# CSCI E65g: Mobile Application Development Using Swift and iOS

## Fall 2018

## 2-dimensional thinking with overloaded operators

When dealing with a grid, it’s important to shift our thinking to 2 dimensions while remaining in the domain of whole numbers. The danger of bounds errors grows the more bounds there are.

Every grid location has a horizontal x-coordinate (or column-number) and a vertical y-coordinate (or row-number). The first stumbling block is remembering that the column number in the grid domain is the *x* coordinate in the Cartesian domain, and that row maps to *y* values. A helpful visualization is that columns are fixed vertical stripes and as we move left or right along the x dimension, the column number decreases or increases. The second stumbling block is that positive *y* (row) values travel *up* the Cartesian plane in standard diagrams, but *down* the screen (or down the spreadsheet).

It would be very helpful to refer to the concept of a 2-dimensional location as a single value when programming. For this to work, everyday operators should work just as conveniently for our compound type as they do for integer types. It takes some extra work to delve into these advanced features, but the result is extremely readable and more importantly, less bug-prone.

First, we’ll define our basic two-dimensional type. This could easily be a struct, but we’ll be creating tuples frequently out of components, and the notation of tuples allows us to build them without referring to an init method. We’ll stick with row-and-column notation as the more common terminology for grid-based games and simulations.

/\* Either a location on the grid, or the extent of the entire grid \*/

typealias GridCoord = (row: Int, column: Int)

Continuing with the idea of one concept ⇔ one type, we’d like to encapsulate the idea of the allowable grid sizes in a single type.

/\* A constraint on the possible extents of a grid \*/

typealias GridLimits = (lower: GridCoord, upper: GridCoord)

We want to be robust in light of Swift runtime behavior of crashing immediately on Array bounds violations. That involves a lot of comparisons. Without a custom operator there will be many expressions of the form:

if wantedIndex >= 0 && wantedIndex < someArray.count { … }

What we’re really saying is: “if wantedIndex is within the array bounds” or equivalently “if wantedBounds is a valid array index”. We want to be as concise (or more) in Swift than in English. It’s a small convenience but adds up quickly over large projects. Swift already provides:

if someArray.indices.contains(wantedIndex)

but this is almost as much typing and requires a Range object or access to the array.

So let us be demanding about compact but readable syntax. If the gains are marginal, well, we’d like to illustrate some Swiftian power anyway and see where it goes. Once the door is open to this technique, it may gain steam as the model grows, so with a little optimism, we introduce…

### The WITHIN operator

infix operator <=>: ComparisonPrecedence

(If we feel math-obsessed and in for some Unicode punishment, we can actually just use ∈ instead of an ASCII-compatible trigraph.)

This declares a new binary operator where the operands go on either side, called an infix operator, just like + or \*. As far as working out all the operator precedence rules, we’ll just use the existing ComparisonPrecedence already used for the existing comparison operators.

Now that Swift understands this new hieroglyphic as legitimate, parseable infix syntax, we have to define how it works. We rely heavily on overloading. First, the simplest variant. Since the entire approach is array based, we always assume 0 is the lower bound.

func <=>(index: Int, arraySize: Int) -> Bool {

return index >= 0 && index < arraySize

}

Now we can say:

guard wantedIndex <=> arraySize else {

return nil

}

If the operator should work directly on the array (why not), overload some more:

func <=><T>(index: Int, array: Array<T>) -> Bool {

return index >= 0 && index < array.count

}

So, 3 <=> 4 is true, because 3 is a valid index of a 4-element array, but 3 <=> 3 is false; and 3 <=> [15, 10, 5, 0] is true and 3 <=> ["a", "b", "c"] is false for the same reasons.

Interesting, at least?

Returning to the 2-dimensional issue, it’s pretty easy to define it with the same pattern. Here we’re treating the left side as a point within the grid, and the right side as describing the bounds of a 2D array. Let’s give some real muscle to tuples.

/\* Is 'point' valid within 'bounds'? \*/

func <=>(point: GridCoord, bounds: GridCoord) -> Bool {

return point.row >= 0 && point.row < bounds.row

&& point.column >= 0 && point.column < bounds.column

}

Rather doing even more overloading to make the WITHIN operator work directly with a 2D array, we recognize that a computed property that creates a bounds object from the array is important for the external interface of the model. Assume the internal 2D array is called \_grid, and that all rows are the same size:

var bounds: GridCoord {

return (row: \_grid.count, column: \_grid.count[0].count)

}

(Did you find the potential bug however unlikely it may occur? Fix it.) Finally, the payoff for all this work:

func someGridAccess(untrustedInput: GridCoord) -> ContentType? {

guard untrustedInput <=> bounds else {

return nil

}

// No array bounds errors now

let content = \_grid[untrustedInput.row][untrustedInput.column]

…

}

Reviewing the alternative:

func someGridAccess(row: Int, col: Int) -> ContentType? {

guard row >= 0 && row < \_grid.count && col >= 0 && col < \_grid[0].count else {

return nil

}

…

}

Is it worth it? Depends on how much use, in the long term.

Time to bring in the aggregate type GridLimits. The grid model initialization requires checking the requested grid size within limits, so let’s make it easier using this operator pattern. This time we treat the left side as a possible size, and the right side is a set of limits on the size.

func <=>(size: GridCoord, sizeLimits: GridLimits) -> Bool {

return size.row >= sizeLimits.lower.row && size.row <= sizeLimits.upper.row &&

size.column >= sizeLimits.lower.column && size.column <= sizeLimits.upper.column

}

Now an input check might look like:

if wantedBounds <=> MyModel.Limits { /\* valid limits, proceed with initialization \*/ }

The whole idea extends easily to more than 2 dimensions and to all built-in and 3rd-party integer types (see the [BinaryInteger](https://developer.apple.com/documentation/swift/binaryinteger) protocol).

## Rounding out the syntax

Once we create an concept like this, it has a pervasive effect: we rely on it and expect it everywhere. Otherwise, other parts of the code look oddly dissonant. In particular, subscript access and initialization are still broken down by component. Fortunately, we have the ability to add 2D syntax using *constrained extensions*.

For the particular case of an Array of Array of a given type, we want to add an initializer.

extension Array where Array.Element == Array<Player> {

init(repeating repeatedVal: Player, extents: GridCoord) {

let oneRow = [Player](repeating: repeatedVal, count: extents.column)

self.init(repeating: oneRow, count: extents.row)

}

}

It simply breaks down the coordinate dimensions into the nested calls to the built-in init:repeating:count: method. Not a huge functionality add, but like the previous enhancements, it achieves a lifelong goal (isn’t it yours too?) of *pushing the details down to where they belong*. Any time we wish to create a 2-dimensional array, the idea will have a simpler expression:

let newGrid = [[Player]](repeating: .nobody, extents: (5, 5))

Not yet satisfied, we still dislike grid accesses like:

func getValue(at loc: GridCoord) -> Player {

return newGrid[loc.row][loc.column]

}

This can add up to a lot of boilerplate. So, let’s overload the subscript operator to accept a GridCoord as well. The following also goes inside the above extension. Note how brevity implies the need for an unnamed parameter.

subscript(\_ indexTuple: GridCoord) -> Player {

get {

return self[indexTuple.row][indexTuple.column]

}

set {

self[indexTuple.row][indexTuple.column] = newValue

}

}

And now we have the magical:

func getValue(at loc: GridCoord) -> Player {

return newGrid[loc]

}

Only now is our GridCoord type truly a first-class Array accessor.

## Representing the Grid on-screen: A custom UIView

Switching perspectives to the View: We know the grid is only a logical entity, and it needs to fit neatly and predictably on-screen. Continuing in the vein of one expression in Swift per actual concept, the following would be very helpful. Then the using code can focus on other issues entirely, like interacting with the iOS touch model, UIBezierPaths, UIView API, or the MVC interaction. There’s enough to worry about!

The implementation is left as an exercise.

class CustomView: UIView {

/\* The size in points of a drawn grid location \*/

private var rectRenderSize: CGSize { … }

/\* Translate screen coordinates into logical coordinates. Screen coordinates arrive as CGPoints. \*/

private func toLogical(from displayPoint: CGPoint) -> GridCoord { … }

/\* Translate logical coordinates into screen coordinates. Because we draw rectangles not points,

this is more convenient. \*/

private func toDisplay(from logicalPoint: GridCoord) -> CGRect { … }

}