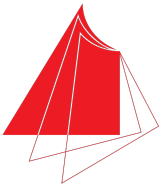
Hochschule Karlsruhe
Technik und Wirtschaft

Fakultät für Informatik und
Wirtschaftsinformatik



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Funktionale Programmierung

in Swift

Seminararbeit vorgelegt von: Agata Jankowski

Matrikelnummer:	37463
Studiengang:	Informatik, B. Sc.
Semester:	SS 2017
Gutachter:	Prof. Dr. Dirk W. Hoffmann

Inhaltsverzeichnis

Abtract

Viele objektorientierte Progammiersprachen bieten sprachintern, durch Standardbibliotheken oder Frameworks die Möglichkeit, funktionales Programmierparadigma in ihren Programmen anzuwenden.
Swift hat den Leitspruch **„Objektorientiert im Großen, funktional im Kleinen“** gekonnt umgesetzt, was in den folgenden Kapiteln über Grundkonzepte der funktionalen Progammierung und deren Umsetzung in Swift dargestellt werden soll.

Funktionale Programmierung versucht Problemstellungen in einzelne Funktionen zu zerlegen, welche unveränderbare Werte annehmen und zurückgeben. Dieses können wir durchaus mit Swift anwenden, jedoch prüft der Swift Compiler nicht, ob Funktionen pur (siehe nächstes Kapitel) sind.
Auch besteht Programmierung mit Swift zu einem großen Teil aus iOS Programmierung, welche objektorientierte Frameworks benutzt, sodass man das funktionale Paradigma realistisch nicht vollkommen umsetzen kann.

Als Ausblick ist das funktionale Paradigma vor allem als Grundstein für weitere Trends wie funktionale reaktive Programmierung, wie sie in Swift schon durch das Framework RxSwift[1] vorzufinden ist, interessant.
Denn das Traversieren von Arrays durch höhere Funktionen (Kapitel 3) ist nichts anderes als ein Stream, welcher zeitlich Elemente abarbeitet und somit reaktive Programmierung beschreibt, für die das Denken in funktionalen Prinzipien erforderlich ist.

[1] https://github.com/ReactiveX/RxSwift

1 Unveränderlichkeit

1.1 Als Teil des funktionalen Paradigma

Objektorientierte Sprachen folgen dem imperativen Progammierparadigma: Beim Ausführen vom Programm werden die Befehle nacheinander abgearbeitet und die Variablen sind in bestimmten (veränderbaren) Zuständen.
Zu den Grundkonzepten der funktionalen Programmierung gehört jedoch die Unveränderlichkeit von Objekten.

Als Gründe dafür sind zu nennen, dass man sich keine Gedanken mehr dazu machen muss, welchen Wert eine Variable zu einem bestimmten Zeitpunkt hat und die Funktion mit der verwendeten Variable **frei von unerwarteten Seiteneffekten** bzw. Nebenwirkungen ist.
Nicht nur ist die so Programmierung berechenbarer, auch besonders bei Multi-Thread Situationen von parallel ablaufenden Prozessen einfacher zu realisieren: Objekte sind zwischen Threads beliebig austauschbar, da diese isoliert sind vom Umfeld.

Listing 1.1: *Man kann f1 und f2 parallel ausführen, da x unveränderlich*

```
1 let a = f1(x, y) + f2(x, z)
```

Eine Funktion in der funktionalen Programmierung ist ähnlich wie mathematische Funktionen. Die $sin(x)$ Funktion gibt immer den selben Wert für das x aus. Praktisch sind Testszenarios so ebenfalls leichter funktional zu organisieren, da bei immer **identischen Parametern auch das gleiche Ergebnis** zustande kommt.
Diese **idempotenten** Funktionen werden auch **Pure Funktionen** genannt - der Rückgabewert ist nur über die Eingabe bestimmt.

> *"Die Definition von Wahnsinn ist, immer wieder das Gleiche zu tun und andere Ergebnisse zu erwarten."* (Albert Einstein)

1.2 Unveränderlichkeit von Swift unterstützt

1.2.1 Unterstützung durch Typdeklaration

Swift hat unveränderlichen Variablen einen besonderen Stellenwert gegeben, indem man die Verwendung von Typdeklarationen mit `let` bei später nicht mehr modifizierbaren Werten besonders hervorhebt.

Um hier auch dem funktionalem Paradigma gerecht zu werden, geht man nach dem „Nicht aktualisieren, sondern neu erschaffen“ Prinzip vor, indem man das Verändern von Variablen vermeidet (siehe Listing 1.2).

Listing 1.2: *Variablen werden neu erstellt statt modifiziert*

```
1 var name = "Max"
2 var name = name + " Mustermann"
3
4 // Funktional
5 let firstname = "Max"
6 let lastname = "Mustermann"
7 let name = firstname + " " + lastname
```

1.2.2 Unterstützung durch Methoden

In Swift hat man ebenfalls Methoden zur Verfügung (Listing 1.3), die einem erlauben, Variablen nicht mutieren zu lassen.

Listing 1.3: *Methoden funktional ausnutzen*

```
1 // mutating
2 x.sort()
3
4 // non-mutating
5 let a = x.sorted()
```

1.2.3 Unterstützung durch Referenztyp

In Swift sind Klassen und Funktionen Referenz-Typen und der Rest (`int`, `Bool`, `Array`, `Dictionary` et cetera) sind Wert-Typen als Struktur umgesetzt.
Dabei haben Klassen `class` als Referenz-Typ die gleichen Eigenschaften wie die Referenz-Typ Strukturen `struct` mit dem Unterschied der Vererbung. Man hat somit die Auswahl, ob man objektorientiert mit Referenzen arbeitet oder funktional mit Wert-Typen, welche kontextunabhängig sind.

2 First Class Funktionen

Funktionen, welche man Variablen zuweisen kann, nennt man First Class Funktionen. Es gibt keinerlei Unterschied zu anderen Variablenwerten wie beispielsweise `int`.

Listing 2.1: *Funktion als Variablenwert*

```
1 let addOne = { $0 + 1 }
2 addOne(5) // 6
```

Diese ermöglichen erst, ein bedeutendes funktionales Werkzeug freizuschalten: Funktionen als Werte an andere Funktionen weitergeben zu können, was bei Funktionen höherer Ordnung (Kapitel 3) wichtig sein wird.

2.1 Closures

Nun ist die Funktion als Wert aus Listing 2.1 kaum erkennbar. Es fehlt die Funktion-Syntax `func FunctionName (Params) -> ReturnType { Statements }`, ersetzt durch einen kurzen Term, welche die Anweisungen andeutet: Es handelt sich hierbei um eine **Closure**.

Als Closures bezeichnet man anonyme Funktionen, welche keinen eigenen Namen besitzen, da sie meistens nur an einer Stelle im Programmierprojekt benutzt werden und nicht namentlich referenzierbar sein müssen.
Die Syntax von Closures ist in Swift vielfälig deklarierbar. Die längste Version wäre `{(Params) -> ReturnType in Statements}`.

Listing 2.2: *Beispiel einer langen Closure Deklaration*

```
1 let even = { (i: Int) -> Bool in
2     return i % 2 == 0
3 }
4
5 even(3) // false
```

2.2 Sehr kurze Closure Syntax

Swift geht sogar so weit, dass man die Argumente anonymisieren kann und diese mit `$0, $1` bis `$n` nach Aufführungsreihenfolge der Parameter ansprechen kann, wie in Listing 2.1 vorgeführt.

Swift unterstützt somit, anonyme Funktionen möglichst minimal zu deklarieren und die Funktionsaufrufe aus Listing 2.3 sind allesamt identisch in ihrer Funktion, gerade Zahlen des Arrays `numbers` in ein neues Array zu filtern.

Listing 2.3: *Austauschbare verkürzte Closure Syntax*

```
1 numbers.filter({ (i:Int) -> Bool in return i % 2 == 0})
2 numbers.filter({ i in return i % 2 == 0})
3 numbers.filter({ i in i % 2 == 0 })
4 numbers.filter({ $0 % 2 == 0 })
5 numbers.filter { $0 % 2 == 0 }
```

Obiges Beispiel zeigt gleich zwei interessante Aspekte der funktionalen Programmierung: Es wird die Funktion `filter` angewendet auf alle Elemente eines Arrays mit dem Statement einer anonymen Funktion.
Somit ist die Closure `{ $0 % 2 == 0 }` Argument für die Funktion `filter`.

3 Funktionen höherer Ordnung

Da alle Funktionen in Swift First Class Funktionen sind, kann man auch Funktionen als Rückgabewerte von Funktionen und/oder als Argumente für Funktionen benutzen, welche Funktionen höherer Ordnung genannt werden.

3.1 Die meistgenutztesten Werkzeuge funktionaler Programmierung

Die bekanntesten Funktionen höherer Ordnung sind `reduce`, `map` und `filter`, welche auf Swift Kollektionen wie `Array` oder `Dictionary` benutzt werden können, welche in der Swift Standardbibliothek vorhanden sind.

3.1.1 Filter

So wird `filter` in Listing 2.3 dazu genutzt, ein neues Array aus allen Elementen des Arrays `numbers` zu bilden, welche die Kondition der Closure erfüllen.

3.1.2 Reduce

Alle Elemente einer Kollektion werden zu einem Array zusammengefasst, dabei nimmt `reduce` zwei Argumente: Einen initialen Wert und eine Closure zum Zusammenfügen der Elemente.
Die Shorthand Closure Syntax ist auch hier anwendbar und sogar nur auf den Operator reduzierbar mit beispielsweise `reduce(0,+)` statt `reduce(0,{$0 + $1})` für das simple Aufsummieren der Elemente.

Listing 3.1: *Reduce Beispiel*

```
1 let nameArray = ["Mustermann", "Max"]
2 let name = nameArray.reduce("",combine:{$1 + " " + $0})
3 name // "Max Mustermann"
```

3.1.3 Map

Bei `map` führt man die Closure Operation auf jedes einzelne Element der Kollektion aus.

Listing 3.2: *Map Beispiel*

```
let numbers = Array(1...3)
let tripleNumbers = numbers.map { $0 * 3 }
tripleNumbers // [3,6,9]
```

3.1.4 Flatmap

Um eine Kollektion in einer Kollektion zu einer Kollektion zusammenzufügen, ist `flatMap` ideal.

Listing 3.3: *FlatMap Beispiel*

```
let nameArrays = [["Max"],[" ","Mustermann"]]
let name = nameArray.flatMap { $0 }
name // "Max Mustermann"
```

3.2 Verketten höherer Funktionen

Das höchste Potential in der Arbeit mit höheren Funktionen ist es, diese miteinander zu verketten. So kann man in Listing 3.4 alle geraden Zahlen vom Array `numbers` aufsummieren mit der Verkettung von `filter` und `reduce`.

Listing 3.4: *Verketten von Funktionen*

```
let numbers = Array(1...10)
let evenSum = numbers.filter {$0 % 2 == 0}.reduce(0,+)
```

3.3 Funktionen als Argument

Um eine Funktion höherer Ordnung zu sein, muss man entweder eine andere Funktion als Parameter übergeben bekommen oder als Rückgabewert eine Funktion haben. In Listing 3.5 dient die Funktion `addOne` als Argument für die höhere Funktion `map`, welche die übergebene Funktion auf alle Elemente vom Array anwendet.

Listing 3.5: *Optionale mit Map auspacken*

```
func addOne(x: Int) -> Int {
    return x + 1 }

(1...3).map(addOne) // [2,3,4]
```

3.4 Funktionen als Rückgabewert

In Listing 3.6 finden beide Fälle ein: Es wird die Funktion `addOne` aus Listing 3.5 als Parameter durch `(f: Int -> Int)` übergeben und es wird eine Funktion `(Int -> Int)` ausgegeben.

Listing 3.6: *Optionale mit Map auspacken*

```
1 func addTwoAfter(f: Int -> Int) -> (Int -> Int) {
2     return { f($0) + 2 }
3 }
4
5 let addThree = addTwoAfter(addOne)
6 addThree(1) // 4
```

Wenn wir beim Funktionsaufruf `addThree` die `1` übergeben, geben wir somit die Funktion `addOne(1) + 2` zurück, also `(1 + 1) + 2`.

4 Optionale aus der Sicht funktionaler Programmierung

In Swift darf kein Wert `nil` sein, außer er ist im Datentyp ein Optional.
Optionale gehören zu den einschränkensten Konstrukten in Swift. Einen Wert in einem Optional verpackt zu haben, fügt eine zusätzliche Bearbeitungsstufe hinzu, um sicherzustellen, dass man nicht mit `nil` arbeitet.

4.1 Leichteres Auspacken von Optionalen

Um an die Werte von Optionalen in Swift zu kommen, muss man bisher umständig mit `if let` Konditionalen oder `guard` Zusicherungen arbeiten, um Fehlermeldungen bei Operationen auf `nil` zu vermeiden.

Listing 4.1: *Optional klassisch auspacken*

```
1  func addOne(someNumber: Int?) -> Int? {
2      if let number = someNumber {
3          return number + 1
4      } else {
5          return nil
6      }
7  }
8
9  addOne(5)   // Optional(6)
10 addOne(nil) // nil
```

Dieses wird schnell unleserlich, wenn man mehrere hintereinander gekettete Optionales auspacken will, wie man es oft in der UI Programmierung machen muss.

Die Implemetierung von `map` in der Swift Standardbibliothek hilft uns dabei, mit einer eleganteren Art mit Optionalen zu arbeiten, da eine Exception geworfen wird im Falle von `nil` und die Operationen somit nur auf alle anderen Werte angewendet werden.
Hierbei ist Listing 4.2 identisch zu Listing 4.1 im Ergebnis.

Listing 4.2: *Optionale mit Map auspacken*

```
1 func addOne(someNumber: Int?) -> Int? {
2     return someNumber.map { number in number + 1 }
3 }
4
5 addOne(5)   // Optional(6)
6 addOne(nil) // nil
```

4.2 `nil` aussortieren

Noch interessanter ist die Implementation von `flatMap`, da diese `nil` gar nicht wiedergeben darf und somit gut missbraucht werden kann, um `nil` aus Kollektionen auszusortieren.

Listing 4.3: *FlatMap darf keine `nil` Werte ausgeben*

```
1 [["a"], [nil, "b"]].flatMap { $0 } // ["a", "b"]
```

4.3 Optionale sind Monaden

Eine Monade ist ein Entwurfsmuster und gehört zum Konzept der funktionalen Programmierung.
Die Definition von Monaden sind Werte, welche gekapselt einen neuen Datentyp ergeben (und in Swift auch teilsweise durch `typealias` realisierbar sind).
Da höheren Funktionen wie `map` auf allen Monaden implementiert sind, ist der wahre Grund, weshalb es sich funktional so leicht mit Optionals leben lässt!