Go vs. Swift, The Languages of The Modern Tech Giants

Jake Rockland
jakerockland@gmail.com

December 5, 2016

Abstract

This project stands as a comparative exploration of Go and Swift, the recent flagship languages developed by Google and Apple, respectively. Specifically, I would like to explore first what pushed these modern tech behemoths to develop two new programming languages and second how these two languages are similar and how they differ, given that they were released at relatively similar times.

1 Introduction

The current technology landscape is often driven by two major companies: Apple and Google. With market capitalizations of $520.93 billion[1] and $586.02 billion[2], respectively, Apple ($AAPL) and Google ($GOOG) are the two highest valued companies in the world at the time of writing. Along with making top-class products that millions of consumers use on a regular basis, both Apple and Google have continually contributed to an ecosystem of tools built for developers to create new software, whether for their platforms or not. In the past decade, one way this has manifested is in the development of two new programming languages, Swift and Go.

The similar statuses of Apple and Google as technology behemoths of the modern world, as well as the similar timing of the releases of Swift and Go, drives us to explore first what pushed these companies to develop two new programming languages and second how these two languages are similar and how they differ. We will explore this by first looking at the historical context and initial goals of these languages, then look to the outside influences of each language, followed by a comparison of the design decisions made for each, finally we will conclude with a consideration of the future prospects for each language.

2 Historical Context

Development for Swift began in early 2010 by Chris Lattner, the main author of LLVM and the Clang compiler[3], who joined Apple in 2005 to integrate these tools into Apple's products at a production quality level[4]. Lattner was later joined by other developers at Apple in late 2011 and by 2013 Swift was a main focus of the Apple Developer Tools team. By the summer of 2014, Swift was ready for public release and was first introduced at WWDC on June 2nd[5]. The next year, on December 3, 2015, Apple announced that they would open Swift up as an open source project, welcoming collaboration from the public[6].

[1] `http://finance.yahoo.com/quote/aapl`
[2] `http://finance.yahoo.com/quote/goog`
[3] `http://llvm.org`
[4] `http://nondot.org/sabre/`
[5] `https://developer.apple.com/videos/wwdc2014/`
[6] `www.apple.com/pr/library/2015/12/03Apple-Releases-Swift-as-Open-Source.html`

The origins of Go date back slightly earlier, to 2007, when Robert Griesemer, Rob Pike and Ken Thompson, all working at Google, started outlining the goals for a new language[7]. By January 2008, Thompson, who had previously designed much of the initial UNIX operating system and the B programming language, had started working on a compiler to implement some of these new goals, which generated C code as its output. By the middle of that year, Go had become a full-time project at Google and late the following year, on November 10, 2009, Go was publicly announced as an open source project[8]. The language is now used in many of Google's production level systems as well as in many open source tools[9].

Looking at both of these stories, it is interesting to note that Swift was initially a private project and it wasn't over until a year after its initial release that it was made open source, whereas Go was released as open source from the get-go. Go was open-sourced under a BSD- style license[10], whereas Swift was open source under an Apache 2.0 license with a Runtime Library Exception[11]. Another interesting thing to note is the influence of notable past Bell Labs employees Thompson and Pike on the development of Go, which may give some explanation as to why Go appears to be more C-influenced than Swift.

3 Initial Goals

Before Swift, software for iOS and OS X was written in Objective-C. Objective-C was powerful in that it allowed the high-performance benefits of a C-based language; however, it was also cumbersome to write software in, as it was essentially an extension of C to allow object oriented support built on top of basic C, which made for many points of confusion and error in programming[12]. Swift was born out of a desire to make it easier to write software for iOS, macOS, watchOS, and tvOS. It takes inspiration from many other languages and was written to be easy to write software in, like a scripting language, but still take full advantage of the hardware it was being run on by being a compiled language like C++. With Swift, interaction with Objective-C code and libraries is still supported, while allowing for safer and more modern development of new software.

Alternatively, Go was born out of Google's need for a better systems programming language for scalable development[13]. Previous to the creation of Go, Google had used a software stack of C++, Python, and Java applications. Programmers ofter were left to make a decision between efficient compilation, efficient execution, or ease of programming, due to restrictions of these mainstream languages. The desire for a simpler way to write software for highly

[7] https://golang.org/doc/faq#history
[8] https://techcrunch.com/2009/11/10/google-go-language/
[9] https://golang.org/doc/faq#Is_Google_using_go_internally
[10] https://golang.org/LICENSE
[11] https://swift.org/LICENSE.txt
[12] https://www.quora.com/What-are-the-reasons-that-Swift-was-created-given-that-Objective-C-was-used
[13] https://golang.org/doc/faq#creating_a_new_language

scalable network servers and distributed systems lead to the creation of Go, a language designed to combine the ease of programming found in interpreted dynamically typed languages with the efficiency of compiled statically typed languages, including built- in support for things like multi-core processing and simple dependency analysis[14].

4 Outside Influence

The design of Swift was influenced by a variety of existing programming languages, primarily C#, CLU, D, Haskell, Objective-C, Python, Rust, and Ruby[15]. Go takes much influence from the C-based languages with respect to its basic syntax while incorporating aspects from the Pascal/Modula/Oberon family as well as ideas from Newsqueak and Limbo [16]. Because of their shared roots in C and other C-based languages, Swift and Go have many similarities in their syntax and the expressiveness that it allows.

5 Language Design

To introduce the language design of these two languages, let us consider, by virtue of tradition, a basic "Hello, world!" application written in each.

In Swift, such an application can be written as:

```
print("Hello, world!")
```

This line alone is a complete program in Swift. No standard libraries need to be imported for string handling and I/O, as such libraries are included by default. Additionally, code written at a global scope is used as the entry point for the program, removing the need for a main() function [16]. To compile and run the above hello world example:

```
swiftc hello.swift -o hello
./hello
```

In Go, we can write such an application as:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, world!")
}
```

Unlike Swift, Go does require that we import the standard "fmt" library and declare this file as a package, main, to define it as our entry point for the program. To compile and run this example:

[14] https://talks.golang.org/2012/splash.article#TOC_4
[15] http://nondot.org/sabre/
[16] https://golang.org/doc/faq#ancestors

```
go build hello.go
./hello
```

Both Go and Swift are compiled languages by design [17][18]. Go is designed explicitly for fast build times and accomplishes this with fast dependency analysis [19]. The Go compiler is part of the GNU tools, while Swift compiles to machine language using LLVM [20] [21].

Swift is strongly and statically typed, meaning that all variables must have a type that is defined or inferred at compile-time and that these types cannot change at run-time [22]. Type inference occurs when initializing a variable that has not been explicitly typed, though typing can also be explicitly declared, consider the following example:

```
/* example of type inference */
var hello = "Hello,"

/* example of explicit typing */
var world: String
world = " world!"

/* valid declaration and assignment */
var hello_world = hello + world

/* later, doing this throws a compile-time error */
hello_world = 125
```

Go is also a statically typed language, with typing working similar to how it does in Swift. Like with Swift, variables in Go can have their types inferred or declared explicitly. Consider the previous example implemented in Go:

```
/* example of type inference */
var hello = "Hello,"

/* example of explicit typing */
var world: string = " world!"

/* valid declaration and assignment */
var hello_world = hello + world

/* like in Swift, this throws a compile-time error */
hello_world = 125
```

Swift has no true primitive types; rather there is only the distinction between named types and compound types [23]. The Int, UInt, Float, Double, String, Character and Boolean types, often primitive or basic types in other languages, are all named types in Swift and are all defined in the standard library as

structs [24]. Swift also supports the Optional type, which represents a variable that either has a value or has a nil value [25].

Go implements a collection of basic types, which include bool, string, int, uint, byte (alias for uint8), rune (alias for int32), float32, float64, complex64, and complex128 [26].

While implemented differently, Swift and Go provide relatively similar built-in types. Their main differences are Go's native support for complex numbers—which must be hacked together with tuples in Swift—and Swift's native support for optional types—which must be hacked together with variadic parameters in Go.

If-else statements in Swift and Go are almost identical [27] [28]:

```
// Swift
if name == "Barack Obama" {
    print("Hello Mr. President")
} else if name == "Joe Biden" {
    print("Hello Mr. Vice President")
} else {
    print("Hello \(name)")
}

// Go
if name == "Barack Obama" {
    fmt.Println("Hello Mr. President")
} else if name == "Joe Biden" {
    fmt.Println("Hello Mr. Vice President")
} else {
    fmt.Println("Hello", name)
}
```

Both Swift and Go support fairly expressive switch statements; however, Swift has the benefit of supporting optionals and Go has the benefit of representing switches without pattern matching, avoiding the requirement that each case is a constant—making Go switch statements more like extended if-else statements [27] [28].

Neither language supports implicit fall-throughs, both support multiple cases as comma separated lists, and both support interval matching/checking. In Swift, switch statements must always be exhaustive, while Go has no such requirement [27] [28].

Such differences between switch statements in Swift and Go can be examined in the following example:

```
// Swift
switch (firstName, lastName) {
case ("Barack", "Obama"):
    print("Hello Mr. President")
case (_, \Obama"), (_, "Biden"):
```

```
    print("Have you seen Mr. President?")
default:
    print("Hello \(name)")
}

// Go
switch {
case firstName == "Barack" && lastName == "Obama":
    fmt.Println("Hello Mr. President")
case lastName == "Obama" || lastName == "Biden":
    fmt.Println("Have you seen Mr. President?")
default:
    fmt.Println("Hello ", name)
}
```

Swift supports for-in, while, and repeat-while loops that take the following formats [27]:

```
for card in deck {
    print("\(card.value) of \(card.suit)")
}

while !deck.isEmpty {
    card = deck.pop()
    print("\(card.value) of \(card.suit)")
}

repeat {
    hand.push(draw.pop())
} while hand.count < 5
```

Go on the other hand does not have a do-while or repeat-while style loop. Instead, Go has only one for-loop that takes influence from C, but combines the functionality of the while and for loops, as is done in the following [28]:

```
for _, card := range deck {
    fmt.Println(card.value, " of ", card.suit)
}

for i := 0; i < deck.count; i++ {
    card := deck.pop()
    fmt.Println(card.value, " of ", card.suit)
}

for i := 0; i < 5; i++ {
    hand.push(deck.pop())
}
```

6 Object Oriented Programming Support

Throughout Apple's history, many of its frameworks have been built around the Object Oriented Programming paradigm. In order to support backward compatibility with Apple's main API, Cocoa, and run within Objective-C's runtime, Swift was built with thorough object oriented support and readily allows for encapsulation, polymorphism, and inheritance [29].

To illustrate, consider the following simple class hierarchy example in Swift:

```
/* base `Car` class */
class Car {
    var make: String
    var year: Int

    init(make: String, year: Int) {
        self.make = make
        self.year = year
    }

    func value() -> Double {
        switch year {
        case 2000..<2010: return 6500.0
        case 2010..<2020: return 9000.0
        default: return 3000.0
        }
    }
}

/* `Tesla` class, subclass of `Car` class */
class Tesla: Car {
    init(year: Int) {
        super.init(make: "Tesla":, year: year)
    }

    override func value() -> Double {
        switch year {
        case 2010..<2015: return 18000.0
        default: return 24000.0
        }
    }
}
```

While Swift offers good support for the object-oriented paradigm, many Swift developers favor a similar but decidedly different paradigm that Swift also readily supports called Protocol Oriented Programming. The protocol-oriented paradigm emphasizes generalization and interfaces over inheritance and sub-

classing, which many argue allows for clearer, lighter, and more modular code [30] [31].

Whether or not Go supports Object Oriented Programming is very much a matter of definition. The official project documentations points towards the unsatisfying answer of "Yes and no" [32]. Go supports the creation of something similar to an object via the struct, a custom data type. However, due to its lack of inheritance, attempting to emulate the above Swift example in Go is not straightforward [33] [34].

Thus, support for the object-oriented paradigm provides one key difference between the two languages—Swift is designed with native support for Object Oriented Programming, whereas Go's support is somewhat lacking.

7 Concurrent Programming Support

Go was designed to be a language that could be scaled across multiple processors—giving it a fundamental native support for concurrency. To make a function capable of running concurrently with other functions, simply use the language defined go keyword before invoking the function to run it as a goroutine [35] [36]. For example, consider the slightly modified version of our hello world program from before:

```
package main
import "fmt"

/* simple hello function */
func hello(name string) {
    fmt.Println("Hello, ", name)
}

/* concurrently will run for all names passed in as args */
func main() {
    var argsWithoutProg := os.Args[1:]

    for _, arg := range argsWithoutProg {
        go hello(arg)
    }
}
```

Go also supports communication and synchronization between concurrent goroutine functions using the chan keyword to create "channels" [35].

When it comes to supporting concurrent programming in Swift, things are much less straightforward. Rather than being built into the language, Swift supports concurrency through Apple's Grand Central Dispatch API, which allows developers to schedule different tasks with dispatch queues and specify whether the tasks are added to the serial queue or to a concurrent queue, and at which level of priority [37] [38]. Because concurrency is supported through an Apple

API rather than being explicit in the language's design, using it is currently very much coupled to iOS or macOS development [39].

Here arises another fundamental difference between Swift and Go—Swift only somewhat supports concurrent programming but in a way that is a bit cumbersome to implement and that relies on Apple API's, while in Go, concurrency is fundamentally built into the language.

8 Future Prospects

Both Swift and Go seem to be languages with promising futures. Looking at the public Github repositories for each, it is not obvious which is being more actively worked on by the open source community. Many more people have forked or starred the Swift repository and it has roughly 10,000 more commits, despite having been released as an open source project for less time [40]. However, Go also has more people who have contributed to the language, with almost twice the number of public contributors as Swift [41].

Looking at the popular open source projects being built with Go, it seems that Go's future will be defined by its power as a server side language for large and highly distributed systems [42]. Swift's future seems to be a bit less clearly defined. Undoubtedly it will continue to be the development language for all Apple related software, including iOS, macOS, watchOS, and tvOS. However, looking at the current projects being explored with Swift, it seems likely that Swift will also be defined by its versatility as a powerful modern compiled language, with many potential applications outside of the Apple ecosystem from server side applications to integrations in cross-platform developer toolkits [43].

9 Final Remarks

Though they were designed with different use-cases in mind, Go and Swift were both developed to provide modern and expressive languages that did not sacrifice readability and writability for performance. Despite the variety of discrepancies that we have explored, both languages are quite similar in their static and strongly enforced typing, the basic data types that they provide, the control structures that they support, and the fact that they are both compiled languages. The greatest divergence between the two is Swift's strength within the object-oriented design paradigm, which Go does not fully support, and Go's strength in concurrent programming, which can only be achieved in Swift by using Apple APIs.

Existing for less than a decade, the legacies of both languages are yet to be determined; however, what is clear is that these languages will both have a long-lasting impact on the world of software development due to the powerful and expressive code that they allow, made further evident by their widespread support among the open source community and their firm backing by the world's two biggest software companies.