جامعة الإسكندرية
ALEXANDRIA
UNIVERSITY
كلية الهندسة

Computer & Systems Engineering Department

CSE 225: Paradigms

Final Project Phase 1

Contributors:

	Name	ID
1	Adel Mahmoud Mohamed Abd El Rahman	20010769
2	Mohamed Hassan Sadek Abd El Hamid	20011539
3	Mahmoud Attia Mohamed Abdelaziz Zian	20011810
4	Mahmoud Ali Ahmed Ali Ghallab	20011811

1. Problem Statement:

In this part of the project, you will investigate the difference between the features of object oriented and functional paradigms by implementing a generic game engine for drawing game boards 2 times (one using JavaScript and the other using Scala). The engine will support drawing six games: Tic-Tac-Toe, Connect-4, Checkers, Chess, Sudoku, and 8-Queens, however, it should be extensible to draw any other board game (e.g., Go Game). The engine will have only two responsibilities:

1. Drawing the board and pieces
2. Enforce the rules of moving pieces.

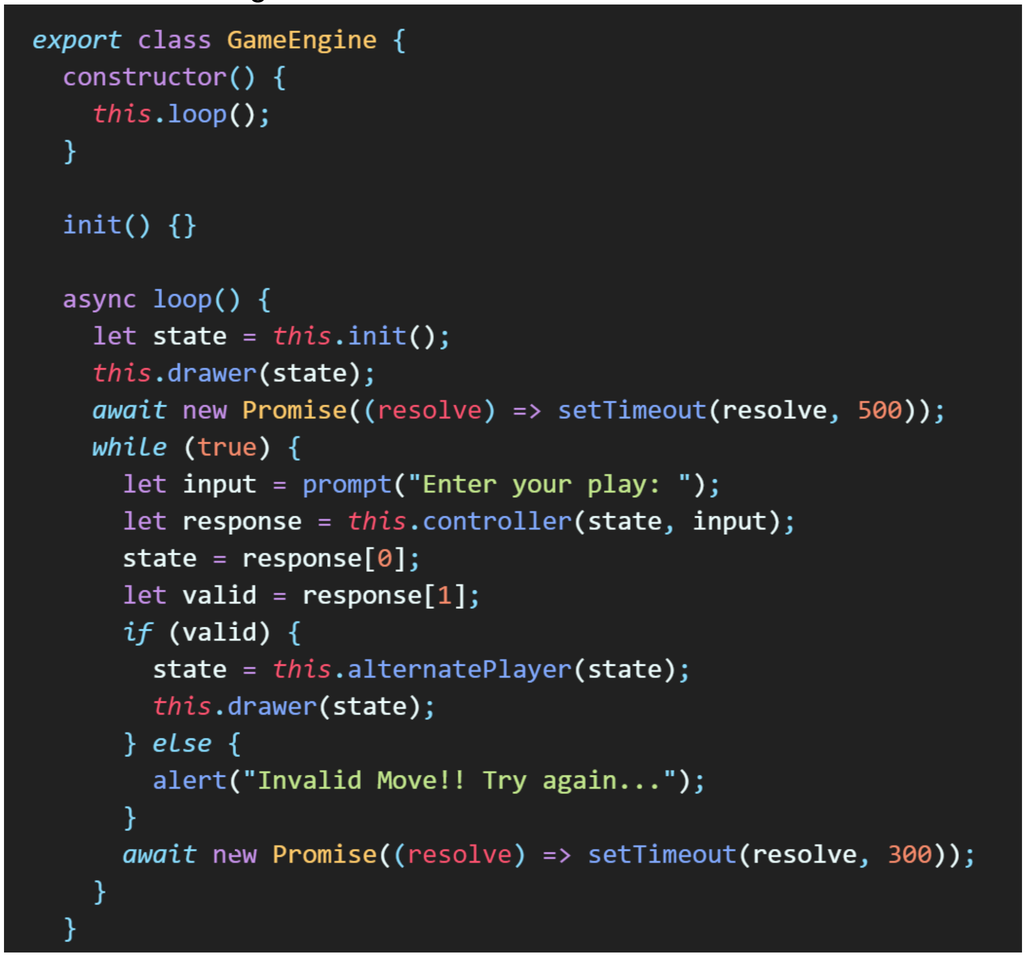
The engine will NOT:

- Have intelligence for playing against the player.
- Check the winner or calculate the scores.
- Save/Load the game status.

2. *Main Features of Each Paradigm:*

- **OOP Paradigm:**

- **Inheritance**: in case of any common logic among classes, it can be implemented in the abstract class **(GameEngine)** and make the classes extend it. This permits code reuse. For example: the game loop, invoking the drawer and controller, alternating the turns, and handling the errors and showing an error message is handled in "GameEngine" abstract class.



```
export class GameEngine {
  constructor() {
    this.loop();
  }

  init() {}

  async loop() {
    let state = this.init();
    this.drawer(state);
    await new Promise((resolve) => setTimeout(resolve, 500));
    while (true) {
      let input = prompt("Enter your play: ");
      let response = this.controller(state, input);
      state = response[0];
      let valid = response[1];
      if (valid) {
        state = this.alternatePlayer(state);
        this.drawer(state);
      } else {
        alert("Invalid Move!! Try again...");
      }
      await new Promise((resolve) => setTimeout(resolve, 300));
    }
  }
```

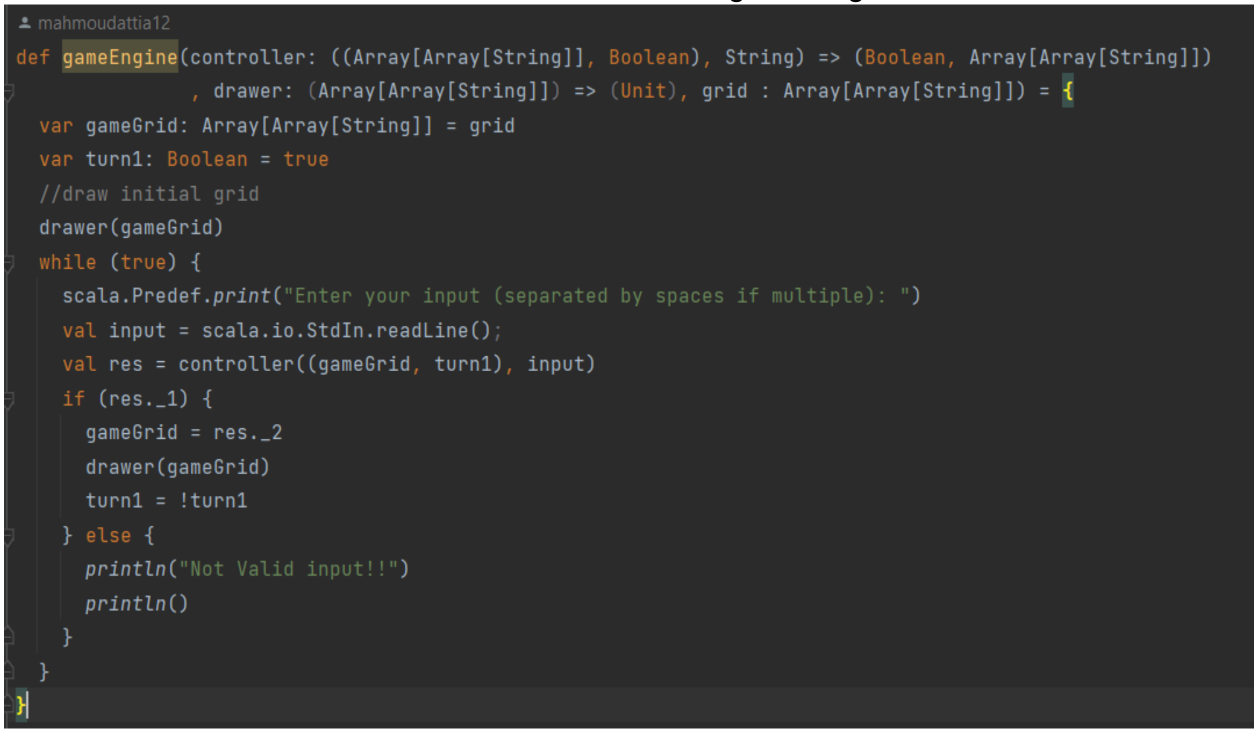
- **Dynamic method selection (Method overriding)**: the drawer and the controller methods are empty in the abstract class, and then implemented in each game class. This is called overriding and this is a powerful feature because any other class (game) that extends **GameEngine** and has its own implementation the drawer controller method then it can work fine, and its methods can be selected dynamically.

- **Encapsulation**: which means that the implementation details of a class are hidden from the outside world by making a class for each game. This can improve security and reduce bugs, as changes made to the class internals won't affect the rest of the system.

- **Abstraction:** it's like riding a bike when we ride a bike, we only know how to ride it but not how it works. We also have no idea how a bike works on the inside, the same idea applies here, the user interacts with the **GameEngine** itself not with the concrete classes of the games.

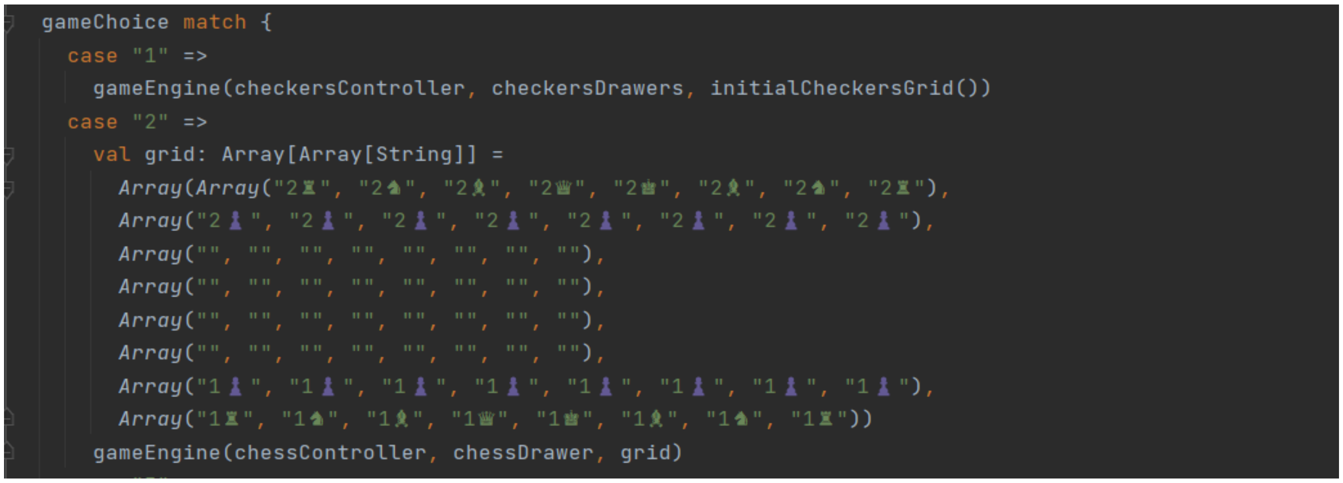
Functional Paradigm:

- **High-order functions:**
    - We applied the feature of high-order functions by passing functions to another function which uses these functions.
    - We applied this feature in the **gameEngine** function which takes as its parameters the drawer function and the controller function of the desired game then uses them to draw and control the chosen game.
    - And this is the definition of the gameEngine function:



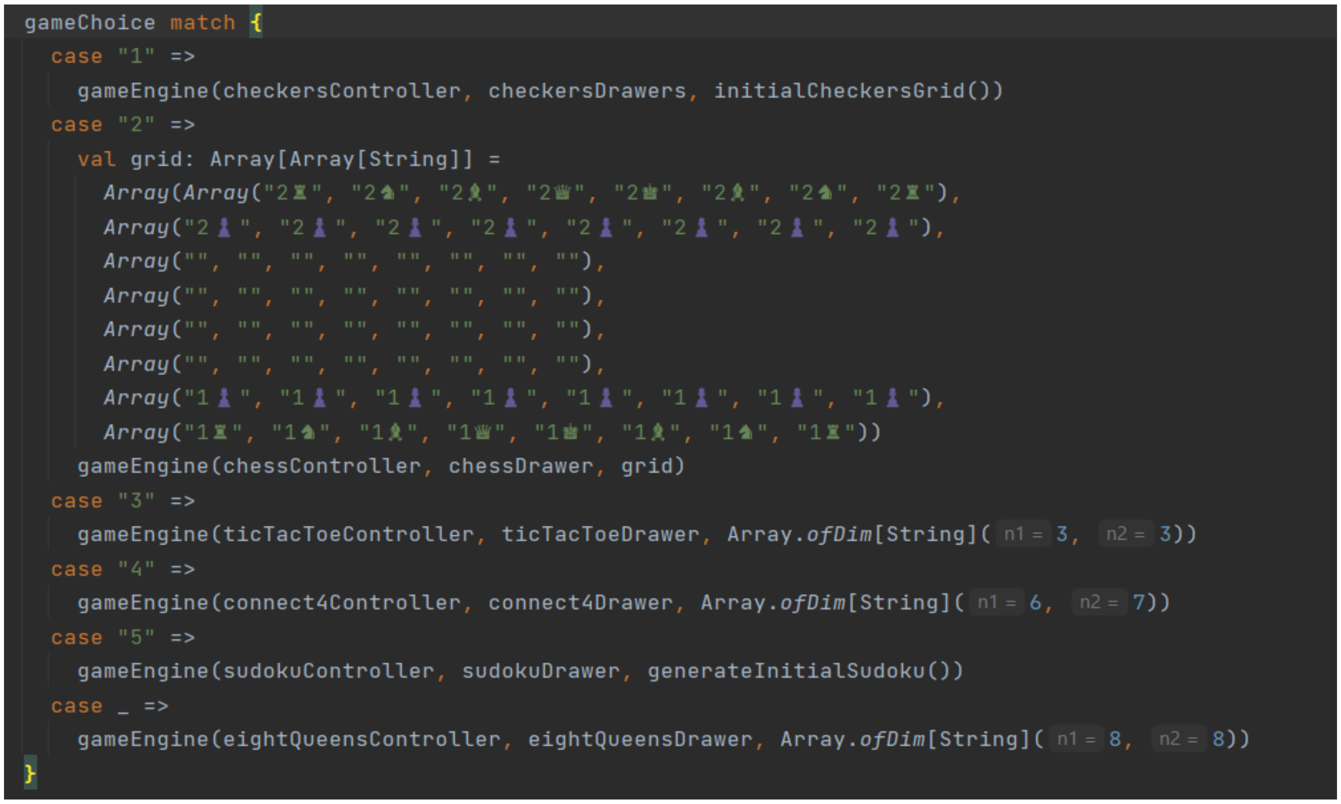
```
mahmoudattia12
def gameEngine(controller: ((Array[Array[String]], Boolean), String) => (Boolean, Array[Array[String]])
               , drawer: (Array[Array[String]]) => (Unit), grid : Array[Array[String]]) = {
  var gameGrid: Array[Array[String]] = grid
  var turn1: Boolean = true
  //draw initial grid
  drawer(gameGrid)
  while (true) {
    scala.Predef.print("Enter your input (separated by spaces if multiple): ")
    val input = scala.io.StdIn.readLine();
    val res = controller((gameGrid, turn1), input)
    if (res._1) {
      gameGrid = res._2
      drawer(gameGrid)
      turn1 = !turn1
    } else {
      println("Not Valid input!!")
      println()
    }
  }
}
```

    - In the **initialScreen** function where I ask the user which game of the 6 games he wants to play I passed to the gameEngine function the Controller and the drawer and the intialBoardGame based on his choice the game Engine starts to use them by draw the initial screen then enters a loop which takes user input, passing the input and the grid to the controller which handles them and return a Boolean for the validity and the newGrid which is passed to the drawer to draw it and so on.



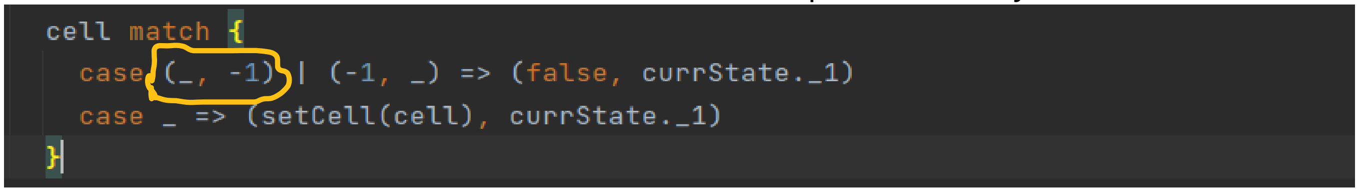
```
gameChoice match {
  case "1" =>
    gameEngine(checkersController, checkersDrawers, initialCheckersGrid())
  case "2" =>
    val grid: Array[Array[String]] =
      Array(Array("2♜", "2♞", "2♝", "2♛", "2♚", "2♝", "2♞", "2♜"),
      Array("2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ "),
      Array("", "", "", "", "", "", "", ""),
      Array("", "", "", "", "", "", "", ""),
      Array("", "", "", "", "", "", "", ""),
      Array("", "", "", "", "", "", "", ""),
      Array("1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ "),
      Array("1♜", "1♞", "1♝", "1♛", "1♚", "1♝", "1♞", "1♜"))
    gameEngine(chessController, chessDrawer, grid)
```

- **Pattern matching:**
  - We applied pattern matching (which is a powerful tool similar to switch statement but with too many additional functionalities) in different situations in our project.
  - We used pattern matching to match a value and do an action depending on this matching which is the first trivial use of pattern matching for example here we match the **gameChoice** and passed on this match we choose the right parameters for the gameEngine function and the last case statement is like the default in the switch statement:



```
gameChoice match {
  case "1" =>
    gameEngine(checkersController, checkersDrawers, initialCheckersGrid())
  case "2" =>
    val grid: Array[Array[String]] =
      Array(Array("2♜", "2♞", "2♝", "2♛", "2♚", "2♝", "2♞", "2♜"),
      Array("2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ ", "2♟ "),
      Array("", "", "", "", "", "", "", ""),
      Array("", "", "", "", "", "", "", ""),
      Array("", "", "", "", "", "", "", ""),
      Array("", "", "", "", "", "", "", ""),
      Array("1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ ", "1♟ "),
      Array("1♜", "1♞", "1♝", "1♛", "1♚", "1♝", "1♞", "1♜"))
    gameEngine(chessController, chessDrawer, grid)
  case "3" =>
    gameEngine(ticTacToeController, ticTacToeDrawer, Array.ofDim[String]( n1 = 3,  n2 = 3))
  case "4" =>
    gameEngine(connect4Controller, connect4Drawer, Array.ofDim[String]( n1 = 6,  n2 = 7))
  case "5" =>
    gameEngine(sudokuController, sudokuDrawer, generateInitialSudoku())
  case _ =>
    gameEngine(eightQueensController, eightQueensDrawer, Array.ofDim[String]( n1 = 8,  n2 = 8))
}
```

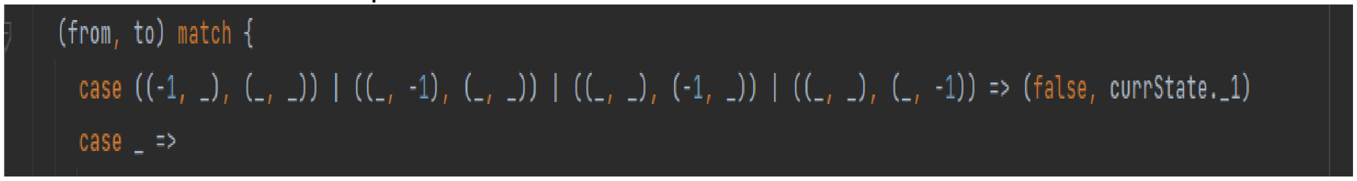
- One of the best things that I found it was better in pattern matching than if statement that it can match a pattern not only a value:



```
cell match {
  case (_, -1) | (-1, _) => (false, currState._1)
  case _ => (setCell(cell), currState._1)
}
```

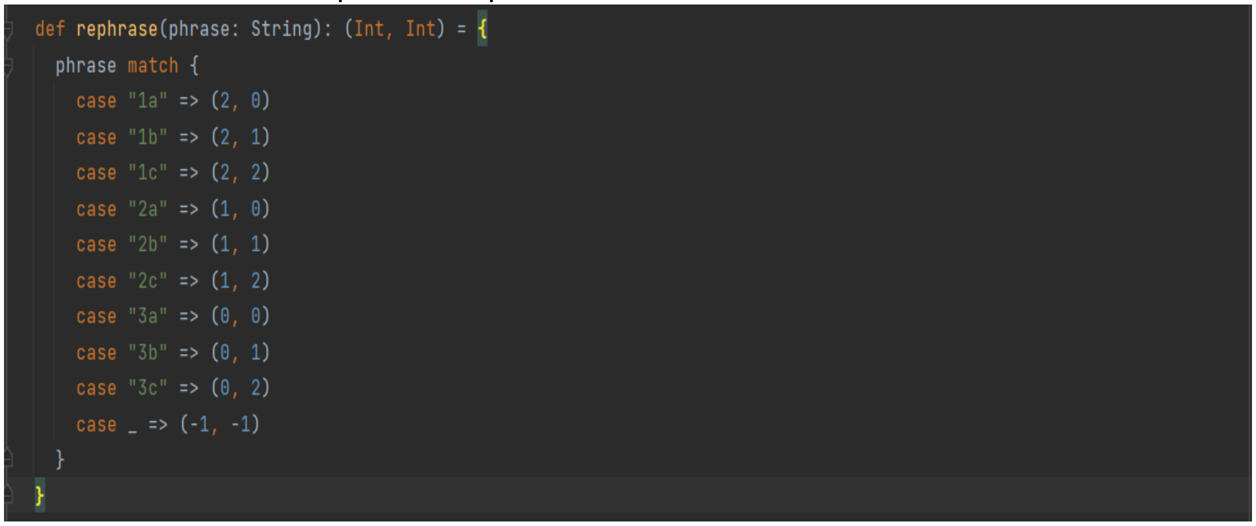
In this example, we can match for example only the second value of the tuple and don't care about the first value which the if statement can't do and should pass to it for example cell._2 if I need to match only the second value.

- Of course we used alternative pattern matching when needed for example:



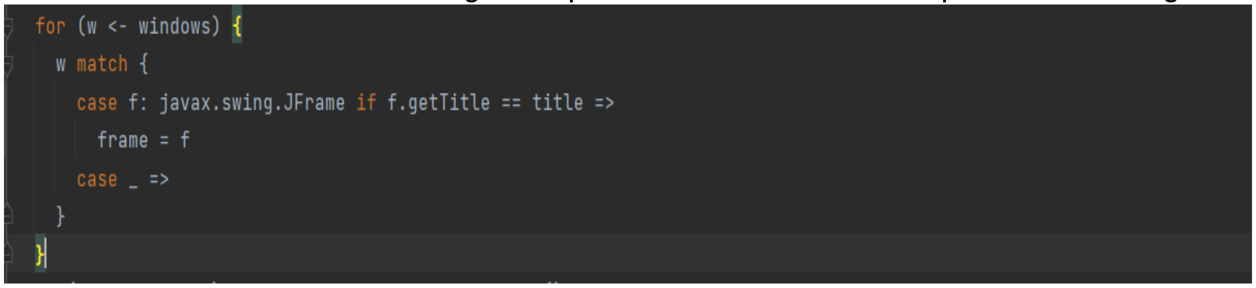
```
(from, to) match {
  case ((-1, _), (_, _)) | ((_, -1), (_, _)) | ((_, _), (-1, _)) | ((_, _), (_, -1)) => (false, currState._1)
  case _ =>
```

- In some case we benefited from pattern matching by using it like a map for example:



```
def rephrase(phrase: String): (Int, Int) = {
  phrase match {
    case "1a" => (2, 0)
    case "1b" => (2, 1)
    case "1c" => (2, 2)
    case "2a" => (1, 0)
    case "2b" => (1, 1)
    case "2c" => (1, 2)
    case "3a" => (0, 0)
    case "3b" => (0, 1)
    case "3c" => (0, 2)
    case _ => (-1, -1)
  }
}
```

- The following example is an enhanced use of pattern matching:



```
for (w <- windows) {
  w match {
    case f: javax.swing.JFrame if f.getTitle == title =>
      frame = f
    case _ =>
  }
}
```

In this example we used pattern matching to check if w in an instance of “javax.swing.JFrame” then match its title with a specific title to perform a specific action.
    - We know that there are much more uses of pattern matching like matching if an object is an instance of a specific class (but we were forbidden from using classes and objects as required) or checking if value (of type Any) is Int/String/… but we didn’t need to use it in the scope of our class.
- **Making the program stateless as much as possible:**
    - making a program stateless is one of the key features of functional programming that we tried to apply in the project. In functional programming, the emphasis is on writing functions that take input and produce output without modifying any global state or relying on mutable data structures. Instead, functional programs are constructed by composing pure functions that transform data immutably.
    - Stateless programs are often more modular and reusable than programs that rely on mutable state. Stateless functions can be composed and combined in many ways, making it easy to build complex programs from smaller building blocks.
    - Stateless programs promote modularity, composability, and ease of testing, and enable better parallelism and concurrency.

2. Main Differences Between the 2 Implementations:

- Focus: OOP focuses on the objects or entities in a program and their interactions, while FP focuses on the behavior of functions and their interactions.
- Mutable vs. Immutable: OOP allows objects to be modified after creation, while FP emphasizes immutability, meaning that data is not changed once it is created.
- Control Flow: OOP uses statements and loops to control the flow of execution, while FP uses functions and recursion.
- Side Effects: OOP allows for side effects to occur, such as modifying global state or accessing external resources, while FP tries to minimize side effects.
- Inheritance: OOP uses inheritance as a way to reuse code and create new classes from existing ones, while FP uses high-order function (composition) to create new functions from existing ones.
- Modularity: OOP typically emphasizes encapsulation and information hiding to create modular programs, while FP emphasizes the creation of small, composable functions.

- State: OOP often relies on maintaining state within objects, while FP avoids mutable states and instead uses pure functions that operate on immutable data. Stateless programs promote modularity, composability, and ease of testing, and enable better parallelism and concurrency.

Overall, OOP and FP have different philosophies and approaches to programming. OOP is well-suited for modeling real-world objects and systems, while FP is often used for mathematical and scientific computations, as well as for building highly concurrent and parallel systems.

3. Pros & Cons of the use of each paradigm:

OOP Paradigm:

- ❖ Pros:
  - ➢ Abstraction
  - ➢ Modularity: OOP paradigm provides a natural way to create modular structure using classes and objects, which is easy to develop and maintain as you can make changes in one class without any dependency on it.
  - ➢ Dynamic Method Selection: each class can redefine extended methods from another class and this will be handled at runtime.
  - ➢ Inheritance: Common logic can be inherited from one class to multi classes

- ❖ Cons:
  - ➢ Overhead in memory and time elapsed in creating classes or manipulating objects.
  - ➢ Mutability: OOP allows for mutable state, which can lead to an unexpected behavior.

Functional Paradigm:

- ❖ Pros:
  - ➢ Modularity and abstraction: Pure functional programming promotes modularity and abstraction, which can help you break down the game engine into smaller, more manageable pieces.
  - ➢ Using pure functions is easier to test and reason about because they don't have side effects and immutable data structures can be safely shared among functions without any risk of accidental modification.
  - ➢ Readability and maintainability: Functional programming languages often have a simple and concise syntax, which can make code more readable

and easier to understand. Additionally, functional programs are often more modular and reusable, which can make them easier to maintain over time.

  - Easier to trace: in functional programming, we divide our program into many small functions each with a specific job so this makes it easier to trace.
  - Stateless architecture: A pure functional programming paradigm enforces a stateless architecture, which can simplify the game engine design and make it easier to reason about the game's behavior. This can also make the game engine more performant by avoiding the overhead of managing the game state as the controller and drawer functions are responsible for handling the game state.

- Cons:
  - Performance: Functional programming languages can sometimes have performance overhead due to the use of immutable data structures and the need to create new objects rather than modifying existing ones as in our drawer functions.
  - There are some types of problems that are difficult to solve in a purely functional way. For example, problems that require mutable state or that rely heavily on side effects may be more difficult to solve in a functional programming language. In my case for example I want to have the turn of the players as a global state that I can alter in the specific functions of 2 players' agent and not alter it in the single agent games but I can't do so and I was forced to alter it in the game engine where I can't know if the user chose single agent or 2 players agent game so I can't use this turn to in a proper way.
  - Limited tooling: Functional programming languages and libraries may have limited tooling and fewer third-party libraries compared to more mainstream programming languages. This can make it more difficult to find solutions to common problems and may require more effort to build and maintain the game engine.
  - Learning curve: Functional programming can be difficult to learn, especially for developers who are used to imperative or object-oriented programming. The functional paradigm requires a different way of thinking about problems and a different set of tools for solving them.

- ***Note***

  the jar file for scala code is in **`out/artifacts/Programming_Paradigms_jar`** directory. If there is a problem in running the jar, you can enter this command in terminal in the jar directory:

```
java -jar Programming-Paradigms.jar
```
