

Game Engine

Programming Paradigms Project

C o n t r i b u t o r s

Ahmed Hesham

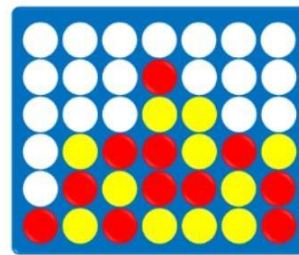
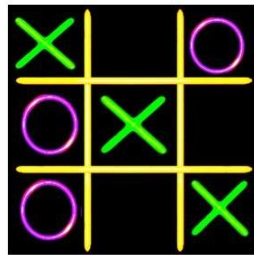
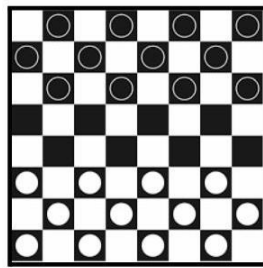
Abdelrahman Wael

Mahmoud Gouda

Mohamed Amin

Part 1: Game Engine

The objective of this project is to explore the distinction between the characteristics of **object-oriented** and **functional programming** paradigms by implementing a generic game engine. The engine will be utilized for drawing game boards, implemented two times, once using JavaScript and once using Scala. Six games will be supported by the engine: *Tic-Tac-Toe*, *Connect-4*, *Checkers*, *Chess*, *Sudoku*, and *8-Queens*. Additionally, the engine should be easily extendable to draw any other board game.



The engine's primary purpose will be to handle two tasks: drawing the board and pieces and enforcing the rules of moving the pieces. This report will examine how the implementation of the game engine differs depending on the programming paradigm utilized, and the advantages and disadvantages of each approach.

Main Features

Scala Implementation

Higher Order Function => Our abstract game engine functions takes 3 functions as parameters which are Drawer, Controller and Init:

```
def AbstractGameEngine(Controller: (Array[Int], (String, Int)) => (Boolean, Array[Int], Int),
    Drawer: Array[Int] => Unit,
    Init: () => Array[Int] ): Unit = {
```

Pattern Matching => We used pattern matching multiple time through out the project to match the pieces to values as in TicTacToe we matched the state value to 3 values:

```
state(i) match {
    case 1 => button.setIcon(new ImageIcon(XSymbol))
    case 2 => button.setIcon(new ImageIcon(OSymbol))
    case _ => button.setText("")
}
```

Pure Functions => Each game has an initializer which generates the same board for every new game (Except sudoku) and takes no arguments:

```
def TicTacToeInit(): Array[Int] = {
    val Board = Array.fill(9)(0)
    Board
}
```

Main Features

Javascript Implementation

Inheritance => It provides reusability of functions of common logic such as sequence in game (Start game, drawGameBoard, drawNames ...) in classes that extend from the parent class (Game).

```
export class Game {  
  
  drawGameBoard(rowNum, colNum, name, ElementType) {...}  
  startGame(init) {...}  
  drawNames(rowNum, colNum){...}
```

Overriding => This feature is used which enables each game class to provide different implementation for a method that is already defined in the parent class. (updateBoard, updateState, isValidMove, ...)

```
class XO extends Game {  
  
  updateBoard(state) {...}  
  
  putPieces(board) {...}  
  isValidMove(state, input) {...}  
  updateState(state, input) {...}  
  
  checkWin(state) {...}  
}
```

Comparison

Main Differences:

In **Scala**, the program is a sequence of functions which call each other. Whereas in **Javascript**, there is a parent class which is extended by each game where each game is a class having its own set of functions.

Pros and Cons:

POC	Pros	Cons
Scala	1) Programs are stateless 2) Well-suited for parallelism	1) Not suitable for all programs
Javascript	1) We can reuse the code multiple times using class 2) Inherit the class to subclass for data redundancy	1) It required a lot of time to create 2) Complex Structure

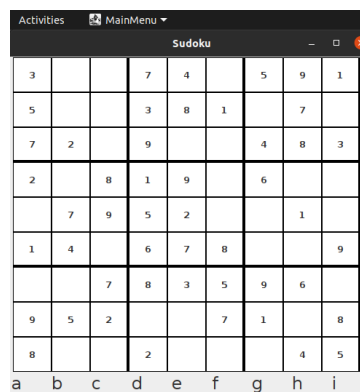
Part 2: Prolog

In this part of the project, you will investigate the features of the logical programming paradigm by implementing a solver for the 2 single player games of part 1: **Sudoku** and **8-Queens**. These Solvers should be implemented using Prolog logical programming language. The prolog is directly connected to the game board throughout the game.

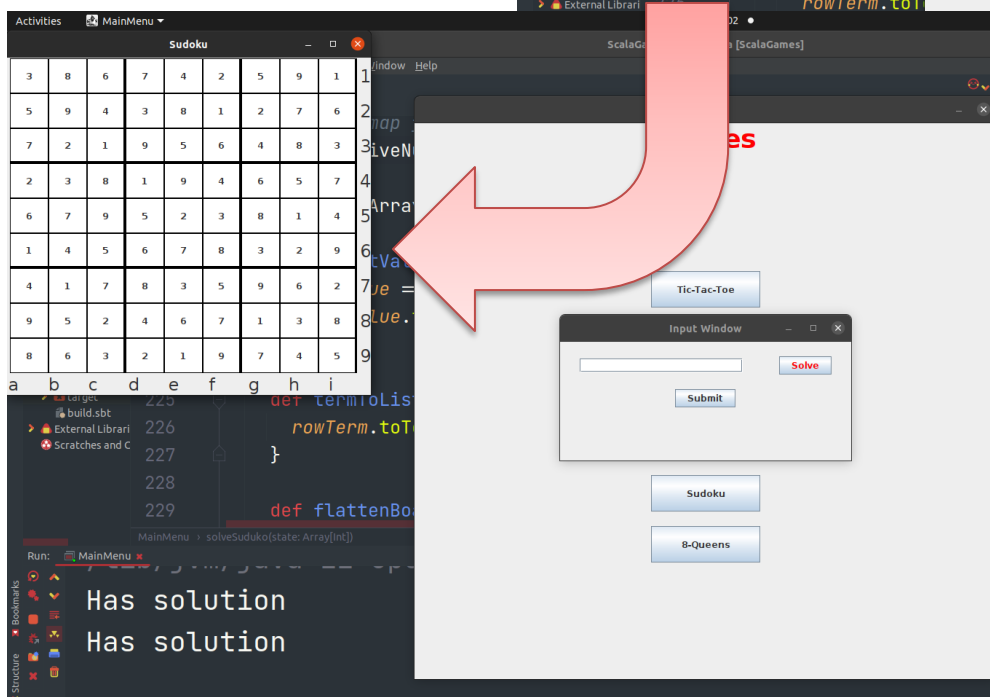
Here are sample runs for the game solvers...

First | Sudoku

This is a normal solve..
Before playing any move the
user clicks on solve button,
the board is solved and sent
to the drawer



3			7	4		5	9	1
5			3	8	1		7	
7	2		9			4	8	3
2		8	1	9		6		
	7	9	5	2			1	
1	4		6	7	8			9
		7	8	3	5	9	6	
9	5	2			7	1		8
8			2				4	5
a	b	c	d	e	f	g	h	i



Has solution
Has solution

Sudoku Solver

5			8	6		1	3	2
8			4					9
7	3	1	9	2	5	6	8	4
9		6	1		2			
2	1			9	4	7	6	8
	7	4	6		8	9	1	
6	2		7		9		4	
1	5					8	9	
		3	5		6	2	1	
a	b	c	d	e	f	g	h	i

5			8	6		1	3	2
8			4					9
7	3	1	9	2	5	6	8	4
9		6	1		2			
2	1			9	4	7	6	8
	7	4	6		8	9	2	1
6	2		7		9		4	
1	5					8	9	
		3	5		6	2	1	
a	b	c	d	e	f	g	h	i

In this case, we inserted a value which leads to the solution of the board. After insertion we call the solver and it finds a solution for it

VALID MOVE
Has solution

Games

Tic-Tac-Toe

Input Window

Solve

Submit

Sudoku

8-Queens

Sudoku Solver

	6	9	7	1		8		
	2	8	9	4	6	1	3	
1			5	2	4	9	6	
		5		7	9	6	8	
9			4			2		5
8				2				9
7	5	1		8	4	9	2	3
6		4	2			5		8
	8	2	5	9	1	7		
a	b	c	d	e	f	g	h	i

Last case, 3 was a valid insert. However, the board can't be solved after that as we have an eight already there in same row. So, solver can't find solution

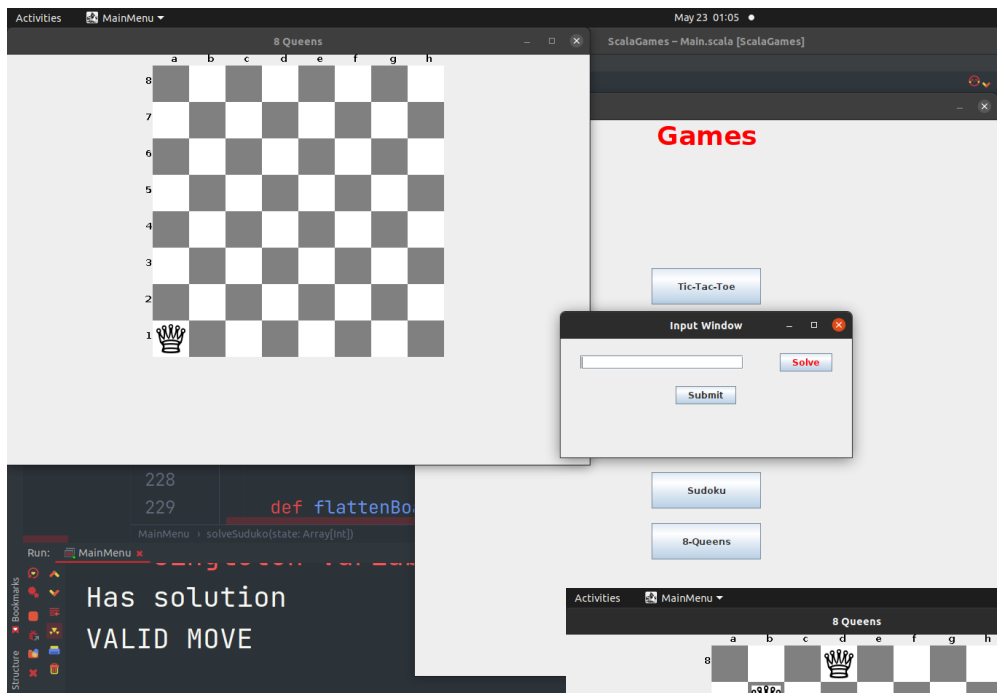
	6	9	7	1				
	2	8	9	4	6	1	3	
1			3	5	2	4	9	6
		5		7	9	6	8	
9			4			2		5
8				2				9
7	5	1		8	4	9	2	3
6		4	2			5		8
	8	2	5	9	1	7		
a	b	c	d	e	f	g	h	i

VALID MOVE
Has no solution !

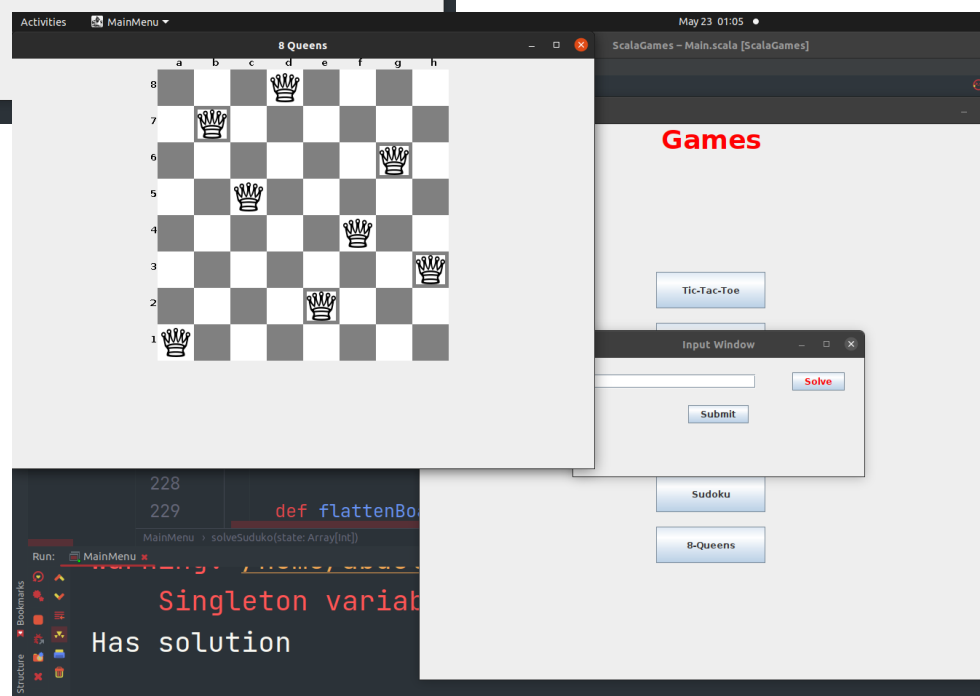
Queens Solver

Second | 8-Queens

The solver for the 8-Queens can either solve the board from the beginning or start from where you called it and solves it (If possible).

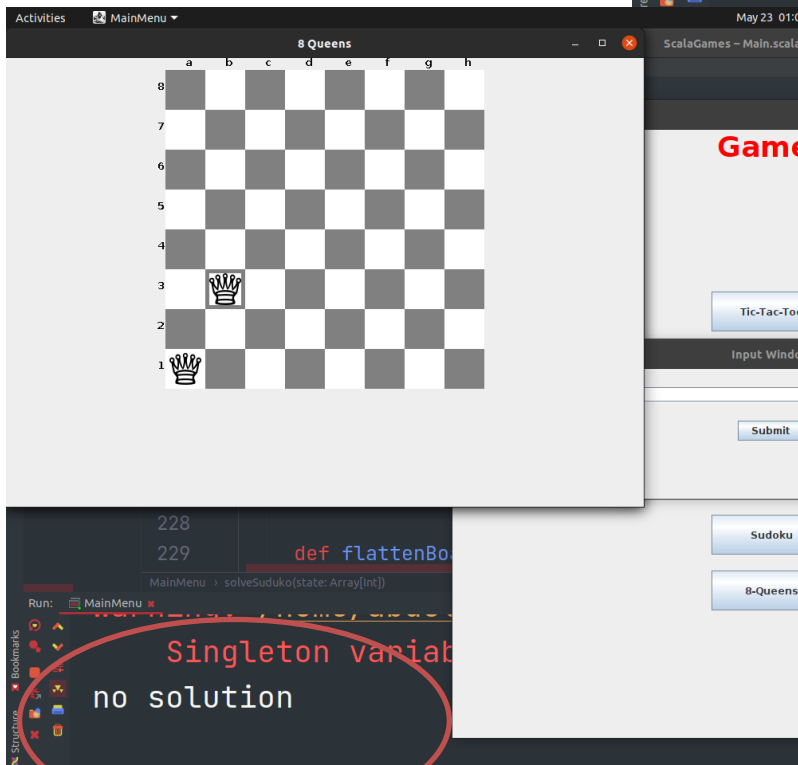
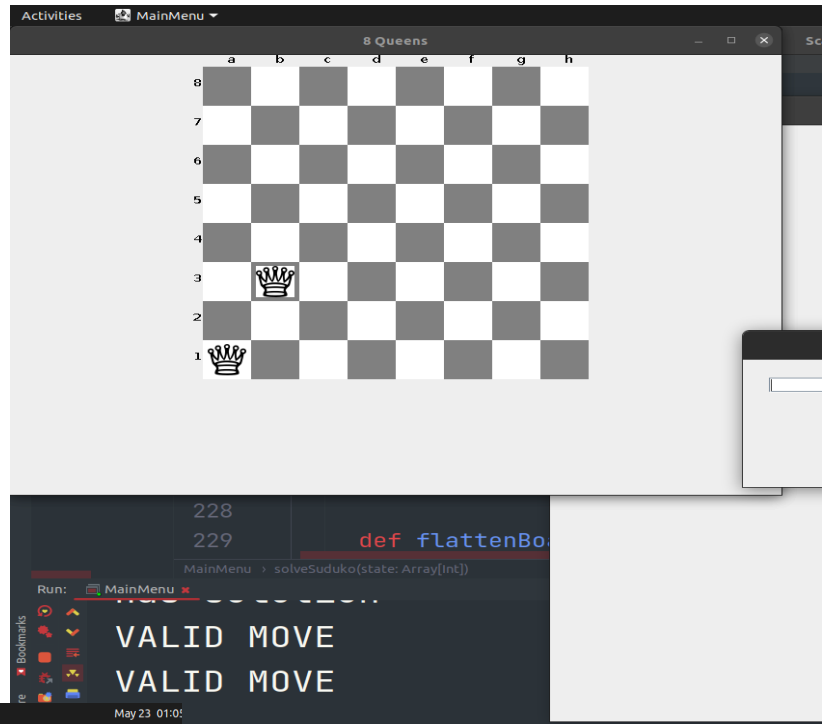


I inserted a queen in 1A
Then called solver and
found a solution after my
insertion



Queens Solver

But in this case, I inserted two queens at 1A and 3B and there is no solution for them. So the solver couldn't find a solution.



For the source code:

<https://github.com/AbdelrahmanWael2/Game-Engine-Scala>