

UNIVERSITÄT HEIDELBERG
SOMMERSEMESTER 2015

Softwareentwicklung für iOS

APP KATALOG

NILS FISCHER

Aktualisiert am 23. April 2015
Kursdetails und begleitende Materialien auf der Vorlesungswebseite:
<http://ios-dev-kurs.github.io>

Inhaltsverzeichnis

1	Über dieses Dokument	3
2	Hello World	4
2.1	Grundlagen der Programmierung in Swift	4
2.2	Objektorientiertes "Hello World!"	8
2.3	Das erste Xcode Projekt	11
2.4	"Hello World!" on Simulator	12
2.5	"Hello World!" on Device	13
2.6	Graphisches "Hello World!"	14

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 ^[1] 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.

¹<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!")
```

¹https://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.

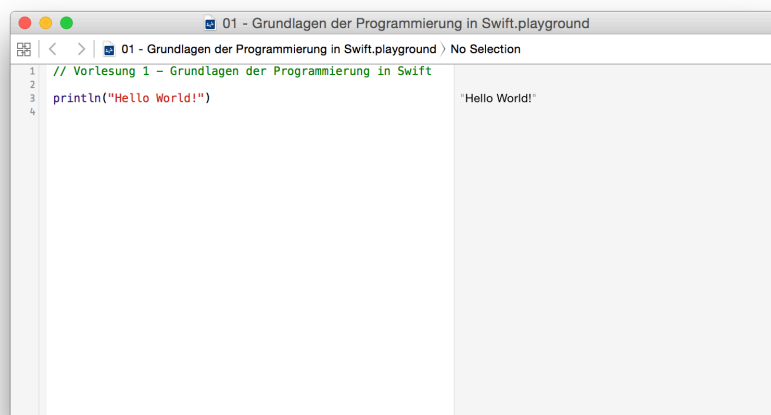


Abbildung 2.1: Playgrounds eignen sich ideal zum Ausprobieren von Swift Code.

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^[2]!

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 PokerHand` 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:

²http://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^[3] 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.

³http://en.wikipedia.org/wiki/List_of_poker_hands

2.3 Das erste Xcode Projekt

iOS Apps schreiben wir natürlich nicht in Playgrounds. Eine App besteht aus vielen Komponenten wie Klassen, Interface-Dateien und weiteren Ressourcen, die wir in einem Xcode Projekt zusammenfassen.

Relevante Kapitel im Skript: Xcode

1. Um nun unsere erste iOS App zu erstellen, rufen wir mit $\text{⌘} + \text{⌥} + \text{N}$ zunächst den Dialog zur Erstellung eines neuen Projekts auf und wählen das Template **iOS Application** **>> Single View Application**.
2. Tragt im erscheinenden Konfigurationsdialog entsprechend der Konventionen den Product Name **"HelloWorld"**, euren Vor- und Nachnamen als Organization Name und **"de.uni-hd.<deinname>"** als Company Identifier ein (s. S. 11, Abb. 2.2). Das führt zu der Bundle ID **"de.uni-hd.<deinname>.HelloWorld"**. Wählt **Swift**, **Universal** und entfernt die Auswahl von **Use Core Data**. Speichert das Projekt in einem Verzeichnis eurer Wahl.

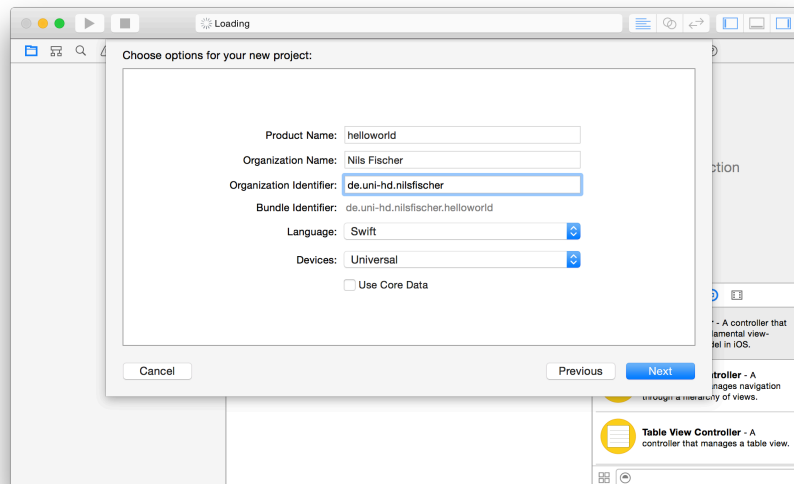


Abbildung 2.2: Damit es keine Konflikte zwischen verschiedenen Apps gibt, gibt es Konventionen bei der Konfiguration

3. Wir sehen nun Xcodes Benutzeroberfläche und können sie mit den Schaltflächen rechts in der Toolbar anpassen. Verwendet zunächst die Konfiguration mit eingeblen-detem Navigator, verstecktem Debug-Bereich und Inspektor und Standard-Editor. Wählt im Project Navigator das Projekt selbst aus (s. S. 12, Abb. 2.3).
4. Im Editor wird die *Projekt- und Targetkonfiguration* angezeigt. Hier können wir bspw. die Bundle ID unserer App anpassen, die wir zuvor bei der Erstellung des Projekts aus Product Name und Company Identifier zusammengesetzt haben.

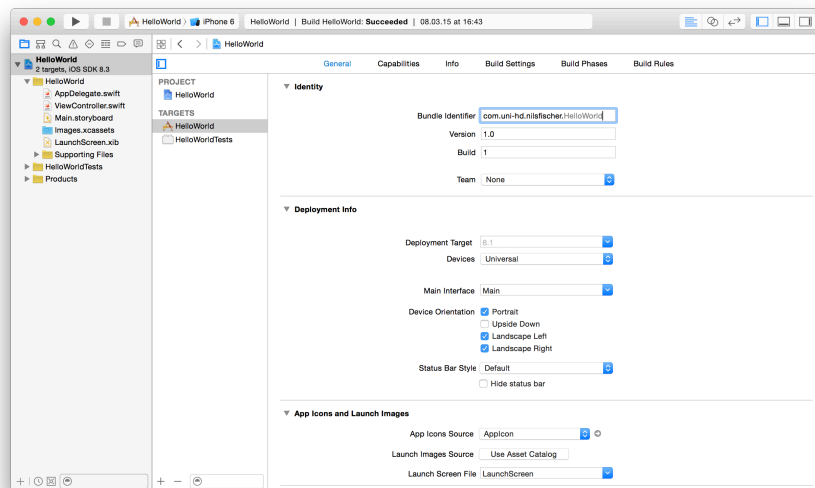
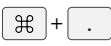
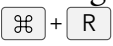


Abbildung 2.3: Wird das Projekt ausgewählt, sehen wir im Editor die Projekt- und Targetkonfiguration.

- Links in der Toolbar sind die Steuerelemente des Compilers zu finden. Wählt das gerade erstellte Target und ein Zielsystem aus, bspw. den *iPhone 6* Simulator, und klickt die *Build & Run* Schaltfläche. Das Target wird nun kompiliert und generiert ein *Product*, also unserer App, die im Simulator ausgeführt wird. In Xcode kann mit  die Ausführung gestoppt und mit  (Tastenkürzel für *Build & Run*) erneut gestartet werden.

2.4 "Hello World!" on Simulator

- Besonders spannend ist diese App natürlich noch nicht. Das ändern wir jetzt spektakulär, indem wir eine Ausgabe hinzufügen. Wählt die Datei *AppDelegate.swift* im Project Navigator aus.
- Die Methode `application:didFinishLaunchingWithOptions:` wird zu Beginn der Ausführung der App aufgerufen. Ersetzt den Kommentar dort mit dem bekannten Gruß zur Ausgabe in der Konsole:

```

1 func application(application: UIApplication,
  ↳ didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
  ↳ -> Bool {
2     println("Hello World!")
3     return true
4 }

```

3. Wenn wir unsere App nun erneut mit *Build & Run* kompilieren und ausführen, sehen wir den Text **"Hello World!"** in der Konsole. Dazu wird der zweigeteilte Debug-Bereich unten automatisch eingeblendet (s. S. 13, Abb. 2.4). Ist der Konsolenbereich zunächst versteckt, kann er mit der Schaltfläche in der rechten unteren Ecke angezeigt werden. Außerdem wird links automatisch zum Debug Navigator gewechselt, wenn eine App ausgeführt wird, in dem CPU- und Speicherauslastung überwacht werden können und Fehler und Warnungen angezeigt werden, wenn welche auftreten.

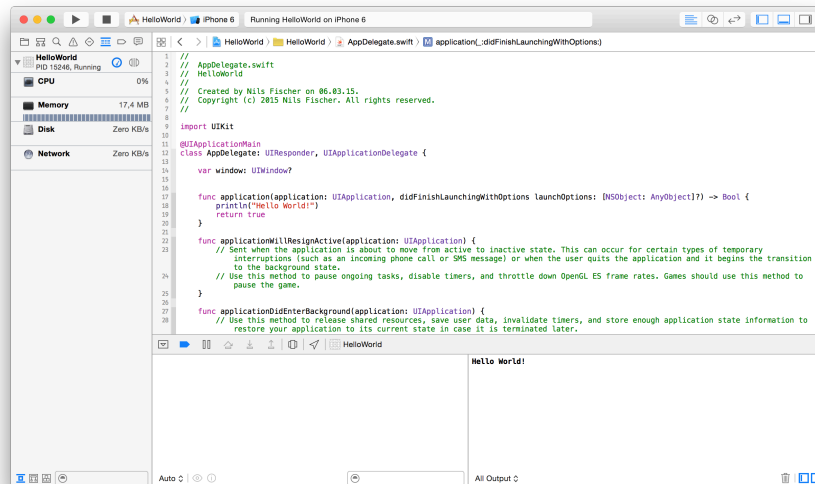


Abbildung 2.4: In der Konsole des Debug-Bereichs werden Ausgaben der laufenden App angezeigt

2.5 "Hello World!" on Device

1. Nun möchten wir unsere neue App natürlich auch auf einem realen iOS Gerät anstatt des Simulators testen. Im Skript findet ihr eine Anleitung, wie ihr mit euren iOS Geräten unserem Developer Team der Uni Heidelberg beitreten könnt.
2. Habt ihr die Schritte befolgt und euren freigeschalteten Apple Developer Account in den Xcode-Accounteinstellungen hinzugefügt, öffnet ihr wieder die Projekt- und Targetkonfiguration im Project Navigator und wählt dort unser Developer Team (s. S. 14, Abb. 2.5) aus. Nun wird automatisch das richtige Provisioning Profile für die Bundle ID des Targets verwendet.
3. Verbindet euer iOS Gerät mit eurem Mac und wählt es in der Toolbar als Zielsystem aus. Mit einem *Build & Run* wird die App nun kompiliert, auf dem Gerät installiert und ausgeführt. In der Konsole erscheint wieder die Ausgabe **"Hello World!"**, diesmal direkt vom Gerät ausgegeben.

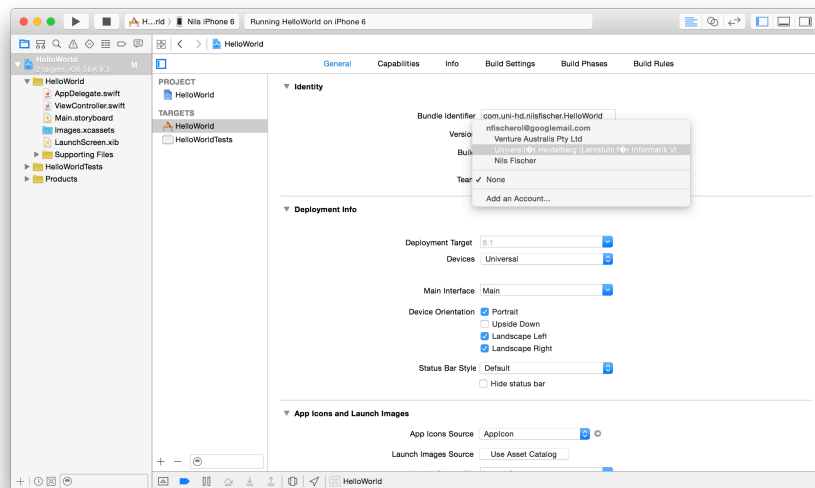




Abbildung 2.5: Mit der Wahl des zugehörigen Developer Teams in der Project- und Targetkonfiguration verwendet Xcode automatisch das passende Provisioning Profile

2.6 Graphisches "Hello World!"

Natürlich wird ein Benutzer unserer App von den Ausgaben in der Konsole nichts mitbekommen. Diese dienen bei der Programmierung hauptsächlich dazu, Abläufe im Code nachzuvollziehen und Fehler zu finden. Unsere App ist also nur sinnvoll, wenn wir die Ausgaben auch auf dem Bildschirm darstellen können.

Relevante Kapitel im Skript: Xcode

1. Zur Gestaltung der Benutzeroberfläche oder *User Interface (UI)* verwenden wir den in Xcode integrierten *Interface Builder (IB)*. Wir haben bei der Projekterstellung der ersten App das *Single View*-Template ausgewählt, daher enthält das Projekt bereits ein *Storyboard*. Öffnet das Projekt und wählt im Navigator die Datei *Main.storyboard* aus.
2. Der Editor-Bereich zeigt nun den Interface Builder. In diesem Modus möchten wir häufig eine angepasste Konfiguration des Xcode-Fensters verwenden, es bietet sich also an, mit  +  einen neuen Tab zu öffnen. Blendet dann mit den Schaltflächen in der Toolbar den Navigator- und Debug-Bereich aus und den Inspektor ein. Wählt dort außerdem zunächst den Standard-Editor (s. S. 15, Abb. 2.6).
3. Unser UI besteht bisher nur aus einer einzigen Ansicht, oder *Scene*. Ein Pfeil kennzeichnet die Scene, die zum Start der App angezeigt wird. Im Inspektor ist unten die *Object Library* zu finden. Wählt den entsprechenden Tab aus, wenn er noch nicht angezeigt wird (s. S. 15, Abb. 2.6).
4. Durchsucht die Liste von Interfaceelementen nach einem Objekt der Klasse `UILabel`, indem ihr das Suchfeld unten verwendet, und zieht ein Label irgendwo auf die erste Scene. Doppelklickt auf das erstellte Label und tippt "Hello World!".

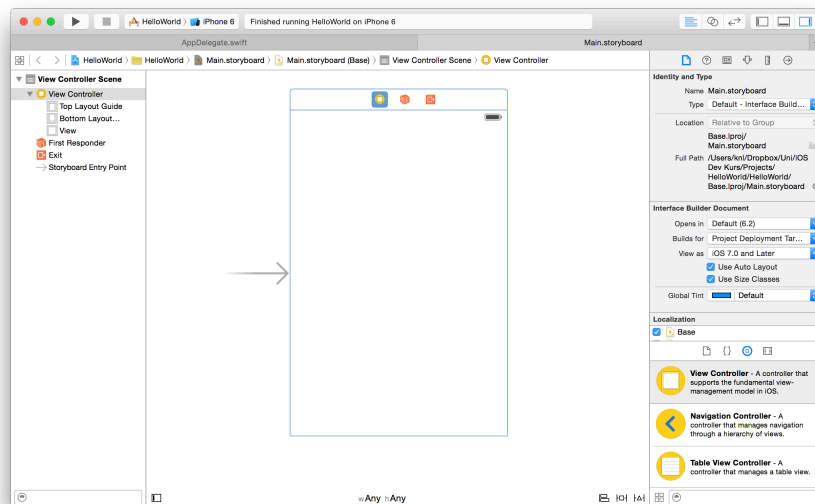


Abbildung 2.6: Für den Interface Builder verwenden wir eine angepasste Fensterkonfiguration mit dem Inspektor anstatt des Navigators

5. Ein *Build & Run* mit einem iPhone-Zielsystem zeigt diesen Gruß nun statisch auf dem Bildschirm an.
6. Habt ihr das Label im Interface Builder ausgewählt, zeigt der Inspektor Informationen darüber an. Im *Identity Inspector* könnt ihr euch vergewissern, dass das Objekt, was zur Laufzeit erzeugt wird und das Label darstellt, ein Objekt der Klasse `UILabel` ist. Im *Attributes Inspector* stehen viele Optionen zur Auswahl, mit denen Eigenschaften wie Inhalt, Schrift und Farbe des Labels angepasst werden können.
7. Natürlich möchten wir unser UI zur Laufzeit mit Inhalt füllen und den Benutzer mit den Interfaceelementen interagieren lassen können. Zieht ein `UIButton`- und `UITextField`-Objekt auf die Scene und positioniert sie passend (s. S. 16, Abb. 2.7). Mit dem Attributes Inspector könnt ihr dem Button nun den Titel "Say Hello!" geben und für das Text Field einen Placeholder "Name" einstellen.
8. Zur Laufzeit der App wird für jedes im Storyboard konfigurierte Interfaceelement ein Objekt der entsprechenden Klasse erstellt und dessen Attribute gesetzt. Um nun im Code auf die erstellten Objekte zugreifen und auf Benutzereingaben reagieren zu können, verwenden wir *IBOutlet*s und *IBActions*.

Blendet den Inspektor aus und wählt stattdessen den Assistant-Editor in der Toolbar. Stellt den Modus in der Jump bar auf *Automatic*. Im Assistant wird automatisch die Implementierung des übergeordneten View Controllers eingeblendet (s. S. 17, Abb. 2.8).

9. *View Controller* sind Objekte einer Subklasse von `UIViewController`, die jeweils einen Teil der App steuern. Diese sind zentrale Bestandteile einer App, mit denen wir uns noch detailliert beschäftigen werden. Ein erster View Controller zur Steuerung dieser ersten Ansicht wurde bereits bei der Projekterstellung automatisch hinzugefügt.

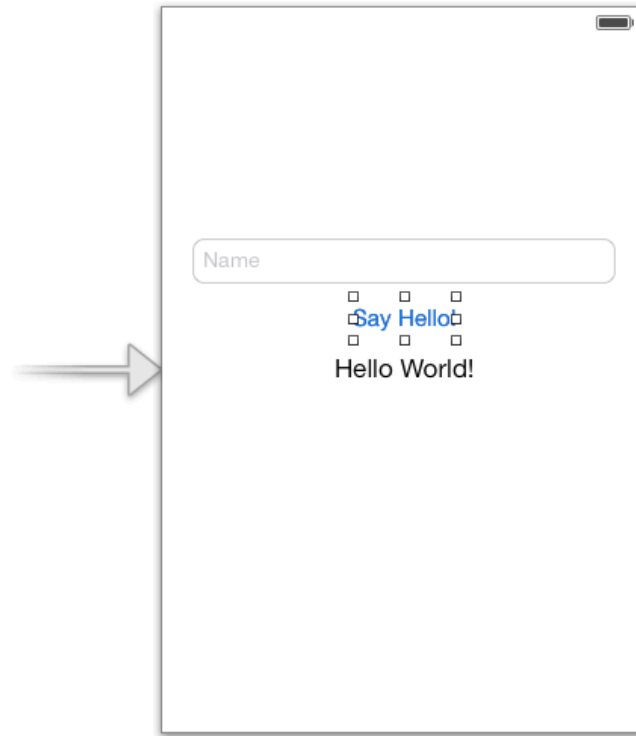


Abbildung 2.7: Mit einem Text Field, einem Button und einem Label erstellen wir ein simples UI

Fügt dieser Klasse ViewController: `UIViewController` Attribute für das `UILabel` und das `UITextField` hinzu und kennzeichnet diese mit `@IBOutlet`. Implementiert außerdem eine mit `IBAction` gekennzeichnete Methode, die aufgerufen werden soll, wenn der Benutzer den `UIButton` betätigt:

```
1 import UIKit
2
3 class ViewController: UIViewController {
4
5     @IBOutlet var nameTextField: UITextField!
6     @IBOutlet var greetingLabel: UILabel!
7
8     @IBAction func greetingButtonPressed(sender: UIButton) {
9         println("Hello World!")
10    }
11
12 }
13 @end
```

Beachtet, dass IBOutlets im Allgemeinen als *Implicitly Unwrapped Optionals* deklariert

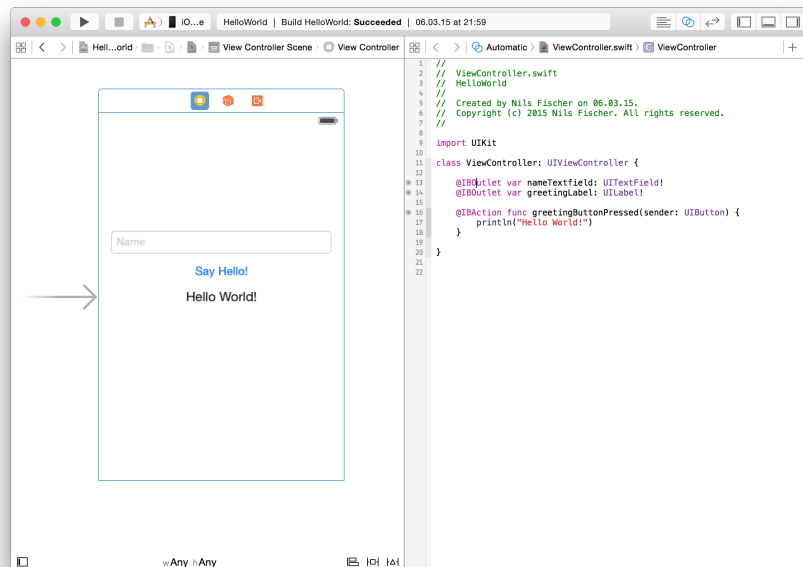


Abbildung 2.8: Mithilfe des Assistants können Interface-BUILDER und Code nebeneinander angezeigt werden.

- werden, da ihr Wert bei der Initialisierung des View Controllers zwar noch nicht existiert, anschließend jedoch unter Verwendung des Storyboards zuverlässig zugewiesen wird. So muss zwar unbedingt aufgepasst werden, dass die IBOutlet Verbindung im Storyboard hergestellt ist und nicht auf das Attribut zugegriffen wird, bevor dieses geladen wurde, doch es ist dann nicht jedes mal notwendig, das Optional zu entpacken.
10. Nun zieht mit gedrückter `ctrl`-Taste eine Linie von dem Textfeld und dem Label im Interface Builder auf das jeweilige Attribut im Code. Die Codezeile wird dabei blau hinterlegt. Zieht außerdem genauso eine Linie von dem Button auf die zuvor definierte Methode. Im Connection Inspector könnt ihr die IBOutlets und IBActions eines ausgewählten Objekts betrachten und wieder entfernen. Dieser Prozess ist im Skript noch detaillierter beschrieben.
 11. Versucht nun einen *Build & Run*. Betätigt ihr den Button, wird die Methode ausgeführt und der Gruß "Hello World!" in der Konsole ausgegeben!
 12. Um die App nun alltagstauglich zu gestalten, muss dieser Gruß natürlich personalisiert und auf dem Bildschirm angezeigt werden. Dazu verwenden wir das Attribut text der Klassen `UITextField` und `UILabel`:

```

1 @IBAction func greetingButtonPressed(sender: UIButton) {
2     self.greetingLabel.text = "Hello \(self.nameTextField.text)!"
3 }

```

Nach einem *Build & Run* erhalten wir unser erstes interaktives Interface, in dem ihr im Textfeld einen Namen eintippen könnt und persönlich begrüßt werdet (s. S. 18, Abb. 2.9)!

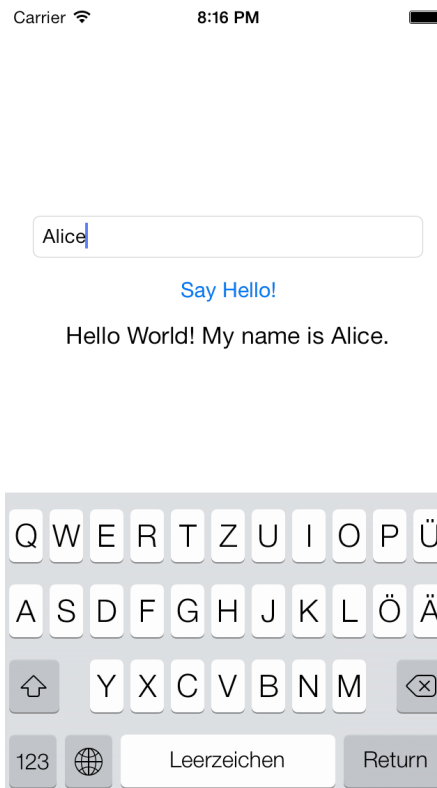


Abbildung 2.9: Drücken wir auf den Button, werden wir persönlich begrüßt. Sehr praktisch!

Übungsaufgaben

5. Simple UI [2 P.]

Erstellt ein neues Projekt und schreibt eine App mit einigen Interfaceelementen, die etwas sinnvolles tut.

Implementiert eines der folgenden Beispiele oder eine eigene Idee. Ich freue mich auf kreative Apps!

Counter Auf dem Bildschirm ist ein Label zu sehen, das den Wert eines Attributs `var count: Int` anzeigt, wenn eine Methode `updateLabel` aufgerufen wird. Buttons mit den Titeln `" +1 "`, `" -1 "` und `"Reset"` ändern den Wert dieses Attributs entsprechend und rufen die `updateLabel`-Methode auf.

BMI Nach Eingabe von Gewicht m und Größe l wird der Body-Mass-Index^[4] $BMI = m/l^2$ berechnet und angezeigt. Als Erweiterung kann die altersabhängige Einordnung in die Gewichtskategorien angezeigt werden.

⁴<http://de.wikipedia.org/wiki/Body-Mass-Index>

RGB In drei Textfelder kann jeweils ein Wert zwischen 0 und 255 für die Rot-, Grün- oder Blau-Komponenten eingegeben werden. Ein Button setzt die Hintergrundfarbe `self.view.backgroundColor` entsprechend und ein weiterer Button generiert eine zufällige Hintergrundfarbe. Ihr könnt noch einen `UISwitch` hinzufügen, der einen Timer ein- und ausschaltet und damit die Hintergrundfarbe bei jedem Timerintervall zufällig wechselt (s. Hinweis).

Hinweise:

- Achtet darauf, dass ihr bei/nach der Projekterstellung in der Targetkonfiguration die Bundle ID `"de.uni-hd.<deinname>.<productname>"` mit `"<productname>"` z.B. `"Counter"`, `"BMI"` oder `"RGB"` eingestellt und unser Developer Team ausgewählt habt, damit die Ausführung der App auf euren eigenen Geräten funktioniert!
- Die Klasse `NSString` aus Apple's Foundation Framework besitzt Instanzmethoden wie `floatValue` zur Umwandlung von Text in Zahlenwerte. `String` und `NSString` sind direkt ineinander überführbar:

```
1 let s = "0.1"
2 let f: Float = (s as NSString).floatValue
```

Eleganter ist die Verwendung eines `NSNumberFormatter`. Solche Formatter, wie auch der `NSDateFormatter`, berücksichtigen bspw. Gerätesprache und länderspezifische Einstellungen und sollten stets verwendet werden, wenn Benutzereingaben interpretiert oder Ausgaben generiert werden:

```
1 let decimalNumberFormatter = NSNumberFormatter()
2 decimalNumberFormatter.numberStyle = .DecimalStyle
3 let s: String = "0.1"
4 let f: Float? =
    ↪ decimalNumberFormatter.numberFromString(s)?.floatValue
```

- Der Initializer `UIColor(red:green:blue:alpha:)` von `UIColor` akzeptiert jeweils Werte zwischen 0 und 1.
- Die Funktion `arc4random_uniform(n)` gibt eine Pseudozufallszahl x mit $0 \leq x < n$ aus.
- Wenn ein `UISwitch` betätigt wird, sendet dieser ein Event `UIControlEvent.ValueChanged`, so wie ein `UIButton` das Event `UIControlEvent.TouchUpInside` sendet. Dieses Event kann genauso mit einer `IBAction` verbunden werden. Mit einem Attribut `var randomTimer: NSTimer?` können wir dann die Methode für das zufällige Wechseln der Hintergrundfarbe implementieren:

```
1 @IBAction func switchValueChanged(sender: UISwitch) {
2     if sender.on {
```

```
3         self.randomTimer =  
           ↳ NSTimer.scheduledTimerWithTimeInterval(0.15, target:  
           ↳ self, selector: "randomButtonPressed:", userInfo: nil,  
           ↳ repeats: true)  
4     } else {  
5         self.randomTimer?.invalidate()  
6         self.randomTimer = nil  
7     }  
8 }
```

Somit wird periodisch die Methode `randomButtonPressed:` aufgerufen, die natürlich implementiert sein muss.