective feedback.

To visualize this outcome, imagine a console-based game display with neatly arranged columns of cards, foundation piles, and a user menu for interacting with the game. Here’s a graphical mock-up representing the game interface:



**8.Conclusion:**

The Solitaire project effectively applies Object-Oriented Programming principles to develop a functional card game in Python. The use of classes such as Card, Deck, and Solitaire ensures a clean, modular design that simplifies maintenance and future enhancements. The game features essential functionalities, including card movement and drawing, and provides a user-friendly interface with basic error handling. This project demonstrates both the practical application of OOP concepts and the creation of an engaging, interactive game experience.

**:**

THANK YOU