5COSC023W - Tutorial 8 Exercises

As part of the tutorial for this week, you should complete **ALL** the tasks described in the following specifications: (**make sure that you ask questions to your tutor for anything that you do not understand or if you are stuck at any point**).

> Tutorial sessions are practical sessions that you need to work towards the exercises set. They will give you the chance to practice the material learned in the lectures and learn new things as well.
>
> **You should not use these sessions to work towards the assessed assignments!** If you decide to work towards your assessed work instead, then you are not considered as part of the tutorial session. You will not get any help on the code of the assessed work by your tutor but you can ask your tutor ONLY about any clarifications you might need regarding the specification of the coursework.

Like all other modules, you are expected to study towards your module outside the lecture and dedicated tutorial slots for a number of hours. If you do not finish all of the exercises in the tutorial session, make sure that you finish them on your own time and by the end of the week. This is a normal process and part of your university learning.

For all the tasks you should use Jetpack Compose and NOT Views!

1 Extending the Colour Game

Extend the "colour game" implemented in the lecture so that not only the score is saved but also the colour displayed in the top button. I.e. restarting the application (or rebooting the device) leaves the state of the system exactly the same as when the game was abandoned for whatever reason.

2 Extending the Tic-Tac-Toe Game

In the last tutorial we developed a tic-tac-toe game where the computer plays random moves. Here we will implement a number of extended features in it.

The source code we developed can be found at the link below:

`https://ddracopo.github.io/DOCUM/courses/5cosc023w/tic_tac_toe_randomplayer.zip`

If you have not saved your previous implementation, create a new project and copy the src directory from the zip above. You will need to change the package names in the files to match the package name of your newly created project..

2.1 Starting a New Game

Implement a button below the board which every time that it is pressed starts a new game.

Hint: The state of the game is defined by the `grid` list. What will happen if you set all the elements of the list back to the space character (i.e. clear the list). If the game has already finished, do you also need to update the other state variable `gameIsPlayable` of the composable function `GUI`?

2.2 The Computer plays Logically

Implement an intelligent version of the computer player. It will be easier if you create your own `Player` class (which is open, i.e. it can have subclasses) with a single `play` function which is also open (i.e. it can be overridden):

```
open class Player {
    // Random play startegy. Make a random move for the computer and return
    // the index of the move. If the board is full return -1
    open fun play(board: MutableList<Char>): Int {
        var emptySlots = mutableListOf<Int>()

        for (i in 0..board.size-1) {
            if (board[i] == ' ')
                emptySlots.add(i)
        }

        // return one of the empty slots randomly, otherwise if it is empty return a -1
        if (emptySlots.isEmpty())
            return -1
        else
            return emptySlots.random()
    }
}
```

> By default, Kotlin classes do not allow inheritance. I.e. by default, they are closed and the `open` keyword must be used to open the base class for inheritance.
>
> Similarly, functions in subclasses cannot be overridden by default. The corresponding function in the parent class needs to be defined as `open`.

In the original code, replace the line `ComputerMove(grid)` with the line `computerPlayer.move()` where `computerPlayer` is a global variable defined before the activity:

```
// the computer player strategy
var computerPlayer: Player = Player()
```

The `Player` class has 2 subclasses `RandomPlayer` and `LogicalPlayer` which you should implement:

Kotlin uses the : operator to denote inheritance:

```kotlin
open class Animal

// Dog is a subclass of class Animal
class Dog: Animal()
```

```kotlin
open class Player {
    // Random play startegy. Make a random move for the computer and return
    // the index of the move. If the board is full return -1
    open fun play(board: MutableList<Char>): Int {
        var emptySlots = mutableListOf<Int>()

        for (i in 0..board.size-1) {
            if (board[i] == ' ')
                emptySlots.add(i)
        }

        // return one of the empty slots randomly, otherwise if it is empty return a -1
        if (emptySlots.isEmpty())
            return -1
        else
            return emptySlots.random()
    }
}


class RandomPlayer: Player() {
    override fun play(board: MutableList<Char>): Int {
        return super.play(board)
    }
}


class LogicalPlayer(): Player() {
    override fun play(board: MutableList<Char>): Int {
         /***** COMPLETE YOUR IMPLEMENTATION HERE  *****/
    }
```

The intelligent computer (class `LogicalPlayer`) plays according to the following logic:

1. The intelligent player is able to choose winning positions, i.e. if the computer player has already placed two `O`s in a row, column or diagonal then it places the next `O` in the slot which completes three `O`s to win the game.

2. The intelligent computer player is able to defend itself, i.e. if the human player has already placed two `X` in a row, column or diagonal, the computer player will choose the free slot which will prevent the human to win in their next move.

3. If there is neither a winning or defending move, the intelligent player will choose a valid move which creates two `O`s in a row, column or diagonal.

4. If none of the above is applicable the computer player will choose a random valid slot.

2.3 Keeping a Score

Extend the tic-tac-toe game so that it keeps and displays an overall score for the number of wins and losses for each player.

The score should be remembered, i.e. every time the application exits (or the device is rebooted) the last score should be displayed.

Hint: Use a `DataStore`.