5COSC023W - Tutorial 8 Exercises - Sample Solutions

1 Extending the Colour Game

```
package uk.ac.westminster.datastoreexample

import android.content.Context
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.RectangleShape
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.core.edit
import androidx.datastore.preferences.core.intPreferencesKey
import androidx.datastore.preferences.preferencesDataStore
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import kotlin.random.Random

// Create the data store
val Context.datastore: DataStore<Preferences> by preferencesDataStore(name = "settings")

var colours = listOf(Color.Black, Color.Red, Color.Green, Color.Blue,
                     Color.Yellow, Color.White)
var colours_str = listOf("Black",  "Red",  "Green",  "Blue",  "Yellow",  "White")

var correct = 0  // number of correct answers
var total = 0   // number of colours presented to the user
var index = 4  // the index of the colour for the colour_chosen value


class MainActivity : ComponentActivity() {
    lateinit var preferences: Preferences
    lateinit var totalKey: Preferences.Key<Int>
    lateinit var correctKey: Preferences.Key<Int>
    lateinit var indexKey: Preferences.Key<Int>
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // create a preferences key (totalKey) for storing an int value in a name called total
        totalKey = intPreferencesKey("total")
        // create a preferences key (correctKey) for storing an int value in a name called correct
        correctKey = intPreferencesKey("correct")
        // create a preferences key (indexKey) for storing an int value in a name called index
        indexKey = intPreferencesKey("index")

        /* read from the data store */
        runBlocking {
            // the preferences need to be retrieved within a coroutine
            preferences = datastore.data.first()
        }
        // restore the values of total and correct from the data store
        total = preferences[totalKey] ?: 0  // assign 0 if the value in the datastore is null
        correct = preferences[correctKey] ?: 0
        index = preferences[indexKey] ?: 4

        setContent {
            GUI()
        }
    }

    override fun onPause() {
        super.onPause()

        // save in the datastore the values for correct and total
        runBlocking {
            datastore.edit { settings ->
                settings[totalKey] = total
                settings[correctKey] = correct
                settings[indexKey] = index
            }
        }
    }
}



@Composable
fun GUI() {
    var colour_chosen by remember{ mutableStateOf(colours[index]) }

    index = colours.indexOf(colour_chosen)
    val colour_chosen_str = colours_str[index]

    // second colour to be displayed as one of the 2 buttons
    var second_colour_str = colours_str[Random.nextInt(colours.size)]
    while (second_colour_str == colour_chosen_str)
        second_colour_str = colours_str[Random.nextInt(colours.size)]

    // determine whether the correct colour will be displayed as the first
    // or second button - correct_button = 0 for the first button, 1 for the second
    val correct_button = Random.nextInt(2)

    var first_button_label = colour_chosen_str
    var second_button_label = second_colour_str
    if (correct_button == 1) {
        first_button_label = second_colour_str
        second_button_label = colour_chosen_str
    }

    Column (
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
    ) {
        Text("Score: $correct/$total",
            fontSize = 32.sp,
```

```
            modifier = Modifier
                .padding(bottom = 30.dp, end = 40.dp, top=30.dp)
                .fillMaxWidth(),
            textAlign = TextAlign.End)
        Button(
            modifier = Modifier.size(height = 100.dp, width = 100.dp),
            onClick = {},
            shape = RectangleShape,
            colors = ButtonDefaults.buttonColors(containerColor = colour_chosen)) {
        }

        Row (
            modifier = Modifier.padding(top = 30.dp)
        ) {
            Button(onClick = {
                ++total
                if (correct_button == 0)
                    ++correct

                colour_chosen = nextGame(colour_chosen)
            }) {
                Text(first_button_label)
            }

            Button(onClick = {
                ++total
                if (correct_button == 1)
                    ++correct

                colour_chosen = nextGame(colour_chosen)
            }) {
                Text(second_button_label)
            }
        }
    }
}


// choose a new colour to display and make sure it is different than the previous one
fun nextGame(previous_colour_chosen: Color): Color {
    var index = Random.nextInt(colours.size)
    var colour_chosen = colours[index]

    // choose a brand new colour if the same colour was produced
    while (previous_colour_chosen == colour_chosen) {
        index = Random.nextInt(colours.size)
        colour_chosen = colours[index]
    }

    return colour_chosen
}
```

2 Extending the Tic-Tac-Toe Game

Add in the dependencies section of your `build.gradle.kts` (Module:app) Gradle file of the module in your project:

```
dependencies {
    implementation("androidx.datastore:datastore-preferences:1.1.3")
}
```

Add the following in the `colors.xml` file in the resources directory (`res->values`):

```
<color name="yellow">#FFFFCC00</color>
```

The `MainActivity.kt`:

```
package com.example.tictactoe_composableapp

import android.content.Context
import android.os.Bundle
import android.widget.Toast
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.ElevatedButton
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.runtime.toMutableStateList
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.RectangleShape
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.colorResource
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.core.edit
import androidx.datastore.preferences.core.intPreferencesKey
import androidx.datastore.preferences.preferencesDataStore
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking


// dimensions of the board
val rows = 3
val cols = 3

var human_wins = 0
var computer_wins = 0


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

// Create the data store
val Context.datastore: DataStore<Preferences> by preferencesDataStore(name = "settings")

class MainActivity : ComponentActivity() {
    lateinit var preferences: Preferences
    lateinit var humanWinsKey: Preferences.Key<Int>
    lateinit var computerWinsKey: Preferences.Key<Int>

    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)

        // create a preferences key for storing an int value for human wins
        humanWinsKey = intPreferencesKey("human_wins")
        // create a preferences key for storing an int value for computer wins
        computerWinsKey = intPreferencesKey("computer_wins")

        /* read from the data store */
        runBlocking {
            // the preferences need to be retrieved within a coroutine
            preferences = datastore.data.first()
        }

        // restore the values of human_wins and computer_wins from the datastore
        human_wins = preferences[humanWinsKey] ?: 0
        computer_wins = preferences[computerWinsKey] ?: 0

        setContent {
            GUI()
        }
    }

    override fun onPause() {
        super.onPause()

        // save in the datastore the values for human_wins and computer_wins
        runBlocking {
            datastore.edit { settings ->
                settings[humanWinsKey] = human_wins
                settings[computerWinsKey] = computer_wins
            }
        }
    }
}

@Preview
@Composable
fun GUI() {
    var grid = remember{MutableList(rows*cols) {i -> ' '}.toMutableStateList()}
    var gameIsPlayable by remember{ mutableStateOf(true) }

    Column(modifier = Modifier
        .fillMaxSize()
        .padding(top = 100.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy(0.dp)) {

        // display the score
        Text(text = "Score: Human=$human_wins, Computer=$computer_wins",
            modifier = Modifier.padding(bottom = 50.dp, end = 20.dp).fillMaxWidth(),
            textAlign = TextAlign.End, fontSize = 18.sp
        )

        val context = LocalContext.current
        // display the board
        // create 3 rows of buttons
        for (r in 0..rows-1) {
            Row {
                // each row has 3 buttons (columns)
                for (c in 0..cols-1) {
                    var index = r*rows + c
                    Button(shape = RectangleShape,
                           colors = ButtonDefaults.buttonColors(containerColor = colorResource(id = R.color.yellow)),
                           modifier = Modifier
                               .size(width = 80.dp, height = 80.dp)
                               .padding(1.dp),
                           enabled = gameIsPlayable,
                           onClick = {
                               // human move
                               if (grid[index] == ' ') {
```

```
                                grid[index] = 'X'

                                val winner = checkWinner(grid)
                                if (winner == 'X' || winner == 'O') {
                                    if (winner == 'X' || winner == 'O') {
                                        if (winner == 'X')
                                            ++human_wins
                                        else if (winner == 'O')
                                            ++computer_wins
                                    }

                                    displayGameOver(winner, context)
                                    gameIsPlayable = false
                                }
                                else {
                                    // now it is the turn of the computer player.
                                    // Initially this is a naive player choosing
                                    // a random cell among the available ones
                                    var chosen_cell = computerPlayer.play(grid)
                                    if (chosen_cell != -1) {  // if board is not full play the move
                                        grid[chosen_cell] = 'O'

                                        val winner = checkWinner(grid)
                                        if (winner == 'X' || winner == 'O') {
                                            if (winner == 'X')
                                                ++human_wins
                                            else if (winner == 'O')
                                                ++computer_wins

                                            displayGameOver(winner, context)
                                            gameIsPlayable = false
                                        }
                                    }
                                    else {
                                        displayGameOver('-', context)
                                        gameIsPlayable = false
                                    }
                                }
                            }
                            else
                                Toast.makeText(context, "Move is invalid", Toast.LENGTH_SHORT).show()
                        }
                    ) {
                        Text(text = ""+grid[index], fontSize = 32.sp, color = Color.Black)
                    }
                }
            }
        }

        // New Game button
        ElevatedButton(modifier = Modifier.padding(top=30.dp),
            shape = RectangleShape,
            colors = ButtonDefaults.buttonColors(containerColor = Color.LightGray),
            onClick = {
                for (i in 0 until grid.size)
                    grid[i] = ' '

                gameIsPlayable = true
            })
        {
            Text(text ="New Game", color = Color.Black)
        }
    }
}



fun displayGameOver(winner: Char, context: Context) {
    var message = "GAME OVER! "
```

```
    if (winner == '-')
        message += "It is a draw"
    else
        message += "$winner wins"

    Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
}
```

The `Board.kt` file:

```
package com.example.tictactoe_composableapp

/* Check if there was a winner and return 'X' or 'O' for the winner,
   otherwise it returns '-'.  */
fun checkWinner(board: MutableList<Char>): Char {
    var countX = 0  // how many consecutive Xs we have during a check in a row, col or diagonal
    var countO = 0  // how many consecutive Os we have during a check in a row, col or diagonal

    // check the rows first
    for (r in 0..2) {
        for (c in 0..2) {
            var index = row_col2Index(r, c)
            if (board[index] == 'X')
                ++countX
            else if (board[index] == 'O')
                ++countO
        }

        if (countX == 3)
            return 'X'
        else if (countO == 3)
            return 'O'

        countX = 0
        countO = 0
    }

    // check the columns for a winner
    for (c in 0..2) {
        for (r in 0..2) {
            var index = row_col2Index(r, c)
            if (board[index] == 'X')
                ++countX
            else if (board[index] == 'O')
                ++countO
        }

        if (countX == 3)
            return 'X'
        else if (countO == 3)
            return 'O'

        countX = 0
        countO = 0
    }

    // check top-left to bottom-right diagonal
    if (board[0] == board[4] && board[4] == board[8])
        // check that the diagonal cells are not empty
        if (board[0] == 'X' || board[0] == 'O')
            return board[0]

    // check top-right to bottom-left diagonal
    if (board[2] == board[4] && board[4] == board[6])
        // check that the diagonal cells are not empty
        if (board[2] == 'X' || board[2] == 'O')
            return board[2]
```

```
    // no winner
    return '-'
}


/* converts the row and column coordinates in the range [0,2]
   to a single index to access the 1-dimensional grid data structure
   which stores the tic-tac-toe board. Cell (0, 0) is mapped to 0,
   cell (0, 1) is mapped to 1 ... up to the last cell (2, 2) i.e.
   the last row and last column is mapped to 8 */
fun row_col2Index(row: Int, col: Int): Int {
    return row*3 + col
}
```

The `Player.kt` file:

```
package com.example.tictactoe_composableapp

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()
    }
}
```

The `RadomPlayer.kt` file:

```
package com.example.tictactoe_composableapp

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

The `LogicalPlayer.kt` file:

```
package com.example.tictactoe_composableapp

class LogicalPlayer(): Player() {
    /* a more intelligent player following the strategy:
    1. choose winning positions, i.e. if the computer player has already
       placed 2 'O's is a row, column or diagonal then it places the next
       'O' in the slot which completes 3 '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 2 'X's 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 2 'O's is a row, column or
        diagonal.
    4. If none of the above is applicable the computer player will
       choose a random valid slot.
```

```
*/

override fun play(board: MutableList<Char>): Int {
    var chosen_cell = get_winning_or_defend_cell(board)
    if (chosen_cell == null)  // no winning or defense slot - return a random cell
        return super.play(board)
    else
        return chosen_cell
}


/* assuming that the player plays always with symbol 'O':
   this will return a cell that it is either a winning move
   if there are already 2 Os in a row, column, diagonal
   or a defending move if there are already 2 Xs in a
   row, column, diagonal. Otherwise return null */
fun get_winning_or_defend_cell(board: MutableList<Char>): Int? {
    var countX = 0  // how many consecutive Xs we have during a check in a row, col or diagonal
    var countO = 0  // how many consecutive Os we have during a check in a row, col or diagonal

    var emptySlot: Int? = null
    var winningSlot: Int? = null
    var defendSlot: Int? = null

    // check the rows first
    for (r in 0..2) {
        for (c in 0..2) {
            var index = row_col2Index(r, c)
            if (board[index] == 'X')
                ++countX
            else if (board[index] == 'O')
                ++countO
            else  // empty cell
                emptySlot = index
        }

        // if a winning cell or defending cell was found note it down
        if (countO == 2 && emptySlot != null)  // winning move
            winningSlot = emptySlot
        else if (countX == 2 && emptySlot != null)
            defendSlot = emptySlot

        // reset counter values for the next row check
        countX = 0
        countO = 0
        emptySlot = null
    }

    // now check the columns
    // check the columns for a winner
    for (c in 0..2) {
        for (r in 0..2) {
            var index = row_col2Index(r, c)
            if (board[index] == 'X')
                ++countX
            else if (board[index] == 'O')
                ++countO
            else  // empty cell
                emptySlot = index
        }

        // if a winning cell or defending cell was found note it down
        if (countO == 2 && emptySlot != null)  // winning move
            winningSlot = emptySlot
        else if (countX == 2 && emptySlot != null)
            defendSlot = emptySlot

        // reset counter values for the next column check
        countX = 0
        countO = 0
```

```
            emptySlot = null
        }

        // check top-left to bottom-right diagonal
        for (i in 0..2) {
            var index = row_col2Index(i, i)
            if (board[index] == 'X')
                ++countX
            else if (board[index] == 'O')
                ++countO
            else  // empty cell
                emptySlot = index
        }

        // if a winning cell or defending cell was found note it down
        if (countO == 2 && emptySlot != null)  // winning move
            winningSlot = emptySlot
        else if (countX == 2 && emptySlot != null)
            defendSlot = emptySlot

        // reset counter values for the next diagonal check
        countX = 0
        countO = 0
        emptySlot = null

        /*  check the top-right to bottom-left diagonal */
        var x = 2
        var y = 0
        for (i in 0..2) {
            var index = row_col2Index(x - i, y + i)
            if (board[index] == 'X')
                ++countX
            else if (board[index] == 'O')
                ++countO
            else
                emptySlot = index
        }

        // if a winning cell or defending cell was found note it down
        if (countO == 2 && emptySlot != null)  // winning move
            winningSlot = emptySlot
        else if (countX == 2 && emptySlot != null)
            defendSlot = emptySlot

        if (winningSlot != null)
            return winningSlot
        else if (defendSlot != null)
            return defendSlot
        else
            return null
    }
}
```