

UNIVERSITÄT HEIDELBERG  
SOMMERSEMESTER 2015

# Softwareentwicklung für iOS

APP KATALOG

NILS FISCHER

Aktualisiert am 18. April 2015  
Kursdetails und begleitende Materialien auf der Vorlesungswebseite:  
<http://ios-dev-kurs.github.io>

# Inhaltsverzeichnis

<b>1</b>	<b>Über dieses Dokument</b>	<b>3</b>
<b>2</b>	<b>Hello World</b>	<b>4</b>
2.1	Grundlagen der Programmierung in Swift . . . . .	4
2.2	Objektorientiertes "Hello World!" . . . . .	8

# Kapitel 1

## Über dieses Dokument

Dieser App Katalog enthält Schritt-für-Schritt Anleitungen für die im Rahmen unseres Kurses erstellten Apps sowie die wöchentlich zu bearbeitenden Übungsaufgaben und wird im Verlauf des Semesters kapitelweise auf der Vorlesungswebseite <sup>[1]</sup> zur Verfügung gestellt.

Er dient jedoch nur als Ergänzung zum parallel verfügbaren **Skript**, auf das hier häufig verwiesen wird. Dort sind die Erläuterungen zu den verwendeten Technologien, Methoden und Begriffen zu finden.

---

<sup>1</sup><http://ios-dev-kurs.github.io/>

## Kapitel 2





# Hello World

Was ist schon ein Programmierkurs, der nicht mit einem klassischen *Hello World* Programm beginnt? Wir werden jedoch noch einen Schritt weitergehen und diesen Gruß vom iOS Simulator oder, soweit vorhanden, direkt von unseren eigenen iOS Geräten ausgeben lassen. Außerdem wird in die objektorientierte Programmierung in *Swift* eingeführt.

*Relevante Kapitel im Skript: Xcode, Programmieren in Swift sowie das Buch The Swift Programming Language [1]*

### 2.1 Grundlagen der Programmierung in Swift

Anhand des ersten Kapitels *A Swift Tour* des Buches *The Swift Programming Language* lernen wir zunächst die Grundlagen der Programmierung in Swift kennen.

1. Öffnet Xcode und erstellt zunächst einen *Playground* mit  +  +  + . Playgrounds sind interaktive Skripte, mit denen sich ideal Code ausprobieren lässt. Gebt der Datei einen Namen wie "**01 – Grundlagen der Programmierung in Swift**" und speichert sie in einem Verzeichnis für diesen Kurs.
2. Ein Playground besteht aus einem Editor- und einem Inspektorbereich und führt geschriebenen Code automatisch aus. Ausgaben und Laufzeitinformationen werden im Inspektor angezeigt. In nur einer Zeile Code können wir den traditionellen *Hello World!*-Gruß ausgeben lassen (s. S. 5, Abb. 2.1).

---

```
1 println("Hello World!")
```

---

---

<sup>1</sup>[https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift\\_Programming\\_Language/](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/)

3. Nun lernen wir anhand des ersten Kapitels *A Swift Tour* des Buches *The Swift Programming Language* zunächst die Grundlagen der Programmierung in Swift.

Auf der Vorlesungswebseite findet ihr den Playground aus der Vorlesung, der in diese Konzepte einführt. Macht euch dabei mit folgenden Begriffen vertraut:

- Variablen (**var**) und Konstanten(**let**)
- Einfache Datentypen (**Int**, **Float**, **Double**, **Bool**, **String**, **Array**, **Dictionary** und **Set**)
- Type Inference
- String-Formatierung
- Einfache Operatoren (+, -, \*, /, \%)
- Abfragen (**if**, **switch**) und Schleifen (**for**, **while**)
- Optionals
- Funktionen

Im zweiten Kapitel *Language Guide* in *The Swift Programming Language* werden diese Konzepte noch einmal detailliert erklärt. Informiert euch dort gerne genauer darüber. Zunächst genügt es jedoch, einen Überblick zu erhalten. Im Verlauf des Kurses werden wir noch viel Übung im Umgang mit diesen Konzepten bekommen.

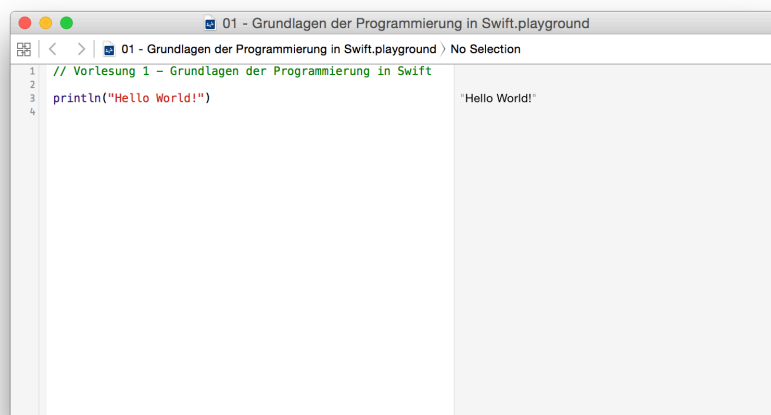


Abbildung 2.1: Playgrounds eignen sich ideal zum Ausprobieren von Swift Code.

## Übungsaufgaben

### 1. Fibonacci [1 P.]

- a) Schreibt einen Algorithmus, der alle Folgenglieder  $F_n < 1000$  der Fibonaccifolge

$$F_n = F_{n-1} + F_{n-2} \quad (2.1)$$

$$F_1 = 1, F_2 = 2 \quad (2.2)$$

in der Konsole ausgibt.

- b) **Extra:** Bei jeder geraden Fibonaccizahl  $F_j$  ist der Abstand  $\Delta n = j - i$  zum vorherigen geraden Folgenglied  $F_i$  auszugeben.

### 2. Primzahlen [2 P.]

- a) Schreibt eine Funktion `primeNumbersUpTo:`, die ein Argument `maxNumber: Int` annimmt und alle Primzahlen bis `maxNumber` als Liste `[Int]` zurückgibt.

**Hinweis:** Mit dem Modulo-Operator `%` kann der Rest der Division zweier Integer gefunden werden:

---

```
1 let a = 20%3 // a ist jetzt Int(2)
```

---

- b) *Optionals* sind eines der elegantesten Konzepte in Swift, und sind auch in anderen modernen Sprachen zu finden. Informiert euch darüber im Kapitel *Language Guide > The Basics > Optionals* in *The Swift Programming Language*. Dieses Kapitel (bis einschließlich *Implicitly Unwrapped Optionals*) ist sehr wichtig, da wir in der iOS App Programmierung häufig mit Optionals arbeiten werden!

- c) Verwendet eure Liste von Primzahlen aus der vorigen Aufgabe, um effizienter zu prüfen, ob eine Zahl eine Primzahl ist.

Schreibt dazu eine Funktion `isPrimeNumber:cachedPrimeNumbers:`, die eine Zahl `n: Int` und eine **optionale** Liste von Primzahlen `cachedPrimeNumbers: [Int]?` annimmt. Verwendet die *Optional Binding Syntax* `if let` um mit dieser Liste zu arbeiten, wenn eine solche übergeben wurde und lang genug ist. Dann genügt es zu prüfen, ob die Zahl in der Liste enthalten ist. Wenn keine Liste übergeben wurde, soll die Primzahl wie in a) manuell geprüft werden.

**Hinweise:**

- Dem Argument `cachedPrimeNumbers` können wir einen *default* Wert `nil` zuweisen:

---

```
1 func isPrimeNumber(n: Int, cachedPrimeNumbers: [Int]? = nil) ->
  ↪ Bool {
2     // ...
3 }
```

---

So kann die Funktion auch ohne dieses Argument aufgerufen werden:

---

```

1 isPrimeNumber(7, cachedPrimeNumbers: [ 1, 2, 3, 7 ]) //
  ↳ vollständiger Funktionsaufruf, verwendet übergebene Liste zum
  ↳ Nachschlagen
2 isPrimeNumber(7) // äquivalent zu:
3 isPrimeNumber(7, cachedPrimeNumbers: nil) // Prüft Primzahl
  ↳ manuell

```

---

- Die globale Funktion `contains` prüft ob eine Element in einer Liste enthalten ist:

---

```

1 contains([ 1, 2, 3, 7 ], 7) // true

```

---

- Testet eure Funktion, indem Ihr bspw. folgenden Code ans Ende des Storyboards setzt:

---

```

1 //: ## Testing
2
3 let n = 499 // Number to test
4 let cachedMax = 500 // Prime numbers up to this number will be
  ↳ cached
5 import Foundation
6 var startDate: NSDate
7
8 startDate = NSDate()
9 let cachedPrimeNumbers = primeNumbersUpTo(cachedMax)
10 println("Time for caching prime numbers up to \(cachedMax):
  ↳ \(-startDate.timeIntervalSinceNow)s")
11
12 startDate = NSDate()
13 if isPrimeNumber(n) {
14     println("\(n) is a prime number.")
15 } else {
16     println("\(n) is not a prime number.")
17 }
18 let withoutCacheTime = -startDate.timeIntervalSinceNow
19 println("Time without cache: \(withoutCacheTime)s")
20
21 startDate = NSDate()
22 isPrimeNumber(n, cachedPrimeNumbers: cachedPrimeNumbers)
23 let withCacheTime = -startDate.timeIntervalSinceNow
24 println("Time with cache: \(withCacheTime)s (((1 - withCacheTime
  ↳ / withoutCacheTime) * 100)% faster)")

```

---

## 2.2 Objektorientiertes "Hello World!"

1. Nun versuchen wir uns an der objektorientierten Programmierung und möchten den Hello World! Gruß von virtuellen Repräsentationen einzelner Personen ausgeben lassen. Erstellt dazu einen Xcode Playground bspw. mit Titel "**02 – Objektorientierte Programmierung in Swift**".
2. Verschafft euch anhand *The Swift Programming Language* und dem Playground aus der Vorlesung (auf der Vorlesungswebseite) einen Überblick über folgende Konzepte der objektorientierten Programmierung in Swift:
  - Klassen und Objekte
  - Attribute mit oder ohne Startwert
  - Initializer
  - Instanz- und Klassenmethoden
  - Subklassen und Überschreiben von Methoden
  - Structs und Enums

### Übungsaufgaben

#### 3. Scientists

[1 P.]

- a) Erstellt (am besten in einem neuen Playground, in den ihr die Klasse `Person` aus der Vorlesung einfügt) eine weitere Klasse `Scientist` als *Subklasse* von `Person`.

Wissenschaftler können rechnen, fügt dieser Klasse also eine Methode `sayPrimeNumbersUpTo:` hinzu, die ein Argument `maxNumber: Int` annimmt und alle Primzahlen bis zu dieser Zahl in der Konsole ausgibt. Verwendet dazu den Algorithmus aus der vorherigen Übungsaufgabe (s. S. 6, Übungsaufgabe 2).

**Hinweis:** Wie in *The Swift Programming Language* beschrieben, erbt eine Subklasse die Attribute und Methoden ihrer Superklasse und kann diese überschreiben:

---

```
1 class Scientist: Person {  
2     ...  
3 }
```

---

- b) Wir wollen uns vergewissern, dass die Klasse `Scientist` die Attribute und Methoden ihrer Superklasse `Person` erbt. Erstellt ein `Scientist`-Objekt, gebt ihm einen Namen und lasst den Hello World-Gruß ausgeben.



- c) Nach dem Prinzip des *Überschreiben* soll ein Wissenschaftler einen anderen Gruß ausgeben als eine "normale" Person. Überschreibt in der `Scientist`-Klasse die Methode `sayHello`, sodass zusätzlich **"Ask me for prime numbers!"** ausgegeben wird.

#### 4. Poker

[3 P.]

In dieser Aufgabe berechnen wir die Wahrscheinlichkeit für einen *Flush* beim Poker.

- a) Zunächst modellieren wir die Spielkarten. Eine Karte hat immer eine *Farbe/Suit* (*Karo/Diamonds*, *Herz/Hearts*, *Pik/Spades* oder *Kreuz/Clubs*) und einen *Rang/Rank* (2 bis 10, *Bube/Jack*, *Dame/Queen*, *König/King* oder *Ass/Ace*).

Schreibt zwei Enums `enum Suit: Int` und `enum Rank: Int` mit ihren entsprechenden Fällen (`case Diamonds` usw.). Bei den Rängen 2 bis 10 schreibt ihr am besten die Zahl aus. Implementiert jeweils eine *Computed Property* `var symbol: String`, in der ihr mithilfe einer `switch`-Abfrage für jeden Fall ein Symbol zurückgibt. **Tipp:** Für die Farben gibt es Unicode-Symbole<sup>[2]</sup>!

Schreibt dann einen `struct Card` mit zwei Attributen `let suit: Suit` und `let rank: Rank`, sowie einer *Computed Property* `var description: String`, die einen aus Farbe und Rang zusammengesetzten String zurückgibt.

- b) Nun können wir eine Poker Hand modellieren. Schreibt den `struct Poker-Hand` mit einem Attribut `let cards: [Card]` und einer *Computed Property* `var description: String`, die die description der Karten kombiniert.

Um einfach zufällige Poker Hände generieren zu können, implementiert einen Initializer `init()`, der eine Hand aus fünf zufälligen Karten erstellt. **Wichtig:** Da aus einem Deck von paarweise verschiedenen Karten gezogen wird, darf keine Karte doppelt vorkommen.

##### Hinweise:

- Da wir `Suit` und `Rank` von `Int` abgeleitet haben, können wir Zufallszahlen generieren und die Enums daraus erstellen:

---

```
1 let rndSuit = Suit(rawValue: Int(arc4random_uniform(4)))!
2 let rndRank = Rank(rawValue: Int(arc4random_uniform(13)))!
3 let rndCard = Card(suit: rndSuit, rank: rndRank) // Eine
  ↪ zufällige Spielkarte
```

---

- Die Funktion `contains` könnte hilfreich sein, um das Vorhandensein von Karten zu überprüfen. Um diese mit `Card` verwenden zu können, müsst ihr erst eine Äquivalenzrelation implementieren: Schreibt `struct Card: Equatable { ... }` und dann außerhalb des Struct:

<sup>2</sup>[http://en.wikipedia.org/wiki/Playing\\_cards\\_in\\_Unicode](http://en.wikipedia.org/wiki/Playing_cards_in_Unicode)

---

```

1 func ==(lhs: Card, rhs: Card) -> Bool {
2     return lhs.suit == rhs.suit && lhs.rank == rhs.rank
3 }

```

---

- c) Erstellt ein paar Poker Hände und lasst euch die description ausgeben. Habt ihr etwas gutes gezogen?

Implementiert nun ein weiteres Enum `enum Ranking: Int` mit den Fällen `case HighCard, Flush, StraightFlush` usw., die ihr bspw. auf Wikipedia<sup>[3]</sup> findet.

Fügt dann dem `struct PokerHand` eine Computed Property `var ranking: Ranking` hinzu. Implementiert hier einen Algorithmus, der prüft, ob ein `Flush` vorliegt. Dann soll `.Flush` zurückgegeben werden, ansonsten einfach `.HighCard`.

- d) Wir können nun einige tausend Hände generieren und die Wahrscheinlichkeit für einen Flush abschätzen. Fügt einfach folgenden Code am Ende des Playgrounds ein:

---

```

1 var rankingCounts = [Ranking : Int]()
2 let samples = 1000
3 for var i=0; i<samples; i++ {
4     let ranking = PokerHand().ranking
5     if rankingCounts[ranking] == nil {
6         rankingCounts[ranking] = 1
7     } else {
8         rankingCounts[ranking]!++
9     }
10 }
11 for (ranking, count) in rankingCounts {
12     println("The probability of being dealt a \(ranking.description)
13     ↪ is \(Double(count) / Double(samples) * 100)%")
14 }

```

---

Die Ausführung kann etwas dauern, justiert ggfs. `samples`. Stimmt die Wahrscheinlichkeit etwa mit der Angabe auf Wikipedia überein?

- e) **Extra:** Ihr könnt das Programm nun noch erweitern und versuchen, die anderen Ränge zu überprüfen. Dabei könnten Hilfsattribute wie `var hasFlush: Bool` oder `var pairCount: Int` nützlich sein. Bekommt es jemand es jemand hin, eine Funktion zu schreiben, die zwei Hände vergleicht und den Sieger bestimmt? **Tipp:** Dazu könnte es hilfreich sein, die Fälle des `enum: Ranking` um *Associated Attributes* zu erweitern.

---

<sup>3</sup>[http://en.wikipedia.org/wiki/List\\_of\\_poker\\_hands](http://en.wikipedia.org/wiki/List_of_poker_hands)