# CSCI E65g: Mobile Application Development Using Swift and iOS

## Fall 2018

## Plowing through Arrays and Dictionaries

Let’s review the methods that take every element from a group of elements and does something with each of them in turn. The most straightforward grouping type is Array. All these related but different methods are designed to express the various types of iteration more *specifically*, *succinctly* and *more powerfully*.

* Specific: Programming has several idiomatic types of iteration. Sometimes we’re transforming one array into another; sometimes we’re boiling an array down into some kind of summarized value; sometimes we’re searching for a particular value. By using these built-in methods (map:\_:, reduce:\_:, and index:of:respectively) we are stating, to ourselves, the compiler, and all future collaborators, which idiom we mean. Once we’ve made our choice, we have to stick with it: A map operation results in an array output, never a single value, and the compiler enforces this with type checking.
* Succinct: The iteration code has already been written in the Swift standard library, so we never have to write a loop, or declare or mishandle an indexing variable again. This may not seem like much but it is a major source of bugs in programming. It *also* means that the iteration can result directly in a value, which can appear nested in other expressions. Some programmers find this very elegant and much closer to the intended mathematical idea. This is impossible with for loops which are *statements* but not *expressions*.
* More powerful: Once the idea of a unit of work has been captured in a helper function, re-factoring is easy. We can name the helper function and use it in many places, or parameterize it: not assume it is a particular function, but allow it to be passed in by the calling function!

Remember that functions are objects in Swift like any other. They can be named or anonymous. They can be defined as local or global variables, and declared as properties of classes and structs. They can be passed as parameters. All of these capabilities are the same as for enums, and classes. Like classes, functions are reference types.

Very often the helper function is very short and highly specialized. It is not worth giving it a name and taking up extra space in our code listing. In this case we define the function in surrounding function call that uses it as a parameter. Because computer science theory, common developer jargon, and official Swift terminology all come into play, we have several synonyms we must learn by heart:

* closure (Computer Science, Swift)
* lambda expressions (Computer Science, other languages)
* Block (Objective C)
* anonymous function (Computer Science, JavaScript)
* function literal (general term, JavaScript)

We use [official Swift terminology: Closure](https://docs.swift.org/swift-book/LanguageGuide/Closures.html).

## A walk through some Collections

Let’s begin with a simple, friendly declaration in the REPL:

let intArray: Array<Int> = [ 1, 2, 3, 4, 5 ]

Ever helpful, the REPL gives instant feedback. Note the helpful type and size annotations at top, and the curly braces that indicate grouping (technically, an Array is actually a struct, but a sort of magic, special one without fixed properties.)

intArray: [Int] = 5 values {

[0] = 1

[1] = 2

[2] = 3

[3] = 4

[4] = 5

}

There a number of closure-based functions that walk through the array and do something useful with it. Each time the closure (anonymous function) runs, it receives a single element which is the Array Element type.

* Perform an arbitrary action for every element. In this case the closure returns nothing, so the useful work comes from an external effect, like doing output.
* intArray.forEach {
* (element: Int) -> Void in
* print("Found element: \(element)")
* }
* Found element: 1
* Found element: 2
* Found element: 3
* Found element: 4
* Found element: 5
* Transform it into another array. In this case we return the new array element type, which can be the same or different.
* let roots: Array<Double> = intArray.map {
* (element: Int) -> Double in
* return sqrt(Double(element) \* 2)
* }
* print("Transformed array: \(roots)")
* Transformed array: [1.4142135623730951, 2.0, 2.449489742783178, 2.8284271247461903, 3.1622776601683795]
* Summarize the array into a single value. This gets a bit more complicated. We need a starting value, and a running value that continually gets updated by incorporating a conrtibution from each element. Often the output type is the same but it need not be:
* let sumOfElements: Int = intArray.reduce(0, {
* (runningSum: Int, element: Int) in
* let updatedSum: Int = runningSum + element
* return updatedSum
* })
* print("The sum of \(intArray) is \(sumOfElements).")
* The sum of [1, 2, 3, 4, 5] is 15.
* As a reminder, a lot of syntax can be shortened (called “syntactic sugar”). It is critical to understand these are nothing but textual shortcuts because of redundant information in the original, similar to online-chat style “CUl8R” instead of “See you later”.
* let sumOfElements = intArray.reduce(0) {
* runningSum, element in
* runningSum + element

}

or even:

let sumOfElements = intArray.reduce(0) { $0 + $1 }

There are other built-in types that group objects together. The most important is Dictionary which has the same superficial syntax for retrieving an element: dict[i]but i is a *key* and the result is a *value* (of Optional type, because there may be nothing associated with that key). We want to be able to perform similar iterative operations on Dictionaries but there is a major complication. Dictionaries contain these *pairs* of objects, in which the type and value each is chosen by the programmer. By contrast, arrays are always indexed by Int.

Let’s assume we have no access to the documentation or it’s simply not comprehensible. How can we experiment to get iteration to work? First, some reasoning.

If iterating through a dictionary worked exactly the same way, we would get back the value but not the key, losing critical information. Let’s try anyway, and hope for a useful error message.

let tryDict: Dictionary<String, Int> = [ "carbon": 12, "nitrogen": 14, "oxygen": 16 ]

[0] = {

key = "nitrogen"

value = 14

}

[1] = {

key = "carbon"

value = 12

}

[2] = {

key = "oxygen"

value = 16

}

The output gives some clue as to internal representation. We can see the three elements. Important observations:

* The internal representation of Dictionary is an Array after all.
* Each element is a pair, where the pieces are named exactly key and value.
* The elements are *not* in the original order. Since lookups are only by key, Swift can store the elements in whatever order is it chooses (in this case, for speed of lookup).

What type is the pair? If it were a struct we ought to see a type. Perhaps it is a tuple. Let’s build a tuple manually and compare output.

let testTuple: (key: String, value: Int) = (key: "carbon", value: 12)

testTuple: (key: String, value: Int) = {

key = "carbon"

value = 12

}

That’s looking awfully similar (identical, actually) to the Dictionary debug output. So win #1 for experimentation: A Dictionary is really an Array of **tuples** where the first element is the key type and labeled key, and the same goes for the value!

Still, peeking inside does not guarantee the external interface. So, let us try an unmodified Array version with the dictionary type, and just hope for a useful error message:

tryDict.forEach { // Won't work, but let's just see what the compiler says

(element: Int) -> Void in

print("Found element: \(element)")

}

error: repl.swift:40:3: error: 'Int' is not convertible to '(key: String, value: Int)'

(element: Int) -> Void in

^

Alright, so our declared type was wrong. The closure does not want to receive just the value type. But look at that delicious clue. It tried and failed to convert our declared type to the *expected* type. Does it look familiar? Certainly does. Look at how it is identical to testTuple. Time to make a daring leap of generalization: *Whereever Array closures take the****value****as a parameter, Dictionaries take a****(key, value)****tuple.*

Based on these new results from our code laboratory, let’s revise the closure. (Note we’re deliberately experimenting with the simplest iterator, which doesn’t need to return anything!) The type declaration gets a little gnarly, because the type of a tuple involves a parenthesis pair, but this is really no big deal.

tryDict.forEach {

(elementPair: (key: String, value: Int)) -> Void in

print("Found element: \(elementPair)")

}

Found element: (key: "nitrogen", value: 14)

Found element: (key: "carbon", value: 12)

Found element: (key: "oxygen", value: 16)

Looks great! We pause to appreciate how nice it is that tuples print themselves so nicely, like all other built-in Swift types, using \() String interpolation.

Tuples are great, but don’t admit computation directly. We need to pull the pieces apart. How do we do that? Well, pieces have unique labels, so although they’re not strictly properties, let’s try that syntax anyway:

testTuple.key

$R10: String = "carbon"

testTuple.value

$R11: Int = 12

Perfect. Let’s continue in small steps, making use of these separated pieces in our developing code.

tryDict.forEach {

(elementPair: (key: String, value: Int)) -> Void in

print("Found element name: \(elementPair.key) with weight: \(elementPair.value)")

}

Found element name: nitrogen with weight: 14

Found element name: carbon with weight: 12

Found element name: oxygen with weight: 16

The syntax is starting to feel stifling, and we’ve got enough repetition to visualize what is left out.

tryDict.forEach { pair in

print("Found element name: \(pair.key) with weight: \(pair.value)")

}

Have data, will travel. Solving the input representation problem was the only real issue. Time to test our generalization that all the other closure-type functions work the same way. But map is a little unclear. Does it produce another Dictionary, or an Array? And of what type? Let’s remain agnostic and just write out an expression. We’ll know the return type of the closure (just type of value, which is Int), but let Swift infer the return type of the entire expression. The REPL will inform us by the debug output. How nice is that?! Just for flair, we’ll do a one-liner, recalling that trivial closures may even leave out the return keyword.

tryDict.map { pair in pair.value }

[Int] = 3 values {

[0] = 14

[1] = 12

[2] = 16

}

Nice. It’s an array. Just to be extra careful, we capture all the pieces separately with explicit declarations.

let weightsOnly: [Int] = tryDict.map { pair in pair.value }

weightsOnly: [Int] = 3 values {

[0] = 14

[1] = 12

[2] = 16

}

let elementNamesOnly: [String] = tryDict.map { pair in pair.key }

elementNamesOnly: [String] = 3 values {

[0] = "nitrogen"

[1] = "carbon"

[2] = "oxygen"

}

It’s all fitting together. The reduce method should follow easily, as long as we know which part of the tuple we want to reduce!

Do not worry about Equatable. I think what you’re missing is that these functions all come in two flavors. They are overloaded. Swift loves overloading.

* Take a sample (template) object as a parameter, and search for a match using ==
* Take a closure (a predicate style closure, that is, one that strictly returns true or false), and rely on the predicate to decide what qualifies as a match

Beware of the first variation. As you’ve seen this only works for types that are already Equatable. In practice, this means you should use this for built-in types only. Full struct equality is expensive (potentially lots of CPU time) for structures and is not what is asked for here.

Let’s look again at the spec. Always ground your thinking in the spec else you go astray. We want to match by item name. This means we need to pull apart the structure and explain to whatever search function that all it should care about (match on, test against) is the name field.

Here’s when you would use the simple, non-closure version:

let someNames = [ "David", "Helen", "Tony", "Carolyn", "John", "Justin", "Ben", "Daniel" ]

if someNames.contains("Carolyn") {

print("Found Carolyn. I don't know where, but I know she is somewhere in the array")

}

else {

print("Carolyn not found")

}

Now let’s suppose we have:

struct Student {

let name: String

let course: String

}

and:

let enrollments = [ Student(name: "Carolyn", course: "E65g"), Student(name: "Ben", course: "F76h"), Student(name: "Carolyn", course: "G87i") ]

We want to again search for "Carolyn". Now, the simple contains doesn’t work; if we ask for "Carolyn", Swift just tells us the Array Element type Student does not match our search object type String:

if enrollments.contains("Carolyn") { ... } // compile error!

Suppose we did manage to get Equatable to work. (Hint: struct Student: Equatable { ... }) That’s placating the compiler but missing the point. it’s not as if we can create a useful Student object to search for; assume we have no idea what course anyone is enrolled in, and we don’t want to restrict ourselves to a particular course anyway. Imagine:

if enrollments.contains(Student(name: "Carolyn", course: "Um, no idea")) { ... } // Will never find what we need

So we want to run a specific piece of narrow matching logic rather than just simple comparison of a match object against all array objects.

We have to use use the closure version. Again, the goal is to find if there are any enrollment records where the student’s name is Carolyn regardless of the value of the course field.

if ( enrollments.contains { (student: Student) -> Bool in

// This closure, a predicate, should return true if, and only if, the current student being examined fits our search criteria.

???

} ) {

print("Found at least one enrollment with a student named Carolyn.")

}

Can you write the body of the closure? It’s a one-liner. After you’ve solved this: The result of contains is just true/false; not great because you want the actual Student record, let’s say. So what related search function is better?

Also, here’s what I would expect you to do, assuming you can’t answer easily. Rather than conceptually sorting it all out in your mind, let Swift reveal things to you. I would expect you to try something like this. What’s the output, and does it all make sense now?

import Foundation

let result = enrollments.contains { (student: Student) -> Bool in

print("No idea how this works, but closure called with \(student)")

return false // Just to make compiler happy while I experiment

}

print("Result of experiment: \(result)")

And finally, change the hard-coded false to true. Why is the output so different?