ENGINEERING

Swift and Kotlin: The Subtle Differences

Kyle Ohanian
Software Engineer

June 17, 2019

Swift and Kotlin are the development languages for iOS and Android respectively. Both have certainly given the mobile development world a more functional feel. Their predecessors, Objective-C and Java, are very object-oriented, but Swift and Kotlin give us mobile developers an Object-oriented paradigm combined with some functional features. Swift and Kotlin were both designed to interop with Objective-C and Java, which allows for newer updates in apps to potentially be written in these languages. Both languages are foundational to the current mobile development world.

great deal about the design of a compiler, I feel like the biggest reward long term was having to force myself to do functional programming. I didn't have lots of experience with a functional programming language, but I did have lots of Java, C++, and C# experience. This made it easier for me when learning Swift and Kotlin because I had experience with both Object-oriented and functional programming. Being at WillowTree has allowed me to sink my teeth into both languages.

We have to remember that we aren't comparing iOS and Android, or XCode and Android Studio: we are comparing the languages that are used to build apps on these platforms. With that being said, I have enjoyed writing code in both of these languages. This is a side-by-side comparison of some important concepts in programming within these two languages.

Simple Syntactical Differences:

Immutable variable	let a = 3	val a = 3
Mutable variables	var a = 3	var a = 3
Self object referencing	self.a = a	this.a = a
arrays	let arr = ["hello", "world"]	val arr = arrayOf("hello", "world")
dictionaries/maps	let dict = ["hello": "world"]	val dict = hashMapOf("hello" to "world")
Type inference	let a: Float = 0.0	val a: Float = 0.0
Function declaration	func sampleMethod() -> Int	fun sampleMethod(): Int
Function default parameters	func sampleMethod(a: Int = 0) -> Int	fun sampleMethod(a: Int = 0): Int
If statements	if a == b {}	if (a == b) {}
For loops	for movie in movies	for (movie in movies)
Default values	let a = b?.value ?? 0	val a = b?.value ?: 0
Safe casting	let b = a as? Int	val b = a as? Int
Safe call	b?.print()	b?.print()
Force unwrap	b!.print()	b!!.print()
Lazy initialization	lazy var array2: [Int] = []	val array2 by lazy { intArrayOf() }

Kotlin Data Class

Kotlin has something called a data class, which is a class that when compiled, will get methods such as hashCode(), toString(), and copy() etc. It can be defined like this:

Tuples

Kotlin has a Pair and a Triple utility class for tuples of 2 and 3 components and it has the data class that can be used to emulate a custom tuple. Swift has tuple capabilities as well and can leverage the typealias for a tuple:

Swift:

```
typealias SampleTuple = (var1: String, var2: String, var3: Stri
```

Kotlin:

```
data class SampleTuple(val var1: String, val var2: String, val
```

Swift Structs:

The difference between Swift Structs and Swift Classes is that Structs are actually value types, while Classes are reference types. Value types are copied when they are assigned to a new variable or when passed as a parameter. Think of a Tuple as an implicitly defined Struct:

```
        var c: String = ""
}
```

Kotlin doesn't have a Struct type. However, we can instead use a copy function to create a new reference:

```
let sample = SampleClass(1, 1, "Sample") let newSample = sample
```

Even though one is a reference type and the other is a value type, Kotlin Data Class and Swift Structs are used to hold data. Based on my experience, I would consider them to be equivalent to each other.

Enums/Sealed Classes and Switch/When

Both Swift and Kotlin have enums. Swift's enums are algebraic data types, which are data types defined by two constructions, products, and sums. Kotlin's definition of an enum uses that of Java, which is a predetermined set of constants for a type. Swift can use associated values. In these examples, you'll also notice the differences between mapping values to defined cases (switch statements vs. when statements):

Swift:

```
        case basketball(Int, Time)
}

let game = Game.baseball(4, "2-2")
switch game {
case .regular:
case .baseball(let inning, let count):
case .football(let down, let yardsToGo, let timeLeft):
case .basketball(let quarter, let timeLeft):
}
```

In order to take advantage of algebraic data types, Kotlin has Sealed Classes. The only difference between Kotlin's Sealed Class and Swift's Enums is that sealed class is a reference type, while Swift's Enums are value types:

```
sealed class Game {
      object RegularGame: Game()
        class BaseballGame(let inning, let count): Game()
        class FootballGame(let down, let yardsToGo, let timeLef
        class BasketballGame(let quarter, let timeLeft): Game
}

let game = BaseballGame(4, "2-2")

when(game) {
     is Game.RegularGame -> // do stuff
```

```
}
```

Optionals:

Swift and Kotlin both allow for optional types. While you can still make mistakes, Swift and Kotlin are considered safer languages than others. This makes null checking much more trivial and allows you to have a clear structure and well-defined parameters on what can null/nil and what can't. Here are some examples of what can be done with optionals.

Swift:

```
struct SampleStruct {
    var id: Int?
    var thing1: String
    var thing2: String
    var thing3: String?
}

func printOut(sampleStruct: SampleStruct) {
    guard let id = sampleStruct.id else {
        print("Invalid id, cannot print things")
        return
    }
    print(sampleStruct.thing1)
```

```
    }
}
```

Kotlin:

```
data class SampleClass(var id: Int?, var thing1: String, var th

fun printOut(sampleClass: SampleClass) {
   val id = sampleClass.id ?: run {
       print("Invalid id, cannot print things")
       return
   }
   print(id)
   print(sampleClass.thing1)
   print(sampleClass.thing2)
   val thing3 = sampleClass.thing3
   if (thing3 != null) {
       // smart cast on a null check
       print(thing3)
   }
   // OR
   sampleClass.thing3?.let {
       print(it)
   }
}
```

its pointer. Kotlin uses the Java Virtual Machine's way of having null values. For Kotlin nullables, there is still use of the null pointer, but with its type system, you can determine if a variable should be allowed to contain a null pointer. The goal of Kotlin nullables is to help with the issue of the NullPointerException and add more tools for type checking.

Implicit Unwrapping vs lateinit

Swift also has implicit unwrapping, which defines an optional, but tells the compiler that it can unwrap it because it will have a value when ready. Kotlin can also do the same thing through the lateinit keyword. As developers, we try to avoid these as much as possible because of the dangers associated with using them, but they are useful for dependency injection (setting values in a type after initialization):

Swift:

```
var string: String!
func doBadStuff() {
   string.doSomething() // Will crash because string doesn't ha
}
---------------------
string = "String"
func doGoodStuff() {
   string.doSomething() // This is good because string was set
}
```

```
fun doBadStuff() {
   string.doSomething() // Will crash because string doesn't ha
}
---------------
string = "String"
fun doGoodStuff() {
    string.doSomething() // This is good because string was set
}
```

Extensions

Both Swift and Kotlin have extension capabilities. Extensions add functionality to an already existing type. You can extend say the Int type, for example, but the syntax for both is totally different.

Swift:

```
extension Int { var doubled: Int { return self * 2 } } let a =
```

Kotlin:

```
var Int.doubled: Int get() = this * 2 val a = 1.doubled
```

a set of functions/variables that a class must conform to if it implements the interface. A couple of differences between Swift and Kotlin is that Kotlin allows you to provide default functionality for a method, while Swift requires an extension of the protocol in order to give an implementation. Another is for generics. Kotlin allows for generics to go in the definition of the interface and Swift allows you to define an associatedType to go along with conformance.

Swift:

```
protocol SampleProtocol {
        associatedType T
        func get(value: T) -> T
}

extension SampleProtocol where Self.T == Self  {
        func get(value: Self) -> Self {
                // do stuff
        }
}

struct SomeClass: SampleProtocol {
     func get(value: SomeClass) -> SomeClass {
           // do stuff
     }
}
```

```
    fun get(value: T): T {
        // do stuff
    }
}

class SomeClass: SampleInterface {
    override fun get(value: Int): Int {
        // do stuff
    }
}
```

Higher Order Functions

Higher Order Functions are essentially functions that accept closures or functions as parameters and return either a new value or another function. With higher order functions, we are able to transform, filter, and iterate through values such as arrays.

Swift:

```
func higherOrderFunctions() {
    let array = [1,2,3,4,5,6,7,8,9,10]
    // $0 is implicitly defined as the current value of the spc
    let mappedArray = array.map {
        $0 * 2
```

```
    }
    // $0 is not an option because the current value in the clo
    let reducedArray = array.reduce(0) { (acc, element) in
        return acc + element
    }
}
```

Kotlin:

```
fun higherOrderFunctions() {
   val array = arrayOf(1,2,3,4,5,6,7,8,9,10)


   // it is implicitly defined as the value of the spot you are
   val mappedArray = array.map {
       it * 2
   }
   // result: [2,4,6,8,10,12,14,16,18,20]

   val filteredArray = array.filter {
       it % 2 == 0
   }
   // result: [2,4,6,8,10]
   // it is not an option because the current value in the clos
   val reducedArray = array.reduce { acc, element ->
```

```
}
```

With regard to mobile...

If we look at just the languages themselves, it's obvious that they are very similar when it comes to syntax. Even if one of the languages may lack a feature of another, we can still accomplish the goal by doing something a little different. On their respective paths, Swift and Kotlin are only going to get better.

2019 has seen some huge advancements that have greatly changed the mobile development landscape. Starting at Google I/O, Jetpack Compose was introduced as a new declarative UI framework to follow a reactive programming paradigm for Android. Then at WWDC, Apple introduced SwiftUI, which is another declarative UI framework for iOS. Both were created to simplify UI development. With these new declarative UI frameworks, and the similarities we see between Swift and Kotlin, I predict that the learning curve for developers who are interested in learning about the other platform will decrease, and that we may see more developers eager to learn both platforms.

Kyle Ohanian
Software Engineer