204: Swift Functional Programming

Introduction

We'll consider how to model Conway's Game of Life in an OOP vs a functional style. The starter playground shows the beginning of an implementation in an OOP style, which relies on mutable variables and impure functions that mutate those variables.

In this session, first we'll note how objects (defined via classes) are liable to bugs due to *aliasing*, while values (defined via structs) do not suffer this problem. Second, we'll describe how to represent a board and time-evolution using pure functions and values. Last, we'll show examples of the map and filter, two classic elementary higher-order functions.

Step 1 - Observe a bug due to aliasing

We set only (0,0) alive. Add this line to check if cell (1,0) is alive:

```
g.isCellAliveAt(1, 0)
```

Observe that it is alive! This is a bug due to *aliasing*, the situation where two names refer to the same object.

Because we create the grid by appending the same initial column, and because MCell is a reference type, all the cells in one row point to the object.

The essential problem is that we need to keep track of both *names* and *objects*.

Step 2 - Fix aliasing problem by instance

management

We need to ensure every column contains a distinct array of distinct MCell instances.

Rename the old initializer to a function `init_old`.

Write a new MGrid initializer which moves the loop that creates a column inside the loop that creates a row:

```
init() {
  var grid:Array<Array<MCell>> = Array<Array<MCell>>()
  for columnIndex in 0..<10 {
    var column:Array<MCell> = Array<MCell>()
    for rowIndex in 0..<10 {
      column.append(MCell())
    }

    grid.append(column)
  }

  self.columns = grid
}
```

Now every cell is a distinct instance.

We fixed the aliasing problem by keeping track of aliases.

Step 3 - Fix aliasing problem with a value type

Now we show another way. Restore the old initializer.

Replace the MCell class definition with a struct definition:

```
struct MCell {
  var alive:Bool
  init() { self.alive = false }
}
```

Now it works.

The moral: if you use value types (structs and enums), then you do not need to track the the *identity* of objects separately from their

names. You can think only about values, more like in mathematics.

(You could do even better by using *immutable* values, whether class types or value types. With immutability, aliasing presents no dangers.)

Step 4 - Consider imperative approach to time evolution

Start implementing a function to update the grid:

```
func updateBoard(inout grid:MGrid) -> Void {
  for col in grid.columns {
    for cell in col {
      let isAlive = cell.alive
      // count living neighbors
      // and put results ... ?
    }
  }
}
```

As these loops run, we move across the board setting setting cells.

But how do we set the cell to its new value and then, when computing its neighbors value, correctly use the original cell's old value?

Moral: Mutation is hard to keep track of. What has mutated? When?

So maybe we need to make a copy of the entire board? This is just common sense but it is also "functional" – we're actually just computing a separate and new value for the board, rather than mutating the one board instance through time.

Step 5 - Consider FP approach to board and cell

Ignore everything above, and define this new immutable struct which represents a *living cell*:

```
struct Cell
{
  let x:Int
```

```
  let y:Int
}

// MARK: Equatable
/// Two cells are equal iff all their members are equal
func ==(lhs: Cell, rhs: Cell) -> Bool {
  return lhs.x == rhs.x && lhs.y == rhs.y
}

// MARK: Hashable
extension Cell : Hashable {
  var hashValue : Int { return self.x.hashValue ^ self.y.hashValue  }
}
```

(Implementing Equatable and Hashable is a necessary pain. You always need equality.)

We do not need a special data structure to represent the grid at all. We can just use an array. Consider this representation for a board with one live cell:

```
let initialBoard = [Cell(x:0,y:0)]
```

We will represent evolution through time as computing one entire immutable value for the whole board from another. This is immutable and simple.

Step 6 - Generating a cell's neighbors:

Paste in this function for generating a cell's neighbors:

```
func neighbors(OfCell cell:Cell) -> [Cell]
{
  let deltas = [(-1,-1),(0,-1),(1,-1),
                (-1, 0),       (1,0 ),
                (-1, 1),(0, 1),(1, 1)]

  var neighbors = [Cell]()
  for delta in deltas {
    neighbors.append( Cell(x: cell.x + delta.0, y: cell.y + delta.1) )
  }
  return neighbors
}
```

This loops over every possible movement away from a cell to build a list of its neighboring cells. That is, it loops over a collection to build a new collection – a common pattern.

Step 7 - Use “map” to simplify neighbor computation

It’s a pattern because there’s a tedious part that’s always the same in every program, and an interesting part that is custom to what we are doing. Map is essentially a helper function that takes care of the tedious part, and lets you focus only on the interesting part.

The function `map` is used to generate a new collection by applying a transform to every element in a collection.

The boring part is creating a new empty collection, looping through values of the old collection, and appending a new value to the new collection. The interesting part is really just the code that does that transforming, that creates the new value.

The function `map` takes two arguments, the *input array* and a *transforming function*, and returns an *output array*.

So we need to look at the loop code we already have, and identify those three parts. Input array: `deltas`. Output array: `neighbors`. Tranforming function: `(delta) -> Cell`.

Past this into `neighbors` to define this transforming function as an inner function:

```
func transform(delta:(Int,Int)) -> Cell {
  return Cell(x: cell.x + delta.0, y: cell.y + delta.1)
}
```

That will give the following:

```
func neighbors2(OfCell cell:Cell) -> [Cell]
{
  let deltas = [(-1,-1),(0,-1),(1,-1),
    (-1, 0),       (1,0 ),
    (-1, 1),(0, 1),(1, 1)]

  func transform(delta:(Int,Int)) -> Cell {
    return Cell(x: cell.x + delta.0, y: cell.y + delta.1)
  }

  var neighbors = [Cell]()
  for delta in deltas {
    neighbors.append( transform(delta) )
  }
```

```
  return neighbors
}
```

Now we replace the loop with the `map` function:

```
func neighbors3(OfCell cell:Cell) -> [Cell]
{
  let deltas = [(-1,-1),(0,-1),(1,-1),
    (-1, 0),       (1,0 ),
    (-1, 1),(0, 1),(1, 1)]

  func transform(delta:(Int,Int)) -> Cell {
    return Cell(x: cell.x + delta.0, y: cell.y + delta.1)
  }

  let neighbors = map(deltas,transform)
  return neighbors
}
```

We can make this even simpler by not using an inner function, but using a *closure*. You can think of the inner function as just a named closure:

```
func neighbors4(OfCell cell:Cell) -> [Cell]
{
  let deltas = [(-1,-1),(0,-1),(1,-1),
    (-1, 0),       (1,0 ),
    (-1, 1),(0, 1),(1, 1)]

  let transform:(Int,Int) -> Cell = {
    (delta:(Int,Int)) -> Cell in
    return Cell(x: cell.x + delta.0, y: cell.y + delta.1)
  }

  let neighbors = map(deltas,transform)
  return neighbors
}
```

Or instead of assigning the closure to the constant `transform`, we can just place it inline directly:

```
func neighbors5(OfCell cell:Cell) -> [Cell]
{
  let deltas = [(-1,-1),(0,-1),(1,-1),
    (-1, 0),       (1,0 ),
    (-1, 1),(0, 1),(1, 1)]

  let neighbors = map(deltas,{
    (delta:(Int,Int)) -> Cell in
    return Cell(x: cell.x + delta.0, y: cell.y + delta.1)
  })
  return neighbors
```

```
}
```

Another iteration: return map's return value directly:

```
func neighbors5(OfCell cell:Cell) -> [Cell]
{
  let deltas = [(-1,-1),(0,-1),(1,-1),
    (-1, 0),       (1,0 ),
    (-1, 1),(0, 1),(1, 1)]

  return map(deltas,{
    (delta:(Int,Int)) -> Cell in
    return Cell(x: cell.x + delta.0, y: cell.y + delta.1)
  })

}
```

(We could make this even shorter by switching to shorthand syntax for closures which relies on type inference but we'll skip that here.)

Step 8 - Add time-evolution logic with raw loops

Paste in the following time-evolution function:

```
/// compute next cells, using no HOFs
func activeCellsOneStepAfter(activeCells:[Cell]) -> [Cell]
{
  // STAGE 1. loop to build array of every "neighboring", i.e., every inst
ance of a living cell being neighbored
  // i.e., "map" every active cell to an array of its neighbors, then conc
atante the arrays.
  var neighborings = [Cell]()
  for cell in activeCells {
    for neighborOfCell in neighbors(OfCell: cell) {
      neighborings.append(neighborOfCell)
    }
  }

  // STAGE 2. loop and count duplicate neighborings
  // i.e. "reduce" an array of Cells to an array of (Cell,Int) tuples coun
ting duplicates
  var neighboringsPerCell = Dictionary<Cell,Int>()
  for neighboringCell in neighborings {
    if let value = neighboringsPerCell[neighboringCell] {
      neighboringsPerCell[neighboringCell] = value + 1
    }
    else {
      neighboringsPerCell[neighboringCell] = 1
    }
  }
```

```
  // STAGE 3. loop to filter only neighborings with certain counts
  // "filter" an array of (Cell,Int) tuples based on the Int value and oth
er conditions
  var neighboringsPerCellActiveNextStep = [(Cell,Int)]()
  for (theNeighbor,neighborCount) in neighboringsPerCell {
    if       (neighborCount == 3) ||
      (neighborCount == 2 && find(activeCells,theNeighbor) != nil)
    {
      neighboringsPerCellActiveNextStep.append((theNeighbor,neighborCount)
)
    }
  }

  // STAGE 4. loop to gather only the neighborings themselves
  // "map" an array of (Cell,Int) tuples to an array of Cells
  var neighboringsActiveNextStep = [Cell]()
  for (theNeighbor,neighborCount) in neighboringsPerCellActiveNextStep {
      neighboringsActiveNextStep.append(theNeighbor)
  }

  // result: the cells alive at the next step in time
  return neighboringsActiveNextStep
}
```

How does this implement Life? We build up a list of all “neighborings” – all instances of any cell neighboring any live cell. This list will include the same cell multiple times if that cell is a neighbor to multiple distinct live cells.

Then for every cell *next* to such a neighboring, we count the number of neighborings it has, and we use that count to determine whether it is alive in the next time step.

(This is tricky. It works because Conway’s algorithm is spatially local. For instance, if the rules allowed certain patterns of live cells to cast spores many spaces away, then just looking near the neighborings would not suffice.)

Step 9 - Use “filter” to simplify stage 3

The `filter` function is much like `map`. It handles the tedious boilerplate part in a common pattern of code. The pattern is creating a new collection which contains only certain items from an existing collection.

`filter` takes two arguments – the *input sequence* which is to be filtered, and a *predicate* function which tells when an item should be

kept. It returns an *output array*, of the items that have been kept.

To rewrite STAGE 3 to use `filter`, we need identify those three parts in the raw loop. Input array: *neighboringsPerCell*. Output array: *neighboringsPerCellActiveNextStep*. Predicate: the conditional clause which tests the count and theNeighbor.

As with map, we can begin by extracting the predicate in the raw loop into an inner function:

```
func predicate(theNeighbor:Cell,neighborCount:Int) -> Bool {
  return (neighborCount == 3) ||
    (neighborCount == 2 && find(activeCells,theNeighbor) != nil)
}

var neighboringsPerCellActiveNextStep = [(Cell,Int)]()
for (theNeighbor,neighborCount) in neighboringsPerCell {
  if predicate(theNeighbor, neighborCount)
  {
    neighboringsPerCellActiveNextStep.append((theNeighbor,neighborCount))

  }
}
```

Now we can replace the loop with the `filter` function:

```
func predicate(theNeighbor:Cell,neighborCount:Int) -> Bool {
  return (neighborCount == 3) ||
    (neighborCount == 2 && find(activeCells,theNeighbor) != nil)
}
let neighboringsPerCellActiveNextStep = filter(neighboringsPerCell, predic
ate)
```

Finally, we can dispense with the inner function and use a closure:

```
let neighboringsPerCellActiveNextStep = filter(neighboringsPerCell,
  { (theNeighbor:Cell,neighborCount:Int) -> Bool in
    return (neighborCount == 3) || (neighborCount == 2 && find(activeCells
,theNeighbor) != nil)
})
```

As with `map`, we could shorten it further using closure shorthand syntax.