COURSE: COMP 4411

PROJECT 1 – THE GAME OF LIFE IN KOTLIN

NAME: DHAVAL THANKI

STUDENT NUMBER: 0871347

Objective

The objective of this project was to program the infamous zero-player game of life designed originally by the late John Conway. The game is designed to run infinitely to show the emergence of patterns within cells. I have linked both the functionality file and the main file for easy access to the code.

Functionality

Initial Game state

The initial state of the game is set by the function initCells(), this function will iterate through the whole world and populate it with either ‘live’ or ‘dead’ cells, this is done through a Boolean val called alive. The Boolean val is set via Math.random() and only allowed to set 40% of the initial cells to alive.

```
17      init {
18          initCells()
19          showAllNeighbours()
20      }
21
22      private fun initCells() {
23          for (Col : Int in 0 until height) {
24              for (Row : Int in 0 until width) {
25                  val alive : Boolean = (Math.random() <= 0.40)   // % of cells that will be Alive
26                  addCell(Row, Col, alive)    // add the cells to the world
27              }
28          }
29      }
30
31      private fun addCell(Row: Int, Col: Int, alive: Boolean = false): Cell {
32          if (cellLoc(Row, Col) != null) { throw LocationOccupied()  }  // Throw safety exception
33              val cell = Cell(Row, Col, alive)
34              cells["$Row-$Col"] = cell
35              return cellLoc(Row, Col)!!  // !! - value expected is !null
36      }
```

Refreshing the world for the next state

The Game has a set of rules that the cells need to follow when terminating and spawning, to implement this we use the function checkWorld(). The purpose of this function is to check if each cell has a neighbour and assert whether that cell may live or die in the nextState based on the game rules, additionally, to act as a counter for the number of Generations within the game.

```
43      fun checkWorld() {
44          for ((_ : String , cell : Cell ) in cells) {
45              val aliveNeighbours : Int = isOccupied(cell)  // check if each cell has occupied neighbouring cells
46              if ((!(cell.alive)) && aliveNeighbours == 3) { cell.nextState = 1 } // checking conditions
47              else if (aliveNeighbours < 2 || aliveNeighbours > 3) { cell.nextState = 0 } // checking conditions
48          }
49          for ((_ : String , cell : Cell ) in cells) {
50              if (cell.nextState == 1) { cell.alive = true }  // set the cell state for next iteration of game
51              else if (cell.nextState == 0) { cell.alive = false }
52          }
53          refresh += 1
54      }
```

Check if cell is occupied

The isOccupied functions main usage is to check how many **alive** neighbours occupy the cell that is currently being targeted, this helps in figuring out whether the cell may be allowed to live or die, this is done via counting the number of neighbours there are within a separate ArrayList. The number is then stored in occupied and returned to the calling function.

```
67      private fun isOccupied(cell: Cell): Int {
68          var occupied = 0
69          val neighbours : ArrayList<Cell>  = neighbourList(cell)
70          for (i : Int  in 0 until neighbours.size) {    // iterate through list of neighbours
71              val neighbour : Cell  = neighbours[i]
72              if (neighbour.alive) { occupied += 1 }  // increment if cell has neighbour
73          }
74          return occupied
75      }
```

Checking neighbouring cells

In order to check the neighbouring cells, we must compile a list of neighbours for every cell, this can be done by checking the adjacent cells if they are alive or dead and then storing the values of the ones that are alive or not null.

```
78      private fun showAllNeighbours() {
79          for ((_ : String , cell : Cell ) in cells) {
80              neighbourList(cell)
81          }
82      }
83
84      private fun neighbourList(cell: Cell): ArrayList<Cell> {
85          if (cell.adjCells == null) { // Only if there are none
86              cell.adjCells = ArrayList()
87              for ((rel_x : Int , rel_y : Int ) in checkAdjDirs) { // check all directions
88                  val neighbour : Cell?  = cellLoc( (cell.xCords + rel_x), (cell.yCords + rel_y) ) // assign to (new) cell
89                  if (neighbour != null) { cell.adjCells!!.add(neighbour) }   // add a new neighbour in the list
90              }
91          }
92          return cell.adjCells!!
93      }
94  }
```

```
7   class World(private val width: Int, private val height: Int) {
8       class LocationOccupied: Exception() // safety measure
9
10      var refresh = 0
11      private val cells = HashMap<String, Cell>()
12      private val checkAdjDirs: Array<IntArray> = arrayOf(
13              intArrayOf(-1, 1),  intArrayOf(0, 1),  intArrayOf(1, 1), // Top (X = -1/0/+1, Y = +1)
14              intArrayOf(-1, 0),                     intArrayOf(1, 0), // look left & Look right (X = =1/+1, Y= 0)
15              intArrayOf(-1, -1), intArrayOf(0, -1), intArrayOf(1, -1) // Down (X = -1/0/+1, Y = -1)
```

Rendering the picture

For the rendering or printing process I used the inbuilt StringBuilder() function to simplify the task of outputting all the data, this along with the use of toChar(), a simple function to output a * if thisCell.alive and a space if !thisCell.alive. Each of the elements or cells is appended and then there is a newline per each row. The function can then return this to the calling function where it will then be printed to the screen.

```
55      fun render(): String {
56          val rendering = StringBuilder() // use StringBuilder to manipulate cellLoc val as string output
57          for (Col :Int in 0 until height) {
58              for (Row :Int in 0 until width) {
59                  val cell :Cell = cellLoc(Row, Col)!!  // expect non-null val
60                  rendering.append(cell.toChar())
61              }
62              rendering.append("\n")
63          }
64          return "$rendering"
65      }
```

The Driver

The main driver code along with a timing sequence to check how long each generation of the game takes to load and refresh.

```
11          fun driver() {
12              val world = World(dimensionWidth, dimensionHeight)
13
14              var totalTickTime = 0.0
15              var totalRenderTime = 0.0
16
17              while(true)  {
18                  val tickT0 :Long = System.currentTimeMillis()
19                  world.checkWorld()
20                  val tickT1 :Long = System.currentTimeMillis()
21                  val tickTime :Double = (tickT1 - tickT0) / 1.0
22                  totalTickTime += tickTime
23                  val avgTickTime :Double = (totalTickTime / world.refresh)
24
25                  val renderT0 :Long = System.currentTimeMillis()
26                  val rendered :String = world.render()
27                  val renderT1 :Long = System.currentTimeMillis()
28                  val renderTime :Double = (renderT1 - renderT0) / 1.0
29                  totalRenderTime += renderTime
30                  val avgRenderTime :Double = (totalRenderTime / world.refresh)
31
32                  var output = "Generation #${world.refresh}"
33                  output += " - World refresh time (${formatOutput(avgTickTime)})"
34                  output += " - Rendering time (${formatOutput(avgRenderTime)})"
35                  output += "\n$rendered"
36                  print("\u001b[H\u001b[2J")
37                  println(output)
```

Results