

Universität Heidelberg

Sommersemester 2016

Softwareentwicklung für iOS

App Katalog

Nils Fischer

Aktualisiert am 24. April 2016

Kursdetails und begleitende Materialien auf der Vorlesungswebseite:

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

Inhaltsverzeichnis

1	Einleitung	3
1.1	Über dieses Dokument	3
1.2	Workflow mit Git	3
1.2.1	Ein Repository forken, klonen und bearbeiten	4
1.2.2	Eine Aufgabe per Pull-Request einreichen	5
2	Hello World	6
2.1	"Hello World!" auf Simulator und Gerät	6
2.2	Graphisches "Hello World!"	7
3	A Swift Tour	15

Kapitel 1

Einleitung

1.1 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.

1.2 Workflow mit Git

Wir arbeiten in diesem Kurs mit der Versionskontroll-Software Git und der Softwareentwicklungs-Plattform GitHub [²]. Mit diesen Werkzeugen kann ich euch Beispielprojekte und Aufgaben bereitstellen, die ihr bearbeiten und mir für Kommentare wieder zur Verfügung stellen könnt. Gleichzeitig lernt ihr dabei direkt den Umgang mit zwei der wichtigsten Werkzeuge in der modernen Softwareentwicklung.

Mit Git können wir Änderungen an einem Projekt, oder *Repository*, in regelmäßigen Abständen in *Commits* speichern. Dann können wir jederzeit zu vorherigen Commits zurückkehren und Änderungen vergleichen. Wer die Speicherpunkte bei Super Mario kennt weiß so etwas zu schätzen.

Außerdem ermöglicht uns Git mit anderen Entwicklern zusammenzuarbeiten. Dazu können wir das Repository auf einem Server wie GitHub bereitstellen. Speichert ein anderer Entwickler Commits in dem Repository, können wir dessen Änderungen mit einem *Merge* mit unseren zusammenführen und dabei gegebenenfalls Konflikte beheben. So wird an Softwareprojekten weltweit zusammengearbeitet.

¹<http://ios-dev-kurs.github.io/>

²<https://github.com/>

Da Git ein Kommandozeilenprogramm ist, bedarf es sicherlich einer Eingewöhnung. Wenn ihr noch wenig Erfahrung im Umgang mit der Kommandozeile habt könnt ihr zum Einstieg die GitHub Desktop App [3] verwenden, mit der ihr Git über eine graphische Oberfläche bedienen könnt.

1.2.1 Ein Repository forken, klonen und bearbeiten

Ich stelle Beispielprojekte und Aufgaben in Repositories wie diesem [4] bereit. Verfahrt wie folgt, um es zum Bearbeiten herunterzuladen:

1. Erstellt einen GitHub Account [5], wenn ihr noch keinen habt. Ladet euch die GitHub Desktop App herunter, wenn ihr eine graphische Oberfläche der Kommandozeile vorzieht.
2. Ihr habt nur Lese-Zugriff auf mein Repository. Ihr müsst daher erst einen *Fork* [6] des Repositories erstellen und es damit auf euren Account kopieren. Klickt dazu einfach auf den *Fork* Button auf der Repository-Seite.
3. Euren Fork könnt ihr nun nach Belieben bearbeiten. In diesem Beispiel ist das Fork-Repository unter der URL <https://github.com/dein-username/helloworld> verfügbar. Ihr könnt die Kommandozeile oder die GitHub Desktop App verwenden um das Repository herunterzuladen, oder zu *klonen* [7]. Im Terminal lautet der Befehl dazu:

```
1 git clone https://github.com/dein-username/helloworld
```

4. Nun könnt ihr an dem Projekt arbeiten. Mit folgendem Befehl könnt ihr jederzeit überprüfen, welche Dateien sich geändert haben:

```
1 git status
```

5. Speichert in regelmäßigen Abständen *Commits* [8]. Jeder Arbeitsschritt sollte durch einen Commit repräsentiert werden. Achtet darauf, dass das Projekt bei jedem Commit funktionsfähig ist. In der Kommandozeile erstellt ihr einen Commit wie folgt:

```
1 # Status überprüfen
2 git status
3 # Alle Änderungen dem nächsten Commit hinzufügen
```

³<https://desktop.github.com>

⁴<https://github.com/ios-dev-kurs/helloworld>

⁵<https://github.com/join>

⁶<https://guides.github.com/activities/forking/>

⁷<http://gitref.org/creating/#clone>

⁸<http://gitref.org/basic/#commit>

```
4 git add --all
5 # Commit durchführen
6 git commit -m "Kurze Beschreibung der Änderungen"
```

6. Ihr könnt euer lokales Repository jederzeit mit eurem Repository auf GitHub abgleichen. Daher könnt ihr auch problemlos auf verschiedenen Rechnern an einem Projekt arbeiten. Führt einen *Push* ^[9] oder *Pull* ^[10] in der Kommandozeile aus, oder klickt den *Sync* Button in der GitHub Desktop App:

```
1 # Fortschritt auf GitHub veröffentlichen
2 git push
3 # Änderungen von GitHub herunterladen
4 git pull
```

1.2.2 Eine Aufgabe per Pull-Request einreichen

Habt ihr eine Aufgabe fertig und möchtet Sie einreichen, oder wenn ihr Hilfe benötigt, erstellt eine *Pull-Request* ^[11]. Damit erhalte ich eine Benachrichtigung mit den Änderungen eures Forks im Vergleich zu meinem Original-Repository. Geht wie folgt vor:

1. Speichert eure Änderungen in einem Commit und veröffentlicht sie auf GitHub, wenn ihr es noch nicht getan habt.
2. Klickt auf der Repository-Seite den Button *New Pull Request*, überprüft die Änderungen und klickt dann *Create Pull Request*.
3. Gebt der Pull-Request einen Titel und beschreibt kurz die Änderungen. Erwähnt, wenn etwas nicht funktioniert, sodass ich euch helfen kann. Klickt schließlich auf *Create Pull Request*.

⁹<http://gitref.org/remotes/#push>

¹⁰<http://gitref.org/remotes/#pull>

¹¹<https://help.github.com/articles/creating-a-pull-request/>

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 graphisch vom iOS Simulator und, soweit vorhanden, direkt von unseren eigenen iOS Geräten ausgeben lassen. Dabei stoßen wir auf unseren ersten *Swift* Code und lernen die IDE *Xcode* kennen. Wir arbeiten außerdem direkt mit der Versionskontroll-Software *Git*, einem der Grundbausteine nahezu jedes Softwareprojekts.

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

2.1 "Hello World!" auf Simulator und Gerät



1. Ich habe ein Beispielprojekt bereitgestellt, anhand dessen wir einen ersten Blick auf die Programmierung einer iOS App werfen. Das Ziel ist, das Projekt herunterzuladen, eine erste, einfache App zu schreiben und mir das Ergebnis für etwas *konstruktive Kritik* zur Verfügung zu stellen. Dazu verwenden wir die Versionskontroll-Software *Git* und die Softwareentwicklungs-Plattform *GitHub*, die zu den Werkzeugen gehören, auf denen Softwareprojekte weltweit aufbauen und ohne die moderne Programmierung kaum noch denkbar ist.

Die erste Anweisung lautet:



Klont einen Fork des Repositories <https://github.com/ios-dev-kurs/helloworld>.

Wenn ihr noch mit keinem dieser Begriffe etwas anfangen könnt, seid beruhigt: Wir werden noch so viel mit *Git* und *GitHub* arbeiten, dass ihr am Ende dieses Kurses Experten im Umgang damit seid. Befolgt zunächst einfach die Anweisungen in Unterabschnitt 1.2.1 *Workflow mit Git* bis ihr das Beispielprojekt heruntergeladen habt.

¹https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

2. Öffnet das Xcode-Projekt **"HelloWorld.xcodeproj"** und macht euch mithilfe des Kapitels *Xcode* im Skript mit der Benutzeroberfläche vertraut. In der Toolbar oben findet ihr auf der linken Seite die Steuerelemente des Compilers. Wählt das *Target HelloWorld* und ein Zielsystem, bspw. den *iPhone 6s Simulator*, und klickt die **Build & Run** Schaltfläche. Das Target wird nun kompiliert und generiert ein *Product*, also unsere App, die im Simulator ausgeführt wird. Das Tastenkürzel für *Build & Run* in Xcode ist  + .
3. Besonders spannend ist diese App natürlich noch nicht. Das ändern wir jetzt spektakulär, indem wir unseren **ersten Swift Code** schreiben um eine Ausgabe hinzuzufügen. Wählt die Datei **"AppDelegate.swift"** links im *Project Navigator* aus.
4. Die Methode `application(_:didFinishLaunchingWithOptions:)` wird zu Beginn der Ausführung der App aufgerufen. Ersetzt den Kommentar dort mit einem Gruß zur Ausgabe in der Konsole:

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

5. 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. 8, Abb. 2.1). 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.
6. Wenn ihr ein iOS Gerät dabei habt, verbindet es per USB-Kabel mit eurem Mac und wählt das Gerät 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.

2.2 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 / Interface Builder

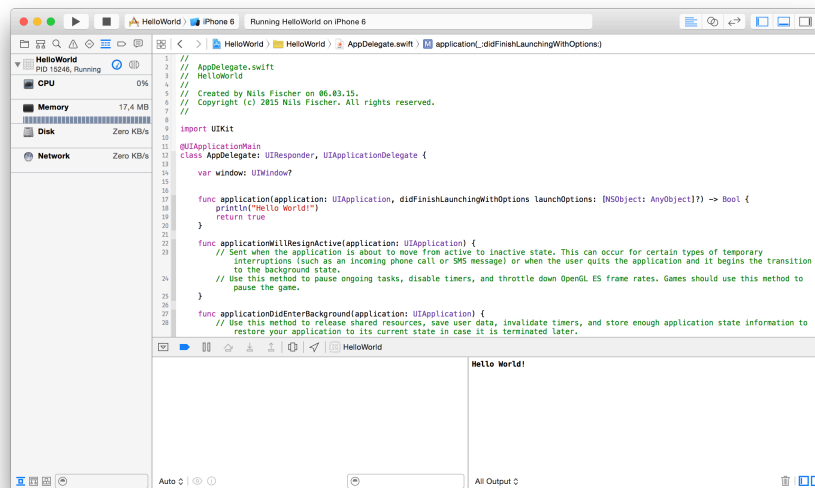




Abbildung 2.1: In der Konsole des Debug-Bereichs werden Ausgaben der laufenden App angezeigt

1. Zur Gestaltung der Benutzeroberfläche oder *User Interface (UI)* verwenden wir ein *Storyboard*. Wählt im Project 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 auf der rechten Seite der Toolbar den Navigator- und Debug-Bereich links und unten aus und den Inspektor rechts ein. Wählt dort außerdem zunächst den Standard-Editor, also die linke der drei Schaltflächen (s. S. 9, Abb. 2.2).
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 rechts ist unten die Object Library zu finden. Wählt den entsprechenden Tab aus, wenn er noch nicht angezeigt wird (s. S. 9, Abb. 2.2).
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!".
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

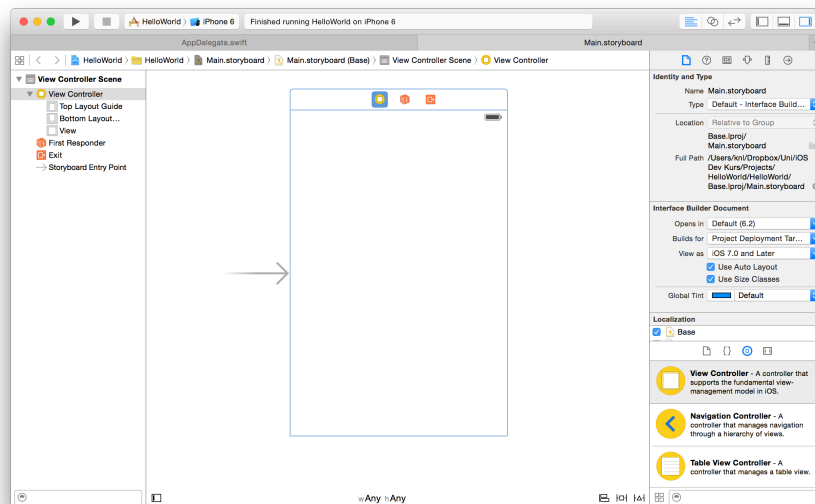



Abbildung 2.2: Für den Interface Builder verwenden wir eine angepasste Fensterkonfiguration mit dem Inspektor anstatt des Navigators

`UITextField`-Objekt auf die Scene und positioniert sie passend (s. S. 10, Abb. 2.3). Mit dem Attributes Inspector könnt ihr dem Button nun den Titel "Say Hello!" geben und für das Text Field einen Placeholder "Name" einstellen.

8. Damit sich das Layout an jede Bildschirmgröße automatisch anpasst, verwenden wir nun *Auto Layout*. Die Schaltflächen dazu findet ihr in der unteren rechten Ecke des Interface Builder Editors. Markiert mit gedrückter -Taste die drei Interfaceelemente und klickt auf das Linke der Symbole mit dem Titel *Stack*, sodass die Elemente in eine *Stack View* eingebettet werden. Dieses praktische Objekt positioniert die enthaltenen Elemente automatisch relativ zueinander. Wählt die Stack View aus und konfiguriert im Attributes Inspector *Axis Vertical*, *Alignment Fill*, *Distribution Fill* und *Spacing 8*. Zusätzlich müssen wir Regeln aufstellen, wie die Stack View auf dem Bildschirm positioniert werden soll. Dazu erstellen wir *Constraints* mit den beiden mittleren der Auto Layout Schaltflächen. Befestigt die Stack View links und rechts am Rand und zentriert sie vertikal.
9. 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 *IBOutlets* und *IBActions*.

Blendet den Inspektor aus und wählt stattdessen den Assistant-Editor (mittlere Schaltfläche) 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. 11, Abb. 2.4).

10. *View Controller* sind Objekte einer Subklasse von `UIViewController`, die jeweils einen Teil der App steuern. Diese sind zentrale Bestandteile einer App, mit de-

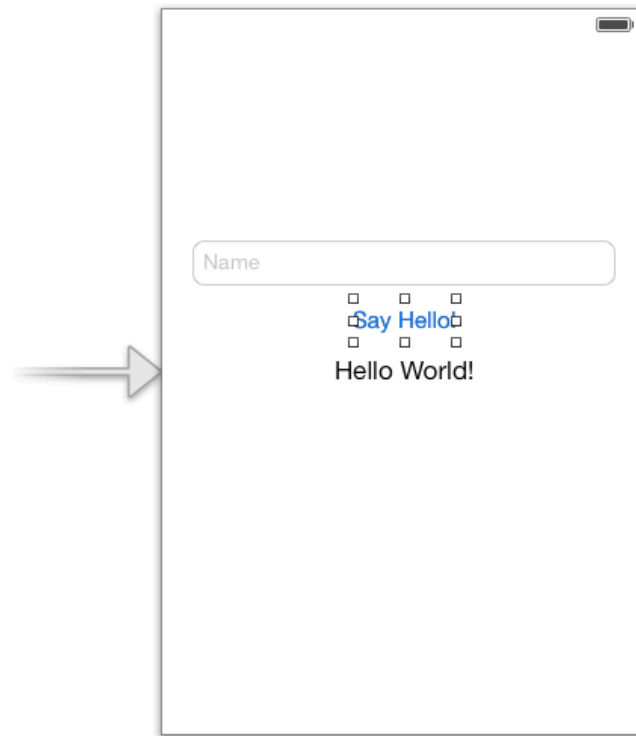


Abbildung 2.3: Mit einem Text Field, einem Button und einem Label erstellen wir ein simples UI

nen wir uns noch detailliert beschäftigen werden. Ein erster View Controller zur Steuerung dieser ersten Ansicht ist im Projekt bereits enthalten.

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          print("Hello World!")
10     }
11
12 }
13 @end

```

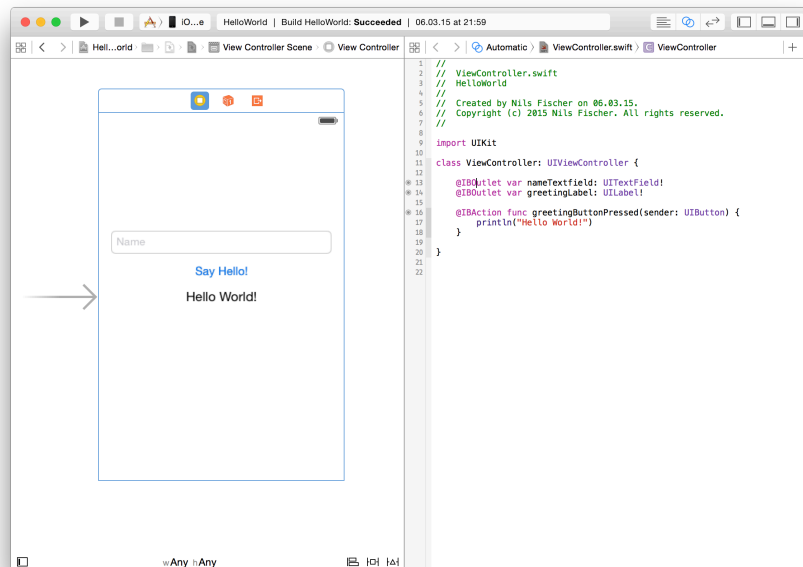


Abbildung 2.4: Mithilfe des Assistants können Interface-BUILDER und Code nebeneinander angezeigt werden.

11. 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 Line von dem Button auf die zuvor definierte Methode. Im Connection Inspector könnt ihr die IBOutlet und IBActions eines ausgewählten Objekts betrachten und wieder entfernen. Dieser Prozess ist im Skript noch detaillierter beschrieben.
12. Versucht nun einen *Build & Run*. Betätigt ihr den Button, wird die Methode ausgeführt und der Gruß "Hello World!" in der Konsole ausgegeben!
13. 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` und zeigen einen personalisierten Gruß an, wenn im Text Field ein Name geschrieben steht:

```

1 @IBAction func greetingButtonPressed(sender: UIButton) {
2     if let name = nameTextField.text where !name.isEmpty {
3         greetingLabel.text = "Hello \(name)!"
4     } else {
5         greetingLabel.text = "Hello World!"
6     }
7 }

```

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. 12, Abb. 2.5)!

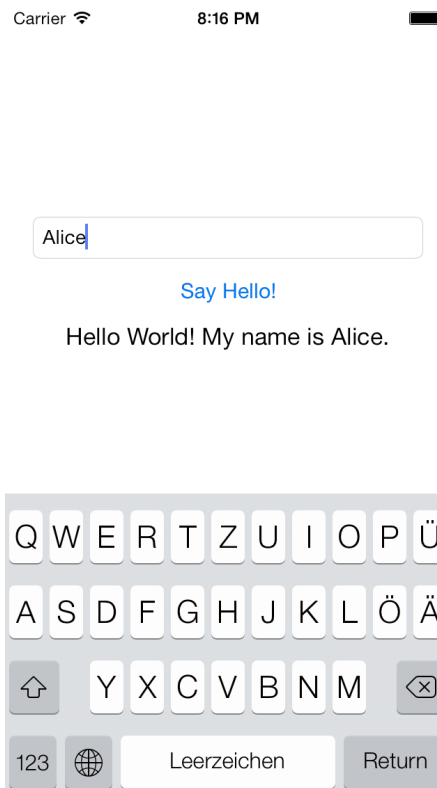


Abbildung 2.5: Drücken wir auf den Button, werden wir persönlich begrüßt. Sehr praktisch!

14. Die App ist fertig! Eure Eltern werden stolz auf euch sein. Gebt mir nun die Gelegenheit, eure Arbeit zu kommentieren. Wir verwendet dazu wieder Git, um die Änderungen, an denen ihr gerade gearbeitet habt, zu speichern, hochzuladen und mir zur Verfügung zu stellen. Befolgt dazu die weiteren Anweisungen in Unterabschnitt 1.2.2, bis ihr mir eine *Pull-Request* geschickt habt.

Übungsaufgaben

1. Simple UI

[2 P.]

Erstellt einen Fork des Repositories <https://github.com/ios-dev-kurs/simpleui> und schreibt eine App mit einigen Interfaceelementen, die etwas sinnvolles tut. Stellt mir das Ergebnis anschließend als *Pull-Request* zur Verfügung.

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-Index² $BMI = m/l^2$ berechnet und angezeigt.

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:

- In der nächsten Vorlesung lernen wir die Objektorientierte Programmierung in Swift systematisch. Orientiert euch für diese Aufgabe an der *HelloWorld* App und versucht die Funktionalität mit den folgenden Hinweisen zu implementieren. Wenn ihr nicht weiter kommt, schickt mir eine Pull-Request mit einem Kommentar und ich helfe euch.
- Das Attribut `text` von `UILabel` und `UITextField` gibt eine *optionale* Zeichenkette `String?` zurück. Ihr werdet euch mit solchen *Optionals* solange herumärgern, bis ihr sie zu schätzen lernt. Verwendet die *Optional Binding* Syntax um das Optional zu entpacken:

```

1  if let name = nameTextField.text {
2      // name existiert und kann verwendet werden
3  } else {
4      // nameTextField.text hat keinen Wert
5  }

```

- Einen `String` könnt ihr schnell in eine ganze Zahl `Int` oder eine Dezimalzahl `Double` umwandeln. Da dies fehlschlagen kann, gibt auch diese Operation einen Optional `Int?` bzw. `Double?` zurück, den wir entpacken müssen:

```

1  // Sei text ein String
2  if let number = Double(text) {
3      // text konnte in eine Zahl number umgewandelt werden
4  }

```

- Definiert ein Attribut wie `var count: Int` mit einem Startwert:

²<http://de.wikipedia.org/wiki/Body-Mass-Index>

```

1  class ViewController: UIViewController {
2
3      var count: Int = 0
4
5      // ...
6  }

```

- Natürlich gibt es die grundlegenden Rechenoperationen `++*/` in Swift. Diese Operationen können mit der Zuweisung zu einer Variablen verbunden werden, um bspw. eine Variable `count` um `1` zu erhöhen:

```

1  count += 1

```

- Eine Farbe wird durch die Klasse `UIColor` repräsentiert. Der *Initializer* `UIColor(red:green:blue:alpha:)` akzeptiert jeweils Werte zwischen 0 und 1:

```

1  let color = UIColor(red: 1, green: 0, blue: 0, alpha: 1) // rot

```

- Die Funktion `arc4random_uniform(n)` gibt eine Pseudozufallszahl x mit $0 \leq x < n$ aus.
- Wenn ein `UISwitch` betätigt wird, kann das Event genauso mit einer `IBAction` verbunden werden wie das eines `UIButton`. Mit einem Attribut `var randomTimer: NSTimer?` können wir dann die Methode für das zufällige Wechseln der Hintergrundfarbe implementieren:

```

1  var randomTimer: NSTimer?
2
3  @IBAction func switchValueChanged(sender: UISwitch) {
4      if sender.on {
5          randomTimer =
6              ↳ NSTimer.scheduledTimerWithTimeInterval(0.15, target:
7                  ↳ self, selector: "randomButtonPressed:", userInfo:
8                  ↳ nil, repeats: true)
9      } else {
10         randomTimer?.invalidate()
11         randomTimer = nil
12     }
13 }

```

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

Kapitel 3

A Swift Tour

Für unsere ersten Apps hat eine gute Portion Intuition für ein wenig Swift Code ausgereicht. Bevor wir tiefer in die App-Programmierung einsteigen beschäftigen wir uns einmal genauer mit der Programmierung in Swift.

Apple bietet mit dem Buch *The Swift Programming Language* eine hervorragende Dokumentation und ein einführendes Kapitel mit den Namen *A Swift Tour*. Das Buch findet ihr sowohl online ^[1] als auch im iBooks Store ^[2] immer in aktueller Version. Auch das letzte Kapitel *Language Reference* ist sehr spannend wenn ihr euch für den detaillierten Aufbau der Sprache und deren Grammatik interessiert.

Löst die Übungsaufgaben mit eurem Wissen aus der Vorlesung und den zugehörigen Materialien auf der Vorlesungwebseite. Zieht zuerst das Buch *The Swift Programming Language* zu Rate wenn ihr nicht weiterkommt. Schickt mir bei weiteren Fragen eine Pull-Request oder einen Xcode Playground per Email.

Xcode Playgrounds eignen sich hervorragend um Swift Code auszuprobieren und um Code zu schreiben der die Infrastruktur einer App nicht erfordert, wie bspw. die Übungsaufgaben *Fibonacci*, *Primzahlen* und *Poker*. Erstellt einen Playground mit  oder .

Die Übungsaufgaben *Fibonacci* und *Primzahlen* sind optional und an Kursteilnehmer gerichtet, die noch wenig oder keine Programmierkenntnisse mitbringen. Auch erfahrene Programmierer können aber anhand dieser Aufgaben die Swift Syntax kennenlernen und versuchen die Aufgaben so *swiftly* wie möglich zu lösen.

Übungsaufgaben

2. Optional: Fibonacci

[+1 P.]

¹https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

²<https://itunes.apple.com/de/book/swift-programming-language/id881256329?mt=11>

Schreibt in einem Xcode Playground einen Algorithmus der alle Folgenglieder $F_n < 1000$ der Fibonaccifolge

$$F_n = F_{n-1} + F_{n-2} \quad (3.1)$$

$$F_1 = 1, F_2 = 2 \quad (3.2)$$

in der Konsole ausgibt.

Hinweis: Versucht's mit einer `while`-Schleife und zwei Variablen für die letzten beiden Folgenglieder, die außerhalb der Schleife definiert wurden. Wer die Aufgabe richtig *swiftly* lösen will kann stattdessen ein `struct`: `FibonacciSequence` schreiben welches das `SequenceType` Protokoll erfüllt.

3. Optional: Primzahlen [+1 P.]

- a) Schreibt eine Funktion `func isPrimeNumber(n: Int) -> Bool` die eine Zahl annimmt und `true` zurückgibt, wenn diese eine Primzahl ist, andernfalls `false`.

Hinweis: Iteriert in einer `for`-Schleife durch alle Zahlen von 2 bis n: `for i in 2... Überspringt den Schleifenschritt mit continue wenn i gleich n ist. Prüft sonst mit dem Modulo-Operator % den Rest der Division n % i der beiden Zahlen. Ist dieser 0 so ist n durch i teilbar und ihr könnt false zurückgeben: return false. Gebt sonst nach dem Durchlauf der Schleife true zurück.`

- b) Schreibt dann eine Funktion `func primeNumbersUpTo(maxNumber: Int) -> [Int]` die alle Primzahlen bis `maxNumber` als Liste `[Int]` (kurz für `Array<Int>`) zurückgibt.

Hinweis: Erstellt zuerst eine leere Liste von `Ints`: `var primeNumbers: [Int] = []`. Iteriert dann in einer `for`-Schleife durch die Zahlen `1..maxNumber` und fügt die Zahl der Liste mit `primeNumbers.append(n)` hinzu wenn sie `isPrimeNumber(n)` erfüllt. Gebt die Liste mit `return primeNumbers` nach Schleifendurchlauf zurück. Richtig *swiftly* könnt ihr die Aufgabe auch in einer Zeile lösen indem ihr die Methode `filter` von `SequenceType` verwendet.

4. Chatter [2 P.]






In dieser Aufgabe schreiben wir zusammen an einer App!

Forkt das Repository <https://github.com/ios-dev-kurs/chatter> und erstellt eine Pull Request um eure Lösung einzureichen oder Fragen zu stellen.

- i. Die *Chatter* App ist in der `"README.md"` Datei beschrieben. Ihr könnt euch die Projektdateien anschauen und die App im Simulator oder auf euren Geräten ausführen und ausprobieren. Wenn euch interessiert, wie die App aufgebaut ist, lest die `"README.md"` und die Kommentare im Code.

- ii. Ihr habt nun sicherlich erkannt worum es in der App geht: Instanzen verschiedener Subklassen von Chatter chatten miteinander. Dabei überschreiben die Subklassen jeweils nur die Implementierung weniger Methoden, die in der Chatter Klasse dokumentiert sind.

Eure Aufgabe ist es nun, eine eigene Subklasse zu schreiben und damit euren Beitrag zu dieser App zu leisten! Ihr könnt einen bekannten Charakter darstellen oder einen Neuen erschaffen. Ich übernehme eure Pull-Request dann in das Original-Repository, sodass euer Charakter mit denen aller anderen Kursteilnehmer chatten kann.

Erstellt dazu mit  +  eine neue **".swift"**-Datei mit dem Namen eures Charakters und platziert sie im Xcode Project Navigator unter   . Orientiert euch an `Yoda.swift` um eure neue Subklasse von Chatter zu implementieren.

- iii. Überschreib die relevanten Methoden in eurer Subklasse wie in der **"README.md"** Datei beschrieben. Hier könnt ihr einfach zufällige Chatnachrichten generieren, oder auch komplexere Mechaniken einbauen, sodass eine etwas natürlichere Konversation zustande kommt.

In eurer eigenen Subklasse könnt ihr dabei beliebig Code schreiben und bspw. Attribute einführen, um den Zustand eures Charakters darzustellen, wenn ihr möchtet. Er oder sie (oder es?) könnte bspw. mit jeder Nachricht wüten oder dergleichen.

- iv. Sichert eure Änderungen regelmäßig in Commits, wenn der Code fehlerfrei kompiliert. Achtet darauf nur Änderungen eurer Subklasse und nur wenn nötig Änderungen in anderen Dateien zu committen. Die **"project.pbxproj"** Datei enthält Informationen zu den Projektdateien - da ihr neue Dateien hinzugefügt habt, gehört diese zum Commit dazu.
- v. Mit eurem Fork des Repositories auf GitHub könnt ihr eure Änderungen jederzeit abgleichen. Bei der Gelegenheit bietet es sich an auch die neuesten Änderungen aus dem Original-Repository herunterzuladen, sodass ihr die neuen Charaktere der anderen Kursteilnehmer erhaltet:

```
1 git pull https://github.com/ios-dev-kurs/chatter.git master
```

Die `pull` Operation versucht, die heruntergeladenen Änderungen mit den lokalen Änderungen zusammenzuführen. Das klappt nicht immer ohne Konflikte. Da jeder von euch dem Projekt eine Datei hinzufügt ändert sich jeweils die **"project.pbxproj"** Datei. Treten Konflikte auf, müsst die die Datei in einem Texteditor öffnen und nach den Konfliktmarkierungen suchen:

```
1 <<<<<<< HEAD:
2 # lokaler Code vor dem Merge
3 =====
```

```

4  # durch den Merge veränderter Code
5  >>>>>>>

```

Behebt den Konflikt indem ihr die Konfliktmarkierungen löscht und den Code dazwischen gegebenenfalls anpasst. Dann könnt ihr den Merge committen:

```

1  git add --all
2  git commit

```

Euer Repository enthält dann sowohl den aktuellen Stand des Original-Repositories, als auch eure Änderungen.

- vi. Wenn ihr mit eurer neuen Chatter Subklasse zufrieden seid schickt mir eine Pull-Request. So werden eure Änderungen in das Original-Repository integriert und tauchen auch bei den anderen Teilnehmern auf, wenn diese das nächste mal einen `git pull` durchführen.

Ich bin gespannt auf eure Implementierungen!

5. Poker

[3 P.]

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

Forkt das Repository <https://github.com/ios-dev-kurs/poker> und erstellt eine Pull Request um eure Lösung einzureichen oder Fragen zu stellen.

- i. Zuerst modellieren wir die Spielkarten. Eine Karte *Card* hat immer eine *Farbe* *Suit* (Karo, Herz, Pik oder Kreuz) und einen *Rang* *Rank* (2 bis 10, Bube, Dame, König oder Ass).

Wir modellieren *Card* als Struct, und *Suit* und *Rank* als Enums. Warum verwenden wir keine Klasse für *Card*? Warum eignet sich ein Enum so hervorragend für *Suit* und *Rank*? Beantwortet diese Fragen kurz stichwortartig in einem Kommentar im Playground.

- ii. Schreibt zwei Enums `enum Suit: Int` und `enum Rank: Int` mit ihren jeweiligen Fällen (`case diamonds` usw.). Bei den Rängen 2 bis 10 schreibt ihr am besten die Zahl aus (`case two` usw.).

Implementiert jeweils eine *Computed Property* `var description: String` in der ihr mithilfe einer `switch`-Abfrage für jeden Fall ein Symbol zurückgibt. **Tipp:** Für Karo, Herz, Pik und Kreuz gibt es Unicode-Symbole³! Außerdem verlangt das Protokoll `CustomStringConvertible` nur das Attribut `description`, schreibt also z.B. `enum Suit: Int, CustomStringConvertible` damit das Symbol in `prints` verwendet wird.

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

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. Lasst auch `Card` das `CustomStringConvertible` Protokoll erfüllen.

- iii. 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 muss `Card` das `Equatable` Protokoll erfüllen. Schreibt extension `Card: Equatable {}` und dann außerhalb:

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

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

- iv. Implementiert nun ein weiteres Enum `enum Ranking: Int` mit den Fällen `case highCard, flush, straightFlush` usw., die ihr bspw. auf Wikipedia^[4] 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`.

⁴http://en.wikipedia.org/wiki/List_of_poker_hands

- v. 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 i in 0...samples {
4     let ranking = PokerHand().ranking
5     if rankingCounts[ranking] == nil {
6         rankingCounts[ranking] = 1
7     } else {
8         rankingCounts[ranking]! += 1
9     }
10 }
11
12 for (ranking, count) in rankingCounts {
13     print("The probability of being dealt a
14         ↪ \(ranking.description) is \(Double(count) /
15         ↪ Double(samples) * 100)%")
16 }
```

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

- vi. **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.