Swift 4
”The powerful programming language that is also easy to learn”

Metehan Dogan
Technische Hochschule Mittelhessen
Fachbereich Mathematik, Naturwissenschaften und Informatik
Email: *metehan.dogan@mni.thm.de*

***Abstract*—Dieses Dokument bietet einen Einblick in die Programmiersprache ”Swift 4”. Zu Beginn wird Swift in einigen Worten erklärt und danach wird über die Entstehung der Sprache berichtet. Anschließend wird die Benutzung von Swift und die Syntax erläutert. Am Ende folgt ein Fazit mit Ausblick in die Zukunft für diese Programmiersprache.**

I. Einleitung

Fig. 1. Swift Logo

Swift ist eine mächtige und intuitive Programmiersprache für Apple Plattformen wie zum Beispiel *macOS, iOS, watchOS and tvOS*. Die Sprache findet eindeutigen Gebrauch bei der Entwicklung von nativen Anwendungen für die Apple Produkte. Durch die Einfachheit der Sprache finden viele moderne Entwickler und Einsteiger eine Möglichkeit effizienten Code zu schreiben. Dazu ist Swift eine stabile Sprache und wird nach ihren Entwicklern als *”Blitzschnelle”* bezeichnet.[1] Um die Programmiersprache Swift einfach und effektiv zu benutzen, sollte man die IDE **Xcode 9** von Apple herunterladen und verwenden.[2] Jedoch ist diese IDE nicht notwendig, um Programme in Swift schreiben zu können. Apple bietet den Swift Compiler als Open Source Ressource an. So ist es dem Entwickler möglich sein Swift Programm in einem einfachem Editor zu editieren und dann zu einem ausführbaren Code zu compilieren.

II. Geschichte

Bevor Swift entwickelt wurde, benutze man noch *Objectiv-C*, eine Programmiersprache aus dem Jahr 1980.[3] *Objective-C* konnte mit den heutigen Programmiersprachen wenig mithalten und besaß zudem einige Mängel. Aus diesem Grund haben viele Entwickler auf bessere Entwicklungsumstände gewartet und die Antwort war ***Swift***. Die damals noch neuartige Sprache wurde das erste mal auf der Entwickler-Konferenz **WWDC** von Apple 2014 präsentiert[4]. *(siehe Präsentationsbild)*

`"Swift ist für Apple ein Berfreiungsschlag"`

nach Michael Koffler [1] S.15.

[1]Vgl. [2] und [3]
[2]Vgl. [4] ***Xcode9 Download:*** https://developer.apple.com/xcode/
[3]Vgl. [1] S.15
[4]Vgl. [5] Videos auf der Webseite zu finden

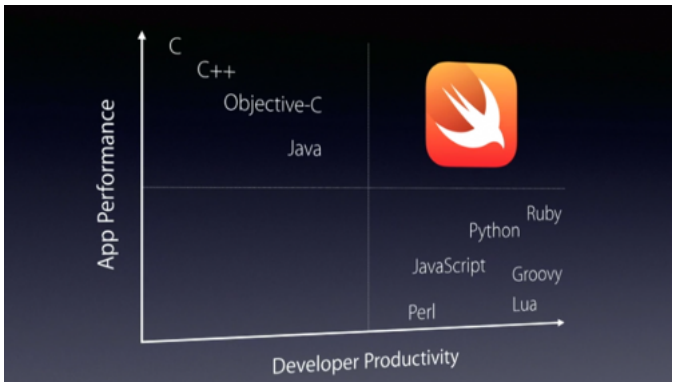

Fig. 2. WWDC 2014 Swift Präsentation

Swift ist leicht an *Objectiv-C* orientiert, damit auch alte Apple-Bibliotheken mit der neuen Sprache funktionieren können. Außerdem wurden viele Elemente von C#, Haskell, Java, Python und weiteren Programmiersprachen in Swift konzeptionell integriert.[5] Daraus ergeben sich folgende Vorteile nach Kofler:

1. Swift gehört mit zu den modernsten Sprachen
2. Syntax simpler als in *Objectiv-C*
3. Lesbarkeit ist deutlich erhöht, wegen der Einfachheit
4. Für moderne Programmierer verständlicher
5. Ist ein Open-Source-Produkt

Seitdem erscheint jedes Jahr ca. nach der WWDC eine neue Hauptversion. Somit liegt die Wahrscheinlichkeit sehr hoch, dass dieses Jahr spätestens Ende 2018 eine neue Version erscheinen wird, welche erneut weitere Funktionen besitzen wird. Darunter eine Optimierung beim Umgang mit Zeichenketten, Generics-Features und ein Grundgerüst für die asynchrone Programmierkonstrukte [6].

III. Die ersten Schritte

Bevor man mit dem Schreiben von Swift-Code beginnt, ist es *wichtig* zu wissen, auf welchem Betriebssystem man dies machen möchte. Die beste Unterstützung findet man auf **macOS-Systemen**, aufgrund der Tatsache, dass man

[5]Vgl. [1] S.15
[6]Vgl. [1] S.16

eine IDE besitzt, um einfacher *grafische Anwendungen* entwickeln zu können. Es besteht auch die Möglichkeit eine *Virtuelle Maschine* zu verwenden, jedoch gäbe es dann *Lizenz-Probleme* mit Apple. Eine *VM mit macOS* darf nur auf Apple Hardware verwendet werden.[7]. Letztendlich ist ein *Apple Produkt* nötig, welches mit macOS bestückt ist, um mit der IDE **Xcode** programmieren zu können. Leider ist es nicht möglich die IDE mit *iOS Produkten* zu benutzen. ***Alternative für nicht Apple Nutzer:***
Eine Installation des Compilers für Swift 4 mit einem Editor.

Fig. 3. Swift is open-source

1. Download und Anleitung für Linux-Distributionen
   https://swift.org/download/
2. Download und Anleitung für Windows
   https://swiftforwindows.github.io/
3. Web-Anwendung für Code schreiben und Compilieren
   https://iswift.org/playground

Zudem wird ein **Editor** benötigt wie zum Beispiel *Atom oder Sublime.*

Fig. 4. Die genannten Editoren Sublime und Atom

1. Atom-Editor
   https://atom.io/
2. Sublime-Editor
   https://www.sublimetext.com/

Sobald alles installiert ist. Testen wir unsere Umgebung mit einem *"Hallo World!"*. Wenn die Konsole unsere gewünschte erste Zeile ausgibt, sind wir bereit die ersten Codezeilen in Swift zu schreiben.

```
print("Hallo World!")
```

IV. Erläuterung der Syntax

A. Grundlagen

Zwar ist Swift 4 eine neuartige Sprache, aber für erfahrene C und *Objectiv-C* Entwickler schnell zu verstehen. Somit besitzt es auch die selben primitiven Typen wie die genannten älteren Programmiersprachen. Dennoch sind neue Elemente vorhanden w.z.B. Tupels, Typenumwandlung und die Möglichkeit den Typen spezifisch anzugeben, wenn nötig. Diese Funktion ähnelt den Funktionen von *Typscript* sehr.
In Swift unterscheidet man eine *Konstante* zwischen einer *Variable*[8]. Während eine *Konstante* einen unveränderbaren Wert beinhaltet, speichert eine *Variable* einen Wert, welchen man im weiteren Verlauf des Programms verändern kann. Wenn man eine *Konstante* verändert möchte, erscheint eine Fehlermeldung, welche den Benutzer darauf aufmerksam macht, dass es sich um eine *Konstante* handelt. Dazu hier ein Beispiel[9]:

```
// Eine Konstante
let pi = 3.14
// Eine veraenderbare Variable
var alter = 23
// Fehler
let gravitation = 9.81
gravitation = 10
// error: gravitation cannot be changed.
```

Um einen expliziten Typen zu bestimmen, fügt man hinter der Deklaration ein Doppelpunkt mit dem gewünschten Typen ein. Ähnlich wie bei einer *UML-Notation*.

```
// Typangabe 1
var anzahlReifen: Int = 4
// Typangabe 2
var name: String = "Klaus"
```

Die Benennung von Variablen und Konstanten ist frei gestaltbar. So kann man jeden Buchstaben oder auch *Unicode Zeichen* verwenden. Natürlich darf man die *print()* Funktion nicht vergessen.

```
// Ausgabe Hallo
print("Hallo")
// Ausgabe: Klaus
print(name)
// Ausgabe: Mein Name ist Klaus
print("Mein Name ist ",name)
// Weitere Methode fuer gleiche Ausgabe
print("Mein Name ist \(name)")
```

Swift braucht keine Semikolons, deswegen kann man sie weglassen oder auch schreiben. Kommentare werden wie bereits gezeigt mit // für einzeilige und /* Code-Block */ für mehrzeilige Kommentare verwendet.

B. Kontrollfluss

Swift beinhaltet eine durchschaubare Menge an Kontrollflüssen. Alle werden im einzeln erklärt. Zuerst betrachten wir uns die Bedingten Anweisungen. Hier ein Beispiel[10].

```
/* If-Anweisungen koennen auch ohne Klammern um
dein booleschen Wert verwendet werden */
if anzahlReifen > 0 {print("Ich habe Reifen")}
else if anzahlReifen == 1 {
print("Ich habe genau ein Reifen")
}
/* Geschweifte Klammern sind nach dem If
notwendig */
```

[7]Vgl. [6] *Apple Software Lizenzverträge*
[8]Vgl. [7] *Apple Swift Dokumentation*
[9]Vgl. [7] und [8]
[10]Vgl. [7] und [8]

```
else print("Wo sind meine Reifen?")
// error: expected '{' after 'else'
```

Auch der Ternärer ?:-Operator findet sich hier wieder. Auf dieser Art werden schnelle Vergleiche mit nur zwei möglichen Ergebnissen schnell unterschieden. Hier das Beispiel[11]:

```
/* Eine schnelle Unterscheidung von
zwei moegllichen Faellen */
x = tank < verbrauch ? "Tanken":"Passt"
```

Eine *switch* Anweisung vergleicht alle Möglichkeiten eines Wertes. Dabei unterscheidet er von den jeweils angegebenen Werten. Ist ein Wert nicht dabei wird ein Standardwert verwendet.

```
var wert = 1
switch wert {
case 0: print("0")
case 1: print("1")
default: print("Wert ist kein Bit")
}
// Ausgabe: 1
```

Die *for-Schleife* wurde in Swift simpel gehalten. Sie eignet sich gut um z.B. über Collections zu iterieren. Oder eine bestimmte Anzahl an Anweisungen durchzuführen.

```
// Fuer das iterieren von 1 bis 4
// Die Fuenf wird nicht mit gezaehlt
for i in 1..<5 {
print(i)
}
// Fuer das iterieren von 0,2,4,6,8
for i in stride(from:0, to:10, by: 2) {
print(i)
}
```

Die *While-Schleife* ist ähnlich wie die *for-Schleife*. Jedoch wird die *While-Schleife* solange durchlaufen bis die Bedingung falsch wird. Die *repeat-while-Schleife* ist unter den anderen Sprachen als *do-while-Schleife* bekannt. Ähnlich ist ihre Funktion. Der Anweisungsblock wird so garantiert einmal durchlaufen.

```
var i = 0
while i < 5 {
print(i)
i+=1
}
// Ausgabe: 0,1,2,3,4 & var = 5
repeat {
print(i);i+=1
} while i < 5
// Ausgabe: 5 weil einmal durchlaufen
```

[11]Vgl. [7] und [8]

C. Optionale Typen

Optionale Typen verwendet man, wenn es nicht sicher ist, dass der Wert tatsächlich eine Zuweisung des angegebenen Typs bekommt oder überhaupt einen Wert besitzt. Sobald kein gültiger Wert vorhanden ist, wird ein *"nil"* dem Wert zugewiesen. Hier dazu ein Beispiel[12]:

```
/* Die Initialisierung einer Variable die
eine Zahl sein kann, sofern das Parsen
funktioniert */
var geparsteZahl: Int? = ("2")
/* Wenn wir uns nun sicher sind, dass dieser Wert
nicht nil sein wird, koennen wir die Variable zu
einem 'Entpacken' zwingen*/
var nichtNil: Int = geparsteZahl!
/* WICHTIG, wenn wir das Unwrappen(eng. fuer
Entpacken) erzwingen und ein nil gelesen wird
fuer dies zu einem Error */
```

Ein sicheres Unwrappen ist durch die *If-let* oder *If-var* Funktion garantiert. Diese fängt das *nil* auf und springt dann in den *else-Rumpf*, wenn einer existiert.

```
if let binIchInt = geparsteZahl {
print("Mein Wert: ", binIchInt )
} else {
print("Ich bin doch Nil")
}
```

D. Zeichenketten

Zeichenketten werden in der Umgangssprache auch als String bezeichnet. Sie dienen der Textdarstellung. Sie bestehen aus Characters, welche die einzelnen Buchstaben sind. So ist es auch möglich Unicode Zeichen darzustellen. In diesem Fall sprechen wir von modernen Smilies und Symbolen.

```
// Das \u{2665} ist der Unicode fuer ein Herz
var str = "Ich \u{2665} Swift"
// Eine Deklarierung eines leeren Strings
let leererString = String()
// Wiederholungen sind auch moeglich
more = String(repeating: "*", count: 5)
// Die Laenge eines Stringes ueberpruefen
let wort = "Hallo"
let wort2 = String()
print(wort.count)
print(wort.isEmpty)
// Ausgabe1: 5 Ausgabe2: false
// String verlaengern
wort += " Welt"
// Stringbereiche
let wort = "Hallo"
let start = wort.startIndex
let ende = wort.index(start, offsetBy: 2)
let bereich = start...ende
let teilWort = wort[bereich]
```

[12]Vgl. [7] und [8]

```
print(teilWort)
// Ausgabe: Hal
```

E. Listen

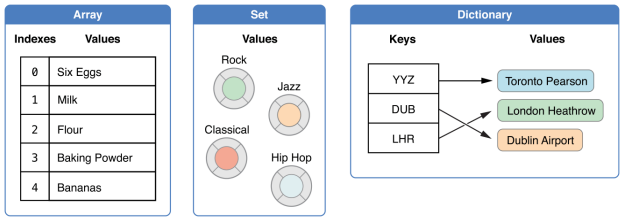

Fig. 5. CollectionTypes

Listen (auch Arrays genannt) sowie auch Dictionaries und Sets sind Typ klar. Dies bedeutet, dass eine Angabe des Typs erforderlich ist, um mit diesen Datenkonstrukten arbeiten zu können. Der Umgang mit Arrays erfolgt intuitiv. Es folgen nun einige Beispiele[13].

```
// Initialisierung eines Arrays
var zahlen: [Int] = [0,1,2,3,4]
let leeresArray1: [Int] = []
let leeresArray2 = [Int]()
// Die Zahl 13 3 mal wiederholen
let reps = [Int](repeating: 13, count: 3)
// Elementenzugriff
let anzahl = zahlen.count
let istLeer = zahlen.isEmpty
let erstesElement = zahlen.first
let zweitesElement = zahlen[1]
let letztesElement = zahlen.last
// Im Array suchen
let woIst = zahlen.index(of: 3)
// Teilbereich im Array
zahlen[0...2] == zahlen[..>3]
// Beide Mengen entsprechen (0,1,2)
// Iteration durch Arrays
var zahlen: [Int] = [0,1,2,3,4]
for zahl in zahlen {
print(zahl)
}
zahlen.forEach { (zahl) in
print(zahl)
}
// Eine Iteration durch enums ist moeglich
for (index, zahl) in zahlen.enumerated() {
print("#", index + 1, ": ", zahl)
}
// Arraymanipulation
zahlen[0] = 0
zahlen.append(5)
zahlen += [ 6, 7 ]
let erster = zahlen.remove(at: 0)
numbers.insert(1, at: 0)
let letzter = zahlen.removeLast()
/* Das Entfernen von Element gibt das
Element zurueck, aehnlich wie bei
einer pop() Funktion
```

F. Blockbasierte Listenoperationen

Mit diesen Funktionen kann man Arrays bearbeiten. Dabei handelt es sich um Operationen wie *map(),filter(), sort() und reduce()*. Auf jedes folgt ein Beispiel[14]:

```
// map und flatmap
let freunde = ["Max", "Tom", "Jana"]
let kleiner = freunde.map {$0.lowercased()}
// => ["max", "tom", "jana"]
var zahlen = [1, 2, 3, 4]
let mapped = zahlen.map {
Array(count: $0, repeatedValue: $0) }
// => [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]
let flatMapped = zahlen.flatMap {
Array(count: $0, repeatedValue: $0) }
// => [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
// Filtern von geraden Zahlen
zahlen.filter { $0 % 2 == 0 }
// Sortieren.
zahlen.sort()
zahlen.sort { $0 > $1 }
zahlen.sort(by: >)
// => 1. aufsteigende Sortierung
// => 2. und 3. absteigende Sortierung
// Summieren
let erg = zahlen.reduce(0) {
(sum, e) in sum + e }
print(erg)
// => 13
```

G. Dictionaries und Sets

Ein Dictionary speichert die Verknüpfungen zwischen einem Schlüssel *key* und einem Wert. Beide gehören zusammen w.z.B. ein Produkt mit seinem Preis. So kann man Abhängigkeiten erzeugen.

```
// Assoziative Dictionaries (Schluessel:Wert)
var produkte = ["Apfel": 4, "Kiwi": 10]
let leeresDict : [String:Int] = [:]
let anzahl = produkte["Apple"]
// Neuen Tupel einfuegen
produkte["Orange"] = 20
for (key, value) in produkte {
print(key, " => ", value)
} /* => Apfel  =>  4
Kiwi  =>  10
Orange  =>  20 */
```

[13]Vgl. [7] und [8]

[14]Vgl. [7] und [8]

Währenddessen sind Sets eine Menge an eindeutigen Werten. Sie werden ohne eine bestimmte Reihenfolge gespeichert. Ein solches Konstrukt eignet sich um z.B. Musik-Genre zu implementieren.

```
let zahlenSet : Set<Int> = Set([2, 4, 8])
// Von dem Set entfernen
zahlenSet.subtracting(Set([4, 16]))//=>[2, 8]
// Schnittmenge des Sets
zahlenSet.intersection(Set([4, 16]))//=>[4]
// Set erweitern
zahlenSet.union(Set([4, 16]))//=>[2, 4, 8, 16]
```

H. Objektorientierte Programmierung

Objektorientierte Programmierung beschäftigt sich mit der Erzeugung flexibler Konstrukte, welche eine größere Menge an Datensätzen beinhalten können und sogar Funktionen implementiert haben. Sie vereinfachen die Arbeit eines Softwareentwickler sehr. Aus größeren Anwendungen sind sie nicht wegzudenken. Swift benötigt keine *Interfaces*. Ein externes *Interface* wird automatisch angelegt. Somit muss man nur zwischen einer Klasse und einem Konstrukt entscheiden. Zuerst betrachten wir uns die Konstrukte/*Structures*. Sie dienen der Zusammenfassung von mehreren Variablen. Deswegen ist Vererbung nicht möglich, dafür werden die Werte immer kopiert. So entstehen keine Referenzprobleme[15].

1) Structures: =>

```
struct Mensch {
let name: String
var alter: Int
}
var lukas = Mensch(
name: "Lukas",
alter: 14 )
print("Das ist ", lukas.name)
// Ausgabe: Das ist  Lukas
```

2) Verwendung von Initializern, Methoden und Eigenschaften: =>

```
msg = String(repeating: "+", count: 5)
let s = msg.startIndex
msg.insert(contentsOf: msg.characters,at: s)
```

3) Klassendefinition, Eigenschaften, Initializer, berechnete Eigenschaften: =>

```
class Kreis {
// Variablen Deklaration
var radius : Double {
willSet { print(radius, "=>", newValue) }
didSet { print(oldValue, "=>", radius) }
}
var durchmesser : Double {
get { return radius * 2 }
set { self.radius = newValue * 0.5 }
}
var flaeche : Double {
return pi * pow(radius, 2)
}
// Konstruktor
init(radius : Double) {
self.radius = radius
}
// Funktion
func skalieren(factor : Double) -> Circle {
let newRadius = self.radius * factor
return Circle(radius: newRadius)
}
}
```

4) Lazy-initialisierte Eigenschaften: =>
Eine *Lazy Stored*-Eigenschaft ist eine Eigenschaft, deren Anfangswert erst bei der ersten Verwendung berechnet wird. Man initialisiert eine *Lazy*-gespeicherte Eigenschaft, indem man den *Lazy-Modifikator* vor der Deklaration schreiben.

```
class LazyKreis {
lazy var Kreis = Kreis(radius: 5)
lazy var Kreis2 : Kreis = {
return Kreis(radius: 5)
}()
}
```

5) Typumwandlung: =>
Für das Parsen von Werten in einen anderen Typen.

```
someValue as? String
// -> String?, nil wenn kein String
someValue as! String
// -> String, Error wenn kein String
```

*6) Protokolle: =>*Protokolle sind *umgangssprachlich* das *Interface* von Swift. Es beschreibt bestimmte Funktionalitäten, die von einer Klasse,Struktur oder einem Enum garantiert werden müssen, wenn das Protokoll implementiert wird[16].

```
protocol Konto {
var stand : Int { get set }
func abbuchen(x : Int)
}
/* Implementierung von dem Protokoll
Konto */
class KontoCheck : Konto {
var stand = 0
func abbuchen(x : Int) {
stand -= x}
}
```

7) Extensions: =>
Extensions fügen einer vorhandenen Klasse, Struktur, Enum oder einem neuen Protokolltyp neue Funktionen hinzu. Dies umfasst die Möglichkeit, Typen zu erweitern, für die man keinen Zugriff auf den Quellcode hat z.B. Int um eine neue

[15]Vgl. [7] und [8]

[16]Vgl. [7] und [8]

Berechnungsfunktion zu erweitern. Extensions ähneln den Kategorien in Objective-C[17].

```
extension Date {
func addDays(days : Int) -> Date? {
let cal = Calendar.current
return cal.date(byAdding: .day,
value: days, to: self)}
}
```

8) Enums: =>

Enums sind Aufzählungen von allen möglichen Werten, die dieser Datentyp haben darf. Bei der Benutzung von Enums in einer switch-Funktionen ist ein *default* nicht mehr nötig, weil der Wert nur bestimmte Werte annehmen kann.

```
enum HimmelsRichtung {
case Norden
case Sueden
// Kurzschreibweise
case Osten, Westen
}
var direction = Direction.North
direction = .South
```

9) Enums: Associated Values: =>

Zu den bereits genannten Eigenschaften von Enums, könnten diese auch einen zugehörigen Wert besitzen. Hier das Beispiel für diesen Fall:

```
enum Result {
case Success(String)
case Error(Error)
}
let result = Result.Success("Test")
switch(result) {
case .Success(let value):
print("Sucess, got \(value)")
case .Error(let error):
print("Error, got \(error)")
}
```

I. Funktionale Programmierung

Die Funktionale Programmierung beschäftigt sich mit der Formulierung von Funktionen in Swift.

```
// Swift-Typen fuer Funktionen
// func x(parameter: Int)-> Rueckgabetyp{}
func triple(value : Int) -> Int {
return value * 3
}
let operation : (Int) -> Int = triple
numbers.map(triple)
// Closures
numbers.map({ (value : Int) -> Int in
return value * 3
})
numbers.map({ (value) in value * 3 })
numbers.map({ $0 * 3 })
numbers.map { $0 * 3 }
// CompletionHandler
class Example {
typealias CompletionHandler = () -> ()
func load(completion: CompletionHandler?) {
// ... operation ...
completion?()
}
}
let example = Example()
example.load { print("completed!") }
```

J. Foundation APIs

Es ist möglich in Swift 4 mit einem minimalen Aufwand Datentypen in JASON-Format zu speichern bzw. einzulesen.

```
// JSON parsen
typealias JSONDict = [String:AnyObject]
func parse(data:Data) throws -> JSONDict {
let obj = try JSONSerialization
.jsonObject(with: data, options: [])
return obj as! JSONDict
}
```

K. Fehlerbehandlung

Die Fehlerbehandlung beschäftigt sich mit dem Abfangen von kritischem Code. Im folgendem werden Beispiele mit verschiedene Abfangmöglichkeiten dargestellt.

```
// try! - Error im Fehlerfall
jsonResult = try! parse(data: data)
// try? - nil im Fehlerfall
jsonResult = try? parse(data: data)
// do ... try ... catch
do {
jsonResult = try parse(data: data)
} catch let error {
print("Error: \(error)")
}
// Eigene ErrorTypes
enum PurchaseError: Error {
case UnknownProduct
case OutOfStock
}
func purchase() throws -> String {
throw PurchaseError.OutOfStock
}
do {
let result = try purchase()
}
catch PurchaseError.OutOfStock { }
catch is PurchaseError { }
catch { }
```

[17] Vgl. [7] und [8]

L. Schwache Referenzen

Eine schwache Referenz ist eine Referenz, die die Instanz, auf die sie sich bezieht, nicht fest im Griff hat, und hält ARC (speichert die Information über Instanzen) nicht davon ab, die referenzierte Instanz zu entfernen. Dieses Verhalten verhindert, dass die Referenz Teil eines starken Referenzzyklus wird. Sie geben eine schwache Referenz an, indem Sie das Schlüsselwort *weak* vor einer Eigenschaft oder einer Variablendeklaration platzieren[18].

```
// Schwache Referenzen
weak var exampleRef : Example?
// Schwache Referenzen in Bloecken
example.load { [weak self] in
print("Completed \(self)!")}
```

V. Fazit

Swift ist eine stabile und zuverlässige Sprache. Sie ist auch wirklich schnell zu erlernen und bietet viele Vorteile, um einfach effizienten Code schreiben zu können. Es wird auch von Apple stark unterstützt, sodass ein jährliches Update mit neuen Funktionen für die nächsten Jahre gesichert ist. Leider kann man als nicht Apple-Nutzer diese Programmiersprache nicht wirklich genießen. Damit man diese Sprache völlig ausschöpfen kann, benötigt man Xcode und dafür Betriebssystem von Apple auf dessen Hardware. Der Weg zur optimalen Swift-Quellcode-Entwicklung ist steinig und nur für Apple Nutzer wirklich garantiert. Swift ist zwar Open-Source, aber nur ”opener” für Kunden, die bereits Produkte von Apple erworben haben. Die Konkurrenz w.z.B. Python oder Ruby machen es dem Nutzer um ein vielfaches einfacher und sind gerade für Neueinsteiger perfekt. Zwar mag Swift gut sein, aber besonders beliebt ist es nicht. Nach der Tiobe (https://www.tiobe.com/tiobe-index/), einer Software Qualität Firma, ist Swift auf Platz 19 (Zur Zeit des letzten Zugriffes). Dies ist weit Unten in der Tabelle[19]. Vielleicht kann Swift 5 mehrere Entwickler begeistern. Im Moment ist Swift, im direkten Vergleich zu anderen Sprachen, eine schlechtere Wahl für den Entwickler. Apple Nutzer können ohne großartige Probleme die Sprache austesten und Benutzer anderer Betriebssysteme können, wenn ihnen der Aufwand nicht groß ist, das gleiche tun. Jedoch wird beiden Gruppen, sofern sie aktuelle Programmiersprachen vergleichen, schnell klar, dass die Wahl da eher auf Java, C, Python oder ähnliches fällt. Trotzdem ist Swift die bessere Wahl als *Objective-C*.

VI. Ausblick

Da die Beliebtheit von Swift 4 nicht gerade hoch ist, kann nur ein Update Swift interessanter machen. Diese neu Auflage würde dann höchstwahrscheinlich auf der diesjährigen WWDC 2018 präsentiert werden. Mit Glück und guter Innovationen ist es möglich, dass Swift das Interesse doch wieder wecken kann. Wenn jedoch alles genannte eintrifft, ist dennoch ein guter Auftritt auf anderen Betriebssystemen notwendig.

[18] Vgl. [7] und [8]
[19] Vgl. [9]

May 2018	May 2017	Change	Programming Language
1	1		Java
2	2		C
3	3		C++
4	4		Python
5	5		C#
6	6		Visual Basic .NET
7	9	⌃	PHP
8	7	⌄	JavaScript
9	-	⌃⌃	SQL
10	11	⌃	Ruby
11	14	⌃	R
12	18	⌃⌃	Delphi/Object Pascal
13	8	⌄⌄	Assembly language
14	16	⌃	Go
15	15		Objective-C
16	17	⌃	MATLAB
17	12	⌄⌄	Visual Basic
18	10	⌄⌄	Perl
19	13	⌄⌄	Swift
20	31	⌃⌃	Scala

Fig. 6. Tiobe Software Qualitätstabelle

References

[1] Michael Kofler, *Swift 4 Das umfassende Handbuch*, Rheinwerk Verlag, Bonn, 1.Auflage 2018, ISBN 978-3-8362-5920-0
[2] Apple Inc, *Swift 4*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 https://developer.apple.com/swift/
[3] Apple Inc, *Swift.org*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 https://swift.org/
[4] Apple Inc, *Xcode 9*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 https://developer.apple.com/xcode/
[5] Apple Inc, *WWDC 2014*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 https://developer.apple.com/videos/wwdc2014//
[6] Apple Inc, *Apple-Rechtliche Fragen*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 https://www.apple.com/de/legal/
[7] Apple Inc, *Swift Dokumentation*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309
[8] Ralf Ebert, *Swift 4 Kurzreferenz*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 http://www.ralfebert.de/ios/Swift-Kurzreferenz.pdf
[9] Tiobe, *Programmiersprachen Ranking*, ***Onlinequelle*** Letzter Zugriff: 13.05.18 https://www.tiobe.com/tiobe-index/

List of Figures