OOP Explained through Checkers

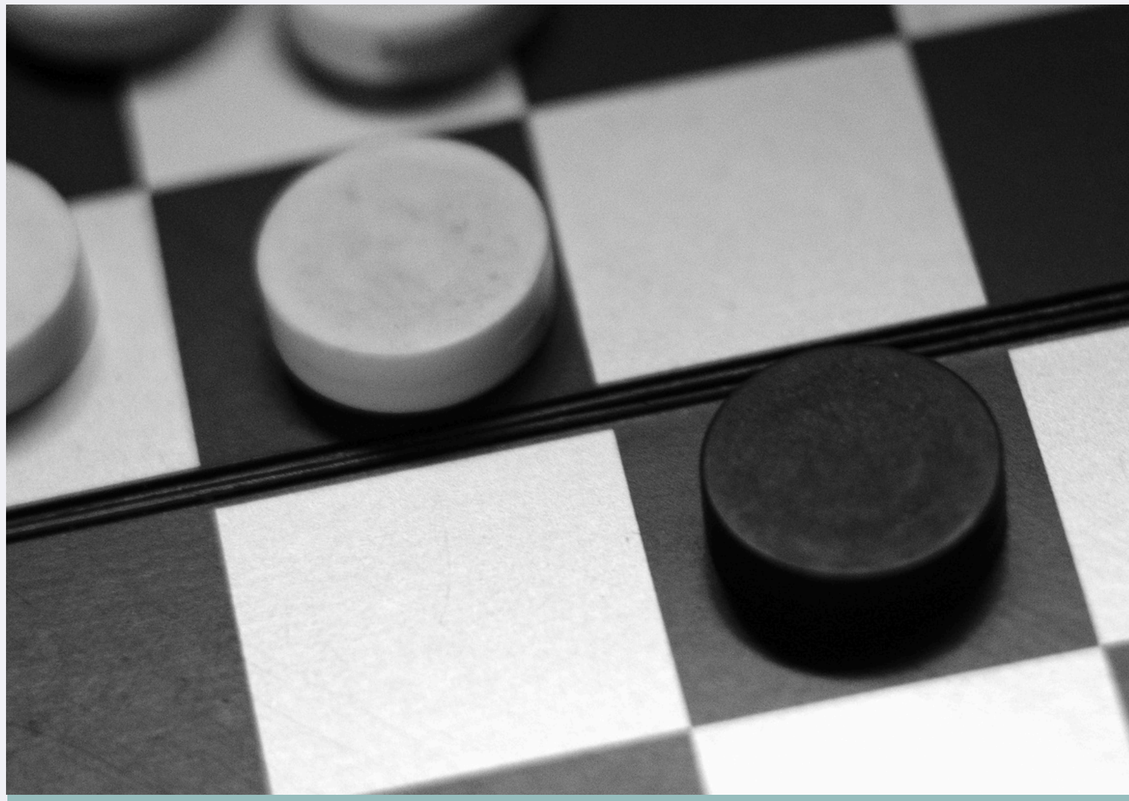
Presented by:
Asma Hammedi
Godwin Chigozie
Sachin Kumar
Yusra Adeyeri

Content


Introduction to
CHECKERS

Introduction to Checkers

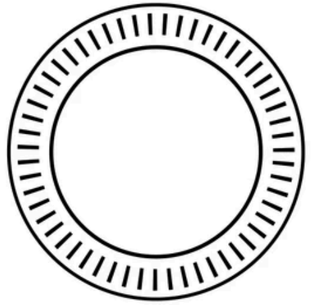
- Checkers is a two-player board game
- played on a checkered board with 64 squares.
- Players take turns moving their pieces diagonally forward.
- The objective is to capture all of the opponent's pieces by jumping over them or to block them so they cannot move.

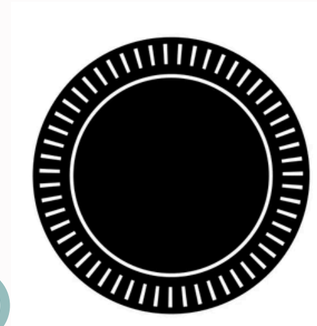


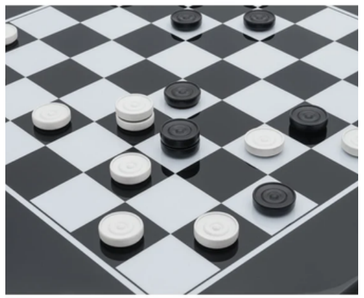
Pieces/Board

Board

- Standard 8x8 board
- Only dark squares are used for playing
- Each player starts with 12 pieces
- Pieces are placed on the first 3 rows of each side







Pieces

- All pieces start as "Men"
- Men can move diagonally forward
- Men can capture by jumping over opponent pieces diagonally

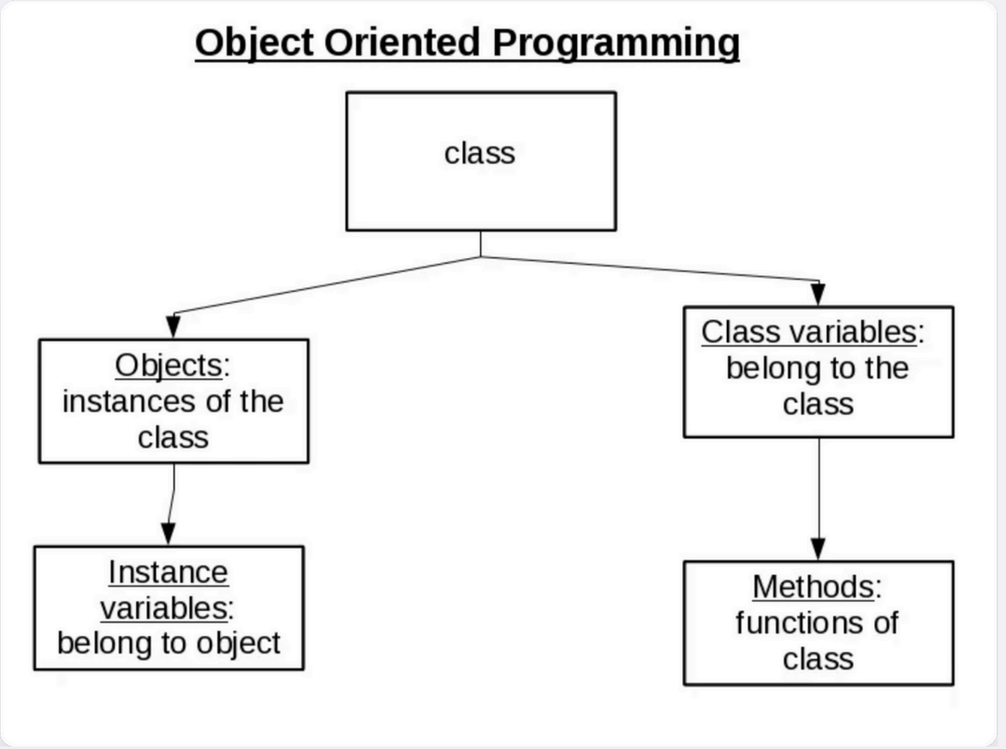
King Promotion

- A piece becomes a King when it reaches the last row
- Men can move diagonally forward
- Kings can move and capture diagonally both forward and backward


WHAT IS
OOP ?

Object Oriented Programming

Object-Oriented Programming is a programming paradigm where software is organized around objects, which combine data (attributes) and behavior (methods).



Characteristics of OOP

Encapsulation: Data security and access control

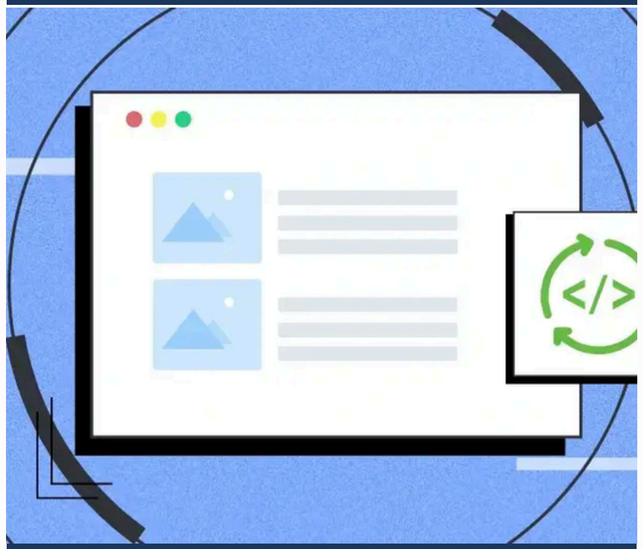
Abstraction : Showing only essential features and hiding the rest

Inheritance: The capability to create new classes derived from existing ones

Polymorphism : Same method name, different behavior depending on the object.

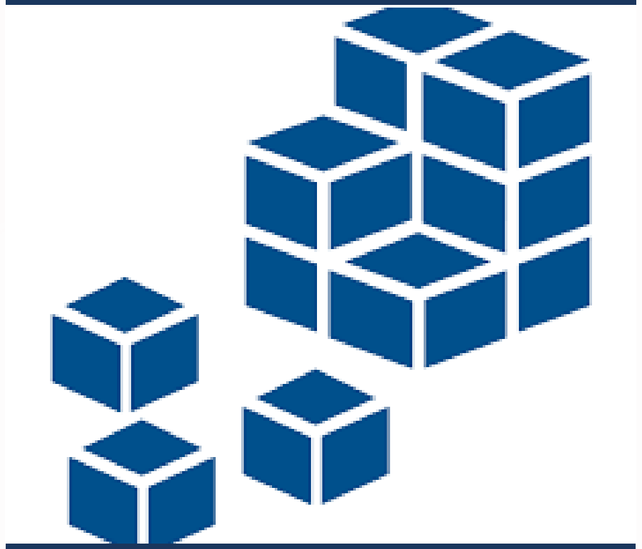
Benefits of using OOP

Code Reusability



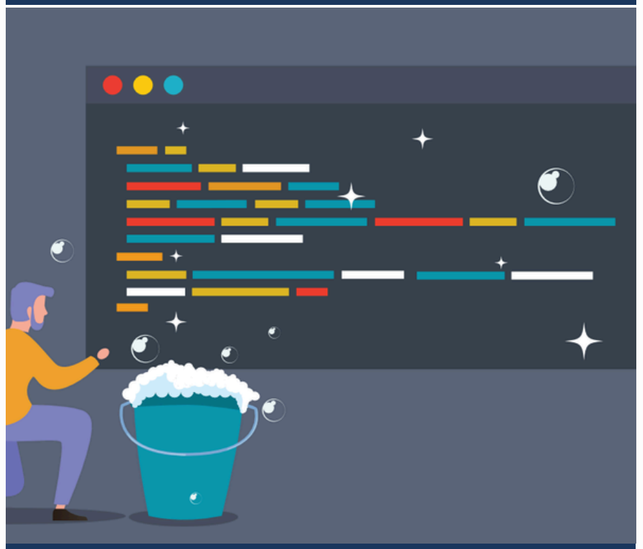
Reuse existing classes through inheritance to avoid rewriting code.

Modularity



Divide the program into independent objects, making it easier to manage and update.

Easier Maintenance



Encapsulation keeps code organized, so fixing bugs or adding features is simpler.

Benfits of using OOP

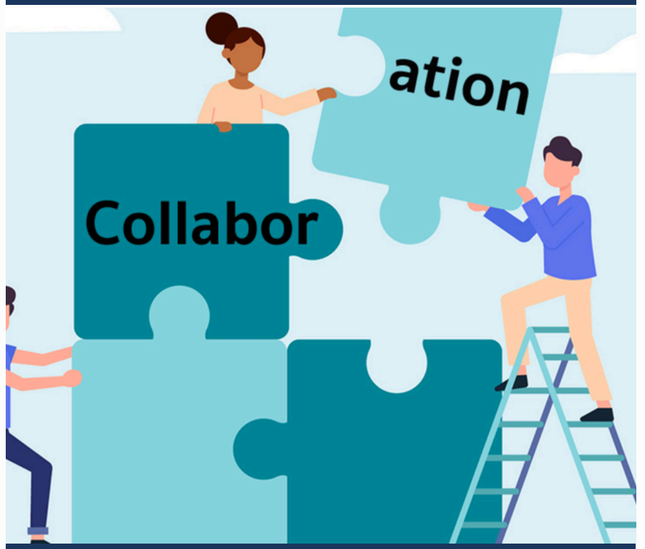
Scalability

Easy to add new features or components without changing existing code too much.

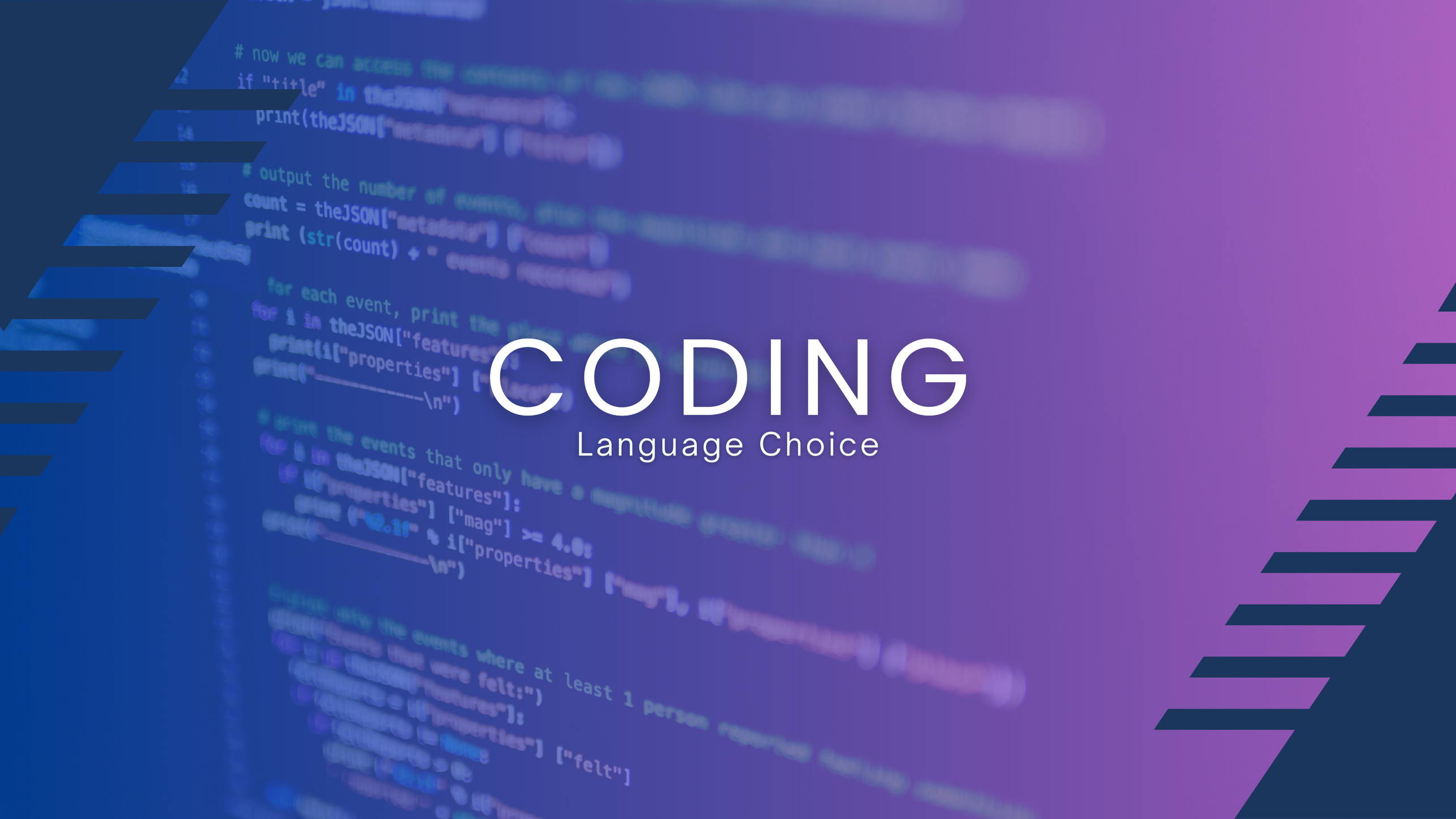
Data Protection

Keeps internal data safe by restricting direct access, improving security and reliability.

Collaboration Friendly



Modular structure helps teams work on different parts of the project without conflicts.


CODING
Language Choice

Python

- High-level, easy-to-read programming language
- Great for beginners and widely used in real-world applications
- Supports object-oriented programming (OOP)

Multi-Paradigm

Rich Standard Library

Large Community Support

High-level

PYGAME

Pygame is a Python library used to create 2D games and multimedia applications. It provides tools for handling graphics, sound, and user input with ease.

2D Graphics Support

Easily draw shapes, images, and game boards

Input Handling

Detects keyboard and mouse actions (like clicks or movement)

Simple & Lightweight

Easy to install and perfect for small game projects


APPLYING
OOP
To Checkers

Class Overview

- Base class for Man and King
- Attributes: color, is_king

Methods:

- __repr__()
- get_possible_moves() → abstract (to be overridden)

Class Overview

Man

- Represents a normal piece

Overrides:

- get_possible_moves() – moves diagonally forward
- __repr__(

Class Overview

King

- Represents a promoted piece

Overrides:

- get_possible_moves() – moves diagonally in all directions
- __repr__()

Class Overview

Player

- Represents a player and their pieces
- Attributes: color, pieces

Methods:

- add_piece()
- remove_piece()
- has_pieces()
- get_all_possible_moves()

Class Overview

Board

- Manages the 8x8 grid and piece logic
- Attributes: board (2D array)

Methods:

- initialize_board()
- get_piece(), set_piece(), remove_piece()
- move_piece()
- is_valid_move()
- get_all_pieces()
- display(), copy()

Class Overview

Match

- Controls the full game logic
- Attributes: board, player1, player2, current_player, game_over, winner

Methods:

- start_game()
- make_move()
- switch_player(), check_game_over()
- get_winner(), get_possible_moves_for_piece()
- is_game_over()

Encapsulation

```python
class Player:
    def __init__(self, color):
        self.color = color    #data (attribute)
        self.pieces = []      #data (attribute)

    def __repr__(self):       # Method that manipulates internal data
        return f'Player({self.color})'

    def add_piece(self, piece):
        self.pieces.append(piece)

    def remove_piece(self, piece):
        if piece in self.pieces:
            self.pieces.remove(piece)

    def get_pieces_count(self):
        return len(self.pieces)

    def has_pieces(self): # Method that gives access without exposing raw list
        return len(self.pieces) > 0
```

➡️ Combining data & the functions that manipulate that data into a single entity.

Data is protected and accessed only through controlled methods.

✅ Encapsulation in action!

Inheritance

```python
# Base class for all pieces (Man and King)
class Piece:
    def __init__(self, color):
        self.color = color
        self.is_king = False
```

```python
#Inheritance: Man class inherits from Piece
class Man(Piece):
    def __init__(self, color):
        super().__init__(color)  # Man class inherits from Piece
```

```python
#Inheritance: King class inherits from Piece
class King(Piece):
    def __init__(self, color):
        super().__init__(color)
        self.is_king = True   # King-specific property set to True
```

➡️ Man and King inherit from the base class Piece.

Shared attributes like color and is_king are reused from Piece.

Write once in Piece, reuse in both Man and King

✅ Inheritance in action!

Polymorphism

```python
def get_possible_moves_for_piece(self, row, col):    #Polymorphism: behaves differently for Man and King
    piece = self.board.get_piece(row, col)
    if piece and piece.color == self.current_player.color:
        return piece.get_possible_moves(self.board.board, row, col)
    return []
```

➡️ Python calls the correct version:
Man.get_possible_moves()
or King.get_possible_moves()
✅ Polymorphism in action!

Abstraction

```
#abstraction:This method hides all the internal game logic
 if match.make_move(from_row, from_col, row, col):
     selected_piece = None
     possible_moves = []
```

```
def make_move(self, from_row, from_col, to_row, to_col):
    piece = self.board.get_piece(from_row, from_col)
    if piece is None or piece.color != self.current_player.color:
        return False

    if self.board.move_piece(from_row, from_col, to_row, to_col):
        self.update_players_pieces()
        self.check_game_over()
        if not self.game_over:
            self.switch_player()
        return True
    return False
```

➡️ python calls :

match.make_move(from_row, from_col, to_row, to_col)

Complex logic hidden behind one simple method.

✅ Abstraction in action!

Comparison

	User Interaction	*Visual Feedback*	*Real-Time Experience*
CLI	✅ Text commands	❌ No visuals	❌ Not real-time
Rest API	✅ Programmatic only	❌ No visuals	❌ Not real-time
Pygame GUI	✅ Mouse & visuals	✅ Graphical	✅ fully interactive

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
=== CHECKERS ===
Welcome To Checkers!

=== COMMANDS ===
1 - Move a piece
2 - Show piece details
3 - Show current board
4 - Leave the game
Help - Display this help

Move format: A1-H8 (ex: A1, B2, etc.)
==============================
```

CLI

```
Player's turn: black
  A B C D E F G H
8 □ ● □ ● □ ● □ ● 8
7 ● □ ● □ ● □ ● □ 7
6 □ ● □ ● □ ● □ ● 6
5 ■ □ ■ □ ■ □ ■ □ 5
```

```
8 □ ● □ ● □ ● □ ● 8
7 ● □ ● □ ● □ ● □ 7
6 □ ● □ ● □ ● □ ● 6
5 ■ □ ■ □ ■ □ ■ □ 5
4 □ ■ □ ■ □ ■ □ ■ 4
3 ○ □ ○ □ ○ □ ○ □ 3
2 □ ○ □ ○ □ ○ □ ○ 2
1 ○ □ ○ □ ○ □ ○ □ 1
  A B C D E F G H

What would you like to do?
Your Choice:
```

REST API

localhost:5000

Impression automatique


```
{
  "endpoints": {
    "GET /board": "Get current board state",
    "GET /info": "Get piece info (params: ?player=Black&piece=2A)",
    "POST /move": "Make a move (JSON: {player, from, to})",
    "POST /reset": "Reset the game"
  },
  "example_move": {
    "from": "2A",
    "player": "Black",
    "to": "3B"
  },
  "message": "Checkers Game API"
```


POST http://localhost:5000/move Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

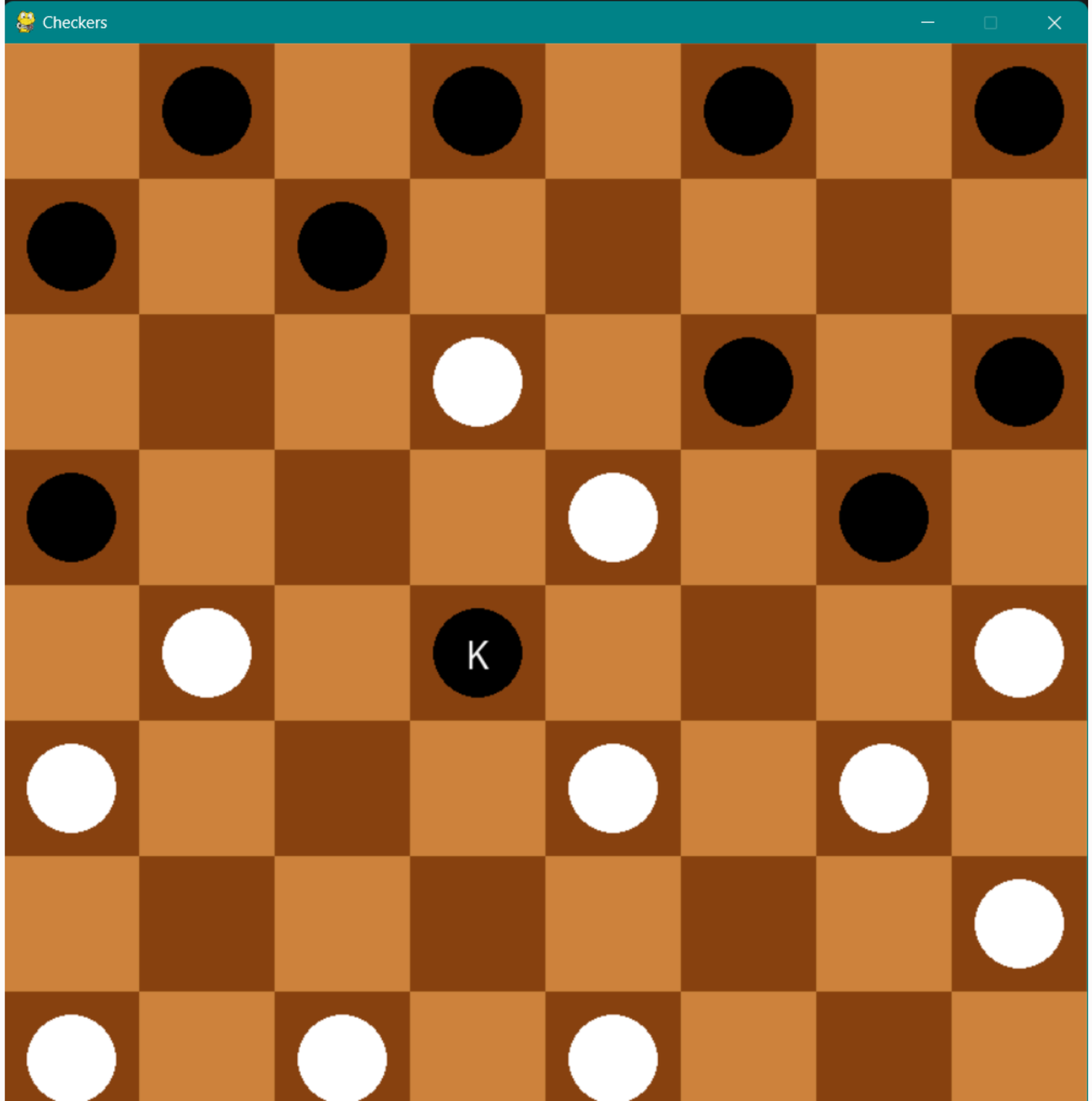

```
{
  "player": "black",
  "from": "B6",
  "to": "A5"
}
```


Status: 200 OK Size: 127 Bytes Time: 7 ms

Response Headers 6 Cookies Results Docs { }


```
{
  "current_player": "white",
  "game_over": false,
  "message": "Move made: B6 -> A5",
  "success": true,
  "winner": null
}
```

Pygame



CONCLUSION

Conclusion

This project demonstrates the flexibility of Object-Oriented Programming by implementing three different interfaces

Each interface serves a distinct purpose:

- CLI is lightweight and ideal for debugging or scripting.
- REST API enables integration with other systems or platforms.
- Pygame GUI offers the most engaging and intuitive experience for real-time users.

By abstracting the core game logic and separating it from the interface layer, the system becomes modular, testable, and easily extensible.

This multi-interface architecture showcases the power of encapsulation, abstraction, inheritance, and polymorphism — the key pillars of OOP.

THANK YOU