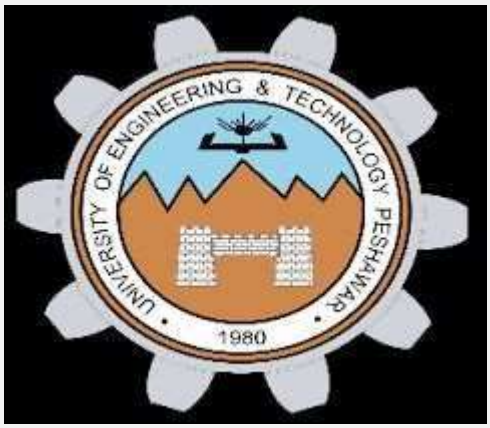
PROJECT PROGRESS REPORT

PROJECT NAME : TIC TAC TOE GAME



PREPARED BY :

M KAZIM AHMAD
ID:23PWCSE2223
AHMAD SHAHZAD

ID: 23PWCSE2265

SUBMITTED TO: MAM SUMMAYAH SALAHUDDIN

Department of computer system engineering university of engineering and technology Peshawar

TIC TAC TOE GAME

Tic-tac-toe is a two-player game played on a 3x3 grid. Players take turns marking X or O in an empty cell. The goal is to align three of their symbols horizontally, vertically, or diagonally. The game ends when one player wins or all cells are filled, resulting in a draw. It is a simple yet strategic game that promotes logical thinking.

OBJECTIVES:

- To develop a fun and interactive Tic Tac Toe game using Flutter for Android and iOS platforms.
- To enhance user engagement by implementing smooth gameplay and an intuitive UI/UX design.
- To strengthen programming skills in Dart and state management techniques within the Flutter

FIRST PROGRESS REPORT:

FILES STRUCTURE IN LIB FOLDER:

```
tic_tac_toe/
 ├— lib/
|   ├— main.dart
|   ├— screens/
|   |   ├— game_screen.dart
|   ├— widgets/
|   |   ├— board.dart
|   |   ├— cell.dart
|   ├— utils/
```

| | ├— game_logic.dart

| ├— styles/

| ├— colors.dart

| ├— text_styles.dart

CODES OF EACH FILE:

Game screen.dart:

IMPORT STATEMENTS:

```
class GameScreen extends StatefulWidget {
  @override
  _GameScreenState createState() => _GameScreenState();
}
```

1 **import 'package:flutter/material.dart';**

- Imports the Flutter Material package, which provides essential widgets for designing the user interface using Material Design principles.
- Includes widgets like Scaffold, AppBar, Text, and more.

2 **import '../widgets/board.dart'**;

Imports a custom file named board.dart located in the widgets directory one level up from the current file. This file likely contains a widget (Board) used to display the game board for Tic Tac Toe.

DEFINING THE GAME WIDGE:

```
class GameScreen extends StatefulWidget {  @override
_GameScreenState createState() => _GameScreenState();

}
```

3. **class GameScreen extends StatefulWidget**

Defines GameScreen as a StatefulWidget.

StatefulWidget is used when a widget's UI might change during its lifetime, which is common for interactive applications like a game.

4. **@override**

   A decorator indicating that the following method overrides a method from the parent class (StatefulWidget).

5. **_GameScreenState createState() => _GameScreenState();**

   Creates and returns the associated state object _GameScreenState, which holds

   the state of the GameScreen widget.

```
class _GameScreenState extends State<GameScreen> {
  @override
  Widget build(BuildContext context) {
return Scaffold(      appBar: AppBar(title:
Text('Tic Tac Toe')),      body: Center(
child: Board(),
      ),
    );
  }
}
```

OOPS CONCEPTS USED IN THIS CODE :

1. Classes:

- **Code**: class GameScreen extends StatefulWidget and class _GameScreenState extends State<GameScreen>
- **Explanation**:
    - Classes are the core building blocks of OOP, used to define objects.
    - GameScreen and _GameScreenState are classes representing components of the application.
    - GameScreen is a Flutter widget that defines the stateful nature of the UI component. _GameScreenState manages its internal state.

2. Inheritance

- **Code**: class GameScreen extends StatefulWidget and class _GameScreenState extends State<GameScreen>
- **Explanation**:
    - The extends keyword shows inheritance, where GameScreen inherits from StatefulWidget, and _GameScreenState inherits from State<GameScreen>.
    - This inheritance allows GameScreen to leverage the functionality of StatefulWidget, such as maintaining state.

3. Encapsulation

- **Code**: _GameScreenState •
  Explanation:
    - The _ prefix makes _GameScreenState private to its file, which encapsulates its implementation details. Other files cannot access this class directly, ensuring modularity and hiding implementation specifics.

4. Abstraction

- **Code**: Scaffold, AppBar, and Board •
  Explanation:

- These widgets abstract complex UI behaviors into simple components.
- For example, Scaffold provides a layout structure without requiring the user to manage the entire UI framework.

5. Polymorphism

- **Code**: @override Widget build(BuildContext context) •
  Explanation:
  - The build method is overridden in _GameScreenState to provide specific behavior for rendering the UI.
  - Polymorphism allows a base class method (build from State) to be redefined in a subclass to tailor its functionality.

BUILDING THE UI :

```
return Scaffold(   appBar:

AppBar(title: Text('Tic Tac Toe')),

body: Center(     child: Board(),

  ),

);
```

Scaffold :

A fundamental layout widget in Flutter that provides a structure for Material Design apps.

It includes an AppBar and a body.

AppBar(title: Text('Tic Tac Toe'))

Adds an application bar at the top of the screen with the title "Tic Tac Toe".

body: Center(...)

The body of the Scaffold contains the main content of the screen.

The Center widget centers its child horizontally and vertically within the available space.

child: Board()

Adds the Board widget (imported from board.dart) as the main content.
This widget likely represents the Tic Tac Toe game board where the game logic will be implemented.

STYLES :

COLOURE:

import 'package:flutter/material.dart';

Imports the Material Design package, which provides predefined classes and constants (like `Colors`) for designing user interfaces.

This is required because you are using the `Colors` class from this package to define color constants.

class AppColors { static const Color primary = Colors.blue; static const Color secondary = Colors.blueAccent; static const Color background = Colors.white;

}

1class AppColors

- Defines a class named AppColors.
- This class is used to store colour constants, making it easier to manage and reuse colours throughout your Flutter app.

2 static const Color primary = Colors.blue;

- static means this variable belongs to the class itself rather than an instance. You can access it using AppColors.primary without creating an object of the class.
- const makes this a compile-time constant, meaning its value cannot change.
- Colour is the data type for storing colour information.
- Colours. blue is a predefined colour constant from Flutter's Colors class, representing the colour blue.

3 static const Color secondary = Colors.blueAccent;

- Similar to the primary colour, this defines another constant for a secondary colour, Colors. Blue accent, which is a lighter variation of blue.

4. static const Color background = Colors.white;

- Defines a constant for the background colour, Colors. white, which represents the colour white.

3. OOPS CONCEPTS USED IN THIS CODE :

1. Class

- **Code**: class AppColors •
  Explanation:
  - A **class** is used to group related properties (primary, secondary, background) into a single entity.
  - The AppColors class acts as a container for color definitions, providing structure and organization to the code.

2. Encapsulation

- **Code**: static const Color primary, static const Color secondary, static const Color background
- **Explanation**:
  - **Encapsulation** is evident as the properties (primary, secondary, background) are contained within the AppColors class, grouping them together.
  - This encapsulation makes it easy to manage and use color constants across the application while keeping them organized in one place.

3. Abstraction

- **Code**: Colors.blue, Colors.blueAccent, Colors.white •
  Explanation:
  - The Colors class provided by Flutter abstracts the underlying implementation of color definitions.
  - You don't need to know how colors are internally managed; you simply use the predefined colors like Colors.blue.

4. Static Members

- **Code**: static const Color primary, static const Color secondary, static const Color background
- **Explanation**:
  - The static keyword indicates that these properties belong to the class itself, not individual instances.
  - This ensures that you can access these colors globally without creating an instance of AppColors.

5. Constants

- **Code**: const
- **Explanation**:
  - The const keyword ensures that these color values are compile-time constants, providing immutability.
  - This is an example of designing classes to manage unchangeable, reusable data.

STYLES :

DEFINING THE APPTEXTSTYLES CLASS:

```
class AppTextStyles {

static const TextStyle title =

    TextStyle(fontSize: 30.0, fontWeight: FontWeight.bold);

static const TextStyle player =
```

TextStyle(fontSize: 30.0, fontWeight: FontWeight.bold);

} **class**

AppTextStyles:

- A utility class designed to store reusable text styles for the app, similar to how AppColors centralizes color definitions.

static const TextStyle title:

- Defines a TextStyle for titles.
- static means it belongs to the class and can be accessed using AppTextStyles.title without creating an instance.
- const ensures the style is immutable and optimized at compile time.

TextStyle(fontSize: 30.0, fontWeight: FontWeight.bold):

- Creates a TextStyle with:

  fontSize: 30.0: Sets the text size to 30 logical pixels.

  fontWeight: FontWeight.bold: Makes the text bold.

static const TextStyle player:

- Defines another TextStyle for player-related text, using the same properties as title.

4. OOPS CONCEPTS USED IN THIS CODE :

1. Class

• **Code**: class AppTextStyles • **Explanation**: ○ A **class** is used to group related properties (title, player) into a single logical unit.

- AppTextStyles serves as a container for text style definitions, making it easy to manage and reuse text styles throughout the application.

2. Encapsulation

- **Code**: static const TextStyle title, static const TextStyle player •
  Explanation:
  - Encapsulation groups related properties (text styles) inside the AppTextStyles class, organizing the code and making it modular.
  - The styles are encapsulated within the class to ensure they can be accessed uniformly without exposing unnecessary details.

3. Abstraction

- **Code**: TextStyle(fontSize: 30.0, fontWeight: FontWeight.bold) •
  Explanation:
  - The TextStyle class abstracts the details of creating and managing font styles.
  - You simply specify parameters like fontSize and fontWeight without worrying about how these styles are rendered internally.

4. Static Members

- **Code**: static const TextStyle title, static const TextStyle player •
  Explanation:
  - The static keyword indicates that these text styles belong to the class itself, not individual instances.
  - This makes the styles globally accessible without needing to create an instance of AppTextStyles.

5. Constants

- **Code**: const
- **Explanation**:
  - The const keyword ensures that the text styles are immutable and evaluated at compile time.
  - This design is efficient and ensures that these styles remain unchangeable, which is desirable for consistent design.

CLASS DEFINITION: **class**

GameLogic {

Defines the GameLogic class, which contains the core logic for the Tic Tac Toe game.

BOARD DEFINITION:

List<List<String>> board = List.generate(3, (_) => List.generate(3, (_) => ''));

List<List<String>> board:

- A 2D list representing the Tic Tac Toe board, where each cell is initialized as an empty string ('').
- List.generate(3, ...) creates 3 rows, each containing 3 columns, resulting in a 3x3 board.

CURRENT PLAYER: **String**

currentPlayer = 'X';

String currentPlayer:

- Tracks the current player's symbol, initially set to 'X'.

MAKING A MOVE :

void makeMove(int row, int col) {

if (board[row][col] == '') {

```
board[row][col] = currentPlayer;
if (checkWinner(row, col)) {
print('$currentPlayer wins!');
resetBoard();
   } else if (board.expand((e) => e).every((cell) => cell.isNotEmpty)) {
print('It\'s a draw!');       resetBoard();     } else {       currentPlayer =
(currentPlayer == 'X') ? 'O' : 'X';
   }
 } }
```

void makeMove(int row, int col):

- Handles a player's move at the specified row and column.

if (board[row][col] == ''):

- Checks if the selected cell is empty before making a move. **board[row][col] = currentPlayer;**:
- Updates the cell with the current player's symbol ('X' or 'O').

if (checkWinner(row, col)):

- Calls the checkWinner method to determine if the current player has won.

print('$currentPlayer wins!');:

- Prints a message indicating the winner.

resetBoard();:

- Resets the board after a win or draw.

else if (board.expand((e) => e).every((cell) => cell.isNotEmpty)):

- Checks if all cells are filled, indicating a draw. **currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';**:

- Switches to the next player if the game continues.

OOPS CONCEPTS USED IN THIS CODE

1. Encapsulation

- The GameLogic class encapsulates all the functionality related to the game mechanics, such as maintaining the game board, switching players, checking for a winner, and resetting the board.
- By encapsulating these operations within the class, the internal details of the game logic are hidden from the outside, and only the relevant methods (makeMove, checkWinner, resetBoard) are exposed for external interaction.

2. Abstraction

- The code separates concerns by defining specific classes for different purposes:
  - AppColors encapsulates the color scheme used in the app.
  - AppTextStyles abstracts the text styling details.
  - GameLogic focuses solely on the game's functionality.
- This abstraction makes the codebase modular and easier to maintain or extend.

3. Modularity (Class Usage)

- The use of separate classes (AppColors, AppTextStyles, and GameLogic) ensures a clear separation of concerns, which is a key principle of OOP.

4. Reusability

- The constants defined in AppColors and AppTextStyles can be reused across the entire Flutter application, ensuring consistent styling.
- The GameLogic class can be reused or even extended for similar games without rewriting the logic.

5. Implied Polymorphism

- While no explicit polymorphism (e.g., inheritance or method overriding) is shown here, the code design supports extensibility. For example: ○ You could subclass GameLogic to create variations of the game logic or use the same methods with a different game rule set.

WIDGETS:

BOARD :

import 'package:flutter/material.dart';

import '../widgets/cell.dart'; import

'../utils/game_logic.dart';

1. **import 'package:flutter/material.dart';**:
   - Provides Flutter's core UI framework.
2. **import '../widgets/cell.dart';**:
   - Imports the Cell widget, which likely represents an individual cell of the Tic Tac Toe board.
3. **import '../utils/game_logic.dart';**:
   - Imports the GameLogic class, which contains the logic for the game.

Class definition:

4. **class Board extends StatefulWidget**:
   - Defines the Board widget as Stateful because the board needs to update dynamically when players make moves. 5. **@override _BoardState createState()**:
   - Creates an instance of _BoardState, the state class that holds the board's dynamic logic.

State Class :

class _BoardState extends State<Board> {

final GameLogic _gameLogic = GameLogic();

final GameLogic _gameLogic = GameLogic();:

Creates an instance of the GameLogic class to manage the game's logic, board state, and moves.

5. OOPS CONCEPTS USED IN THIS CODE :

Encapsulation :

- **Class Structure**:
  - The Board widget encapsulates the game's UI logic within its own class.
  - It uses the GameLogic class to manage the state of the game (board, moves, and players), abstracting the game mechanics from the UI implementation.
- **State Management**:
  - The _gameLogic object is private to _BoardState, ensuring that the game logic cannot be accessed or modified directly from outside this class. This prevents unintended changes and maintains a controlled flow of data.

2. Abstraction

- The code abstracts complex operations:
  - **Game Logic**: Delegates game mechanics (like validating moves, determining winners, and resetting the board) to the GameLogic class. ○ **Cell Representation**: Uses the Cell widget to abstract the individual tiles of the board, making the code more modular and reusable.
- The Board widget only focuses on the layout and interaction between cells, keeping its responsibilities clear.

3. Modularity

- The Board widget is a modular component of the app. It operates independently and integrates easily with other parts of the application (e.g., a parent widget or the game screen).

- The code reuses the Cell widget for each board cell, ensuring consistency in behavior and appearance while avoiding redundancy.

4. Inheritance

- The Board class inherits from StatefulWidget, and _BoardState inherits from State<Board>. This inheritance structure is part of Flutter's framework and allows:
    - Managing dynamic state changes (e.g., updating the board after a player move).
    - Separating the widget's immutable configuration (Board) from its mutable state (_BoardState).

5. Object Composition

- The Board widget composes objects from other classes:
  - It uses GameLogic for game mechanics. - It utilizes the Cell widget for rendering individual cells of the board.
  - It combines rows and columns to construct the complete game board.

6. Reusability

- The Cell widget and GameLogic class can be reused or extended for other projects or games.
- The modular design allows for easy integration of enhancements, like animations or custom themes, without modifying the core logic.

7. Dynamic Polymorphism

- The setState method of _BoardState dynamically updates the UI when changes occur in the game state (e.g., a move is made). While not explicit polymorphism through inheritance, this runtime behavior is a form of dynamic behavior provided by Flutter's widget lifecycle.

BUILDING THE UI :

```
@override
Widget build(BuildContext context) {   return
Column(    mainAxisAlignment:
MainAxisAlignment.center,    children:
List.generate(3, (row) {      return Row(
MainAxisAlignment: MainAxisAlignment.center,
children: List.generate(3, (col) {         return
Cell(
          value: _gameLogic.board[row][col],
          onTap: () {
SetState(() {
             _gameLogic.makeMove(row, col);
           });
          },
        );
       }),
     );
   }),
 );
}
```

Column:

- Arrange the rows of the board vertically in a column.
- **mainAxisAlignment: MainAxisAlignment.center**:
- Centres the board vertically within its parent container.
- **List.generate(3, (row) {...})**:
- Dynamically generates 3 rows for the board.
- **Row**:
- Represents a single row in the Tic Tac Toe board.
- **children: List.generate(3, (col) {...})**:
- Dynamically generates 3 cells (columns) for each row.

Cell:

- Represents an individual cell on the board.
- **value: _gameLogic.board[row][col]**:
- Passes the value of the cell ('X', 'O', or '') from the GameLogic board to the Cell widget.
- **onTap: () {...}**:
- Defines what happens when a cell is tapped.
- Calls setState to update the UI and invokes _gameLogic.mak Move(row, col) to handle the move logic.

CELLS:

CLASS DEFINITION :

class Cell extends StatelessWidget {

final String value; final

VoidCallback onTap;

Cell({required this.value, required this.onTap});

}

- **class Cell extends StatelessWidget**:
  - Defines the Cell widget, which is a single, immutable cell on the Tic Tac Toe board.
- **final String value**:
  - A property that holds the value of the cell ('X', 'O', or '').
- **final VoidCallback onTap**:
  - A callback function executed when the cell is tapped. ○ VoidCallback is a Flutter type for functions that take no parameters and return no value.
- **Cell({required this.value, required this.onTap});**:
  - Constructor that requires the value and onTap properties to be passed when creating a Cell.

OOPS CONCEPTS USED IN THIS CODE :

- **Encapsulation of Properties**:
  - The Cell class encapsulates the properties value (to display the cell content) and onTap (to handle user interaction) within its scope.
  - These properties are passed to the widget through the constructor, allowing controlled initialization.
- **Encapsulation of UI Logic**:
  - The widget's UI (design and interaction behavior) is encapsulated within the build method, abstracting it away from other parts of the application.

2. Abstraction

- The Cell widget abstracts the behavior and appearance of a single cell in the game board:
  - The game board doesn't need to know how the cell looks or reacts to user interaction; it simply interacts with the cell through the value and onTap properties.
- This abstraction makes it easier to modify or extend the cell's behavior without affecting the rest of the application.

3. Reusability:

- The Cell widget is reusable:
    - It can be instantiated multiple times within the game board to represent each cell of the Tic Tac Toe grid.
    - It can be used in other contexts or games requiring a similar grid-based interface.

4. Inheritance:

- The Cell class inherits from StatelessWidget, which is part of Flutter's widget hierarchy.
    - This inheritance allows Cell to benefit from the lifecycle, rendering logic, and other features provided by the StatelessWidget base class.

5. Modularity:

- The Cell widget is a modular component:
    - It focuses solely on the visual representation and user interaction for a single game cell.
    - This modularity ensures a clear separation of concerns, making it easier to debug and test the widget independently.

6. Composition:

- The Cell widget composes several smaller widgets (GestureDetector, Container, Text) to create a complete UI component.
    - The GestureDetector handles user interaction. ○ The Container manages layout and styling. ○ The Text widget displays the cell's content.
- This demonstrates how object composition is used to build complex UIs from smaller, reusable components.

7. Dynamic Polymorphism:

- Although the widget itself is stateless, the behavior of the onTap callback is determined dynamically at runtime:
    - Different onTap functions can be passed to each Cell instance, allowing varied behavior for each cell while reusing the same widget class.

BUILDING THE UI:

```
@override
Widget build(BuildContext context) {   return
GestureDetector(     onTap: onTap,     child:
Container(       margin: EdgeInsets.all(4.0),       width:
80.0,       height: 80.0,       decoration: BoxDecoration(
color: Colors.blue[100],         border: Border.all(color:
Colors.blue, width: 2.0),
     ),
     child: Center(
child: Text(
value,
        style: TextStyle(fontSize: 32.0, fontWeight: FontWeight.bold),
      ),
     ),
   ),
 );
}
```

GestureDetector:

Wraps the Container to detect user interactions, such as taps.

The onTap callback is triggered when the cell is tapped.

Container:

A box widget used to define the visual structure of the cell.

margin: EdgeInsets.all(4.0):

Adds a margin of 4 pixels around the cell for spacing. **width:**

80.0, height: 80.0:

Sets the dimensions of the cell to 80x80 pixels.

BoxDecoration:

Defines the visual appearance of the Container.

color: Colors.blue[100]:

Sets the background color of the cell to a light blue shade.

border: Border.all(color: Colors.blue, width: 2.0):

Adds a blue border around the cell, 2 pixels wide.

Center:

Centers the child widget (the Text displaying the cell value) within the Container.

Text:

Displays the value of the cell ('X', 'O', or '').

TextStyle(fontSize: 32.0, fontWeight: FontWeight.bold):

Styles the text with a font size of 32 and bold weight for visibility.

MAIN FUNCTION :

void main() {

runApp(TicTacToeApp());

} **void main()**:

The entry point of the application.

runApp(TicTacToeApp());:

Launches the app by injecting the root widget (TicTacToeApp) into the widget tree.

Tic Tac Toe App Class:

```
class TicTacToeApp extends StatelessWidget {   @override
  Widget build(BuildContext context) {
return MaterialApp(       title: 'Tic Tac
Toe',       theme: ThemeData(
primarySwatch: Colors.blue,
```

), home: GameScreen(),);

} } class TicTacToeApp extends

StatelessWidget:

- Represents the root widget of the app.
- Stateless because it does not maintain any state; it only sets up the app's structure and theme.

MaterialApp:

- A Flutter widget that wraps the entire app and provides Material Design features.
- **title: 'Tic Tac Toe'**:
- Specifies the app's title, displayed in the task manager and possibly by the operating system.
- **theme: ThemeData(primarySwatch: Colors.blue)**:
- Sets the app's theme, using a blue color scheme for the primary UI components (e.g., AppBar, buttons).
- **home: GameScreen()**:
- Sets the GameScreen widget as the default screen when the app starts.

OOPS CONCEPTS USED IN THIS CODE :

Encapsulation :

- **Encapsulation of the App**:
  - The TicTacToeApp class encapsulates the entire Flutter app's configuration, including the theme and the home screen (GameScreen).
  - All the internal details related to the app's UI (theme, routing) are hidden within the TicTacToeApp class, and the rest of the application interacts with it as a whole unit.

2. Abstraction:

- **Abstraction of the App's Structure**:

    - The app's structure (theme, routes, screens) is abstracted within the TicTacToeApp class. The app is not concerned with the individual widgets or UI elements inside the app. It only interacts with the higher-level structure (like GameScreen).
- **Abstraction of Theme**:
    - The theme configuration (primarySwatch: Colors.blue) is abstracted within MaterialApp. The app doesn't need to know how the theme is applied; it just specifies it, and Flutter handles the rest.

3. Inheritance:

- The TicTacToeApp class inherits from StatelessWidget, which is part of Flutter's widget system. ○ By inheriting from StatelessWidget, the app can be rendered and rebuilt efficiently without needing internal mutable state.
    - Flutter handles the lifecycle of the widget, including when to rebuild it, making use of the inheritance from StatelessWidget.

4. Modularity:

- **App's Modular Structure**:
    - The app is modular, with a clear separation between the overall app (TicTacToeApp) and the screen displayed (GameScreen).
    - This modularity ensures that the app can easily scale or change without affecting unrelated parts (e.g., changing the theme, adding more screens, etc.).

5. Reusability:

- **Reusable TicTacToeApp Class**:
    - The TicTacToeApp class is reusable across different Flutter applications by modifying only the core functionality (theme, title, home screen) while keeping the app structure intact.
- **Reusable MaterialApp Widget**:
    - The MaterialApp widget is a reusable class in Flutter for setting up a basic app structure. This is part of the composable widget system in Flutter.

6. Composition:

- **Composition of Widgets**:

- The TicTacToeApp class is composed of multiple widgets like MaterialApp, which is itself composed of several smaller widgets (like ThemeData, GameScreen, etc.).
- GameScreen is a custom widget that can be composed of more widgets inside it, showing how multiple classes and objects can be composed together to create complex UIs.

7. Polymorphism:

- Though there isn't explicit polymorphism through inheritance, the use of MaterialApp shows a form of polymorphism: ○

- MaterialApp could be swapped with another app structure or theme, providing different behaviors and looks while maintaining the same interface for the rest of the app.

REFERENCES :

https://github.com/kazim7287

https://github.com/openai
https://github.com/features/copilot