

**Herzlich Willkommen
im iOS-Modul! ☺**

Programmieren fürs iOS

0. Einführung



Herzlich Willkommen 😊

- Programm heute:
 - Einführung
 - Vorstellung
 - Übersicht
 - Lernziele
 - Didaktisches Konzept
 - Organisatorisches
- Swift (Crash Course) [→ eigene Folien]

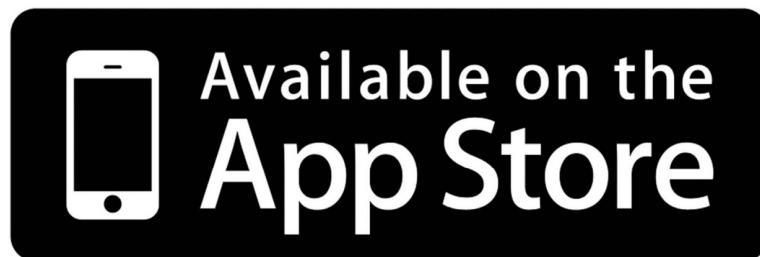
Kurze DEMO ☺

- Xcode + iOS Simulator...



iOS-Programmierung

- Programmiersprachen: Swift, Objective C, (C, C++)
- App läuft nativ auf iOS (!= VM wie z.B. bei Android)
- IDE: Xcode, (AppCode)
- Entwicklerkosten: \$99 (jährlich)
- App-Vertrieb: Apple App-Store, Enterprise Deployment



App Store

Warum iOS?

- Sehr erfolgreiche Smartphone-Plattform!
 - CH: lange iOS vor Android (seit 2014 nicht mehr...)
 - Global: iOS 2. Platz hinter Android
- Prägend für Smartphone-Markt
 - "iPhone = erstes mega-erfolgreiches Smartphone"
 - iPad = erstes mega-erfolgreiches Tablet
 - Erfolgskonzept "App Store"
 - Sehr einfacher App-Kauf & Installation
 - Direkter weltweiter Vertrieb
- ~1'800'000+ Apps im Store (2020)
 - 130'000'000'000+ Downloads (!) (bereits 2016)
 - http://en.wikipedia.org/wiki/App_Store_%28iOS%29

Warum iOS? – technisch(er)...

- Spannende Programmier-Konzepte
 - Entwurfsmuster (MVC, Delegation, Target-Action, ...), Memory-Management
- Andere Programmiersprachen: Swift (& Objective C)
 - Swift != Java, Swift != C#, ...
 - Interessante Sprachkonstrukte & Syntax (Properties, Extentions, Closures, Tupel-Typen, Optionals, nil-Behandlung, ...)
 - SwiftUI, Storyboards, ...

...prima Horizont-Erweiterung für
Java-ProgrammiererInnen! ☺





<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Vorstellungsrunde

Prof. Dr. Ruedi Arnold

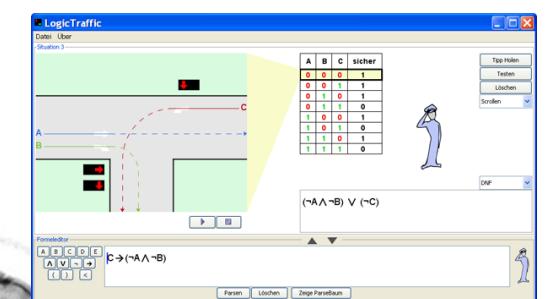
2002 Dipl. Informatik Ing. ETH
Diplomarbeit: "CreatureBrain - Verhalten und Kognition einfacher Kreaturen"

2003 Höheres Lehramt

2007 Dr. sc. ETH
Dissertation: "Interactive Learning Environments for Mathematical Topics"

2008-11 Ergon Informatik AG, Zürich
Entwickler, Projektleiter & Lehrlingsbetreuer
Fokus: Wetter- und Banken-Apps (App-award von Apple für iWeather.ch ☺)

2012- HSLU (Dozent Informatik)

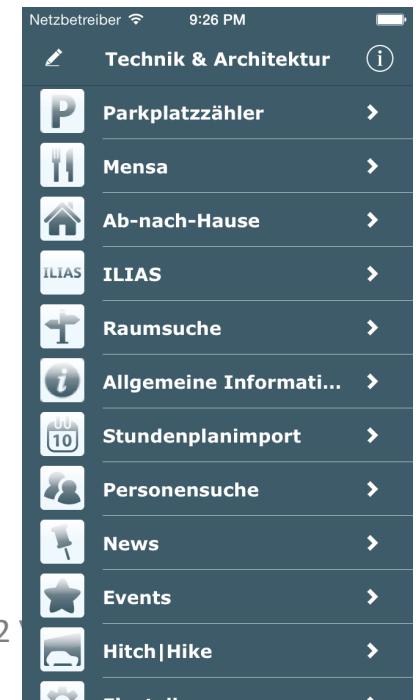
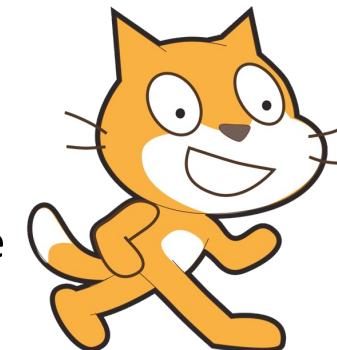


Ruedi Arnold @ HSLU

- Bsc-I-Module
 - PCP: Programming Concepts & Paradigms ☺
 - Mobile Programming & Lab (Android, hybride Apps)
 - iOS (auch in der Weiterbildung)
 - PLAB ("OOP++" im 1. Semester)
- Studiengangleiter "Master Fachdidaktik Medien & Informatik"
- MSE-Advisor, Forschung, ...
- Nachwuchsförderung
 - Programmierworkshops für Jugendliche
 - <http://hslu.ch/scratch>
- iOS- und Android-Apps für HSLU I + T&A
 - Blog: <http://blog.hslu.ch/mobapp/>
 - iOS: <https://apps.apple.com/ch/app/hslu-informatik/id1169402052>
 - Android: https://play.google.com/store/apps/details?id=ch.hslu.mobile_app_dept_i



iOS



**...Informatik ist für mich ein sehr
spannender Beruf, privat habe ich
andere "Hobbys" ☺**



Nicolas Märki

Informatik ist
ein Hobby ☺

ubique 



Erste iOS App

Erstes iPhone

2008

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019

2020

2021

2022



ETH zürich

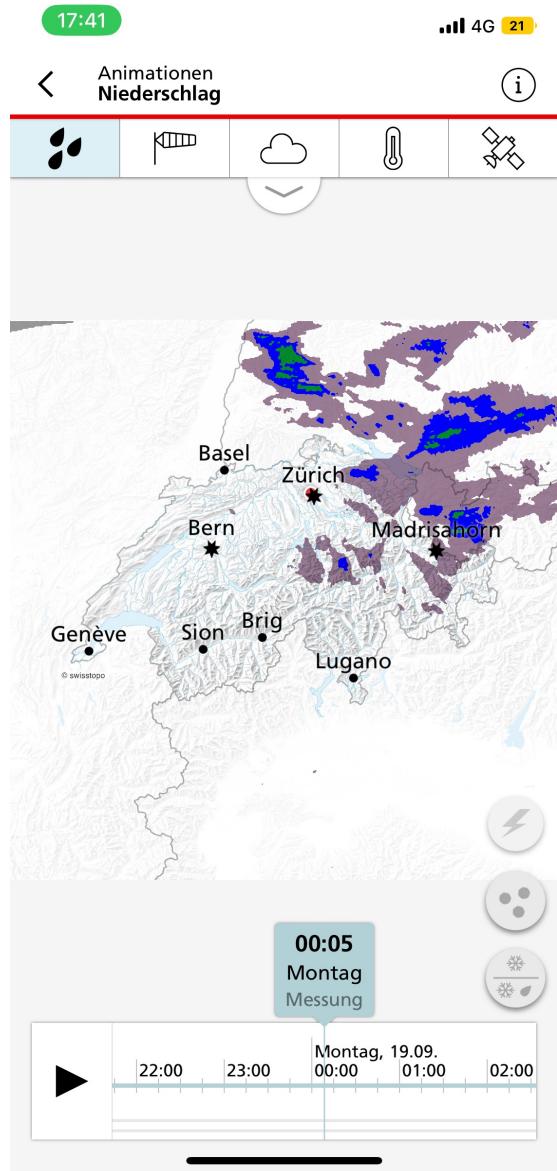
**HOCHSCHULE
LUZERN**



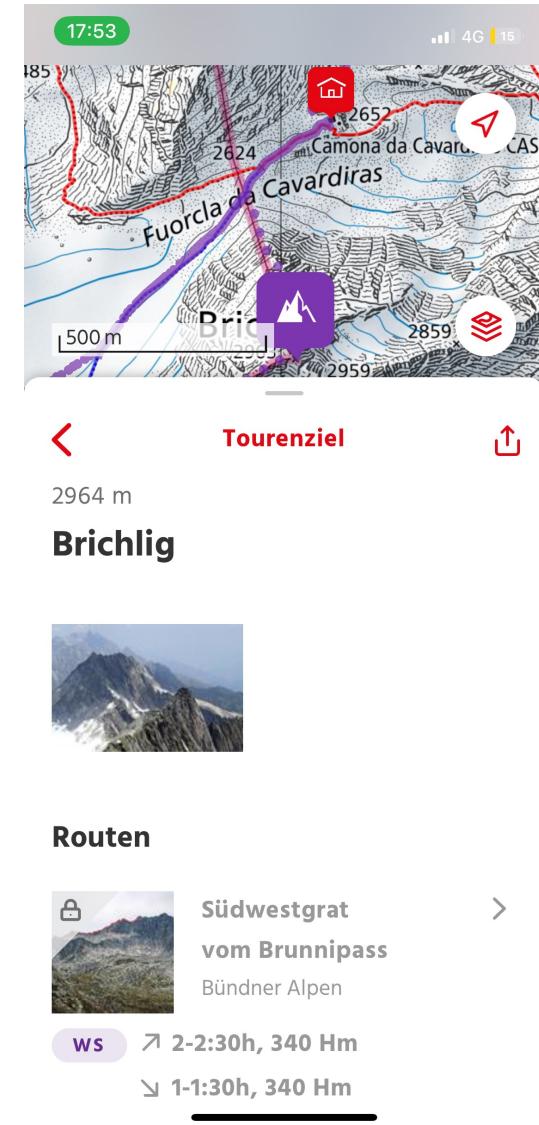
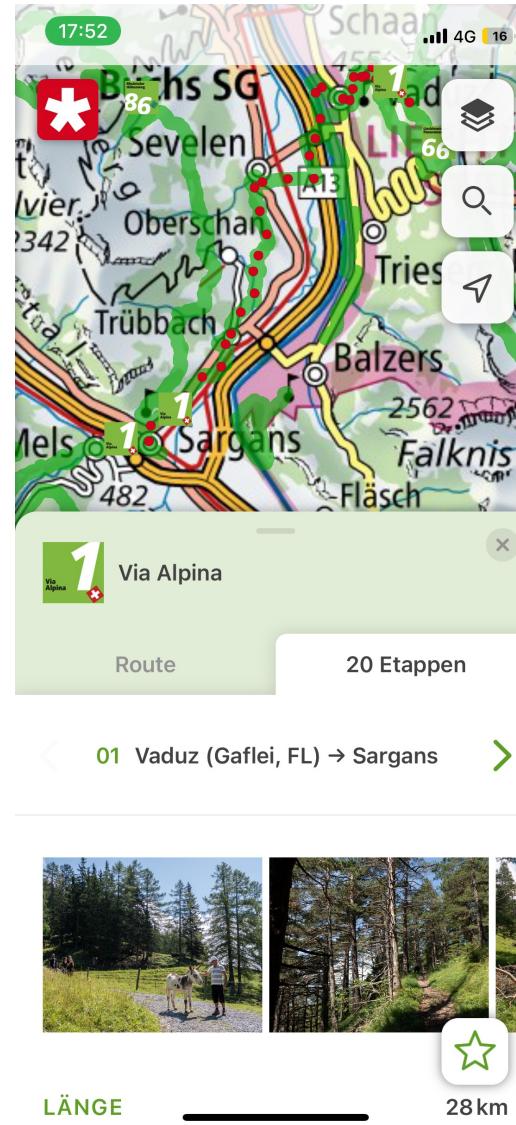
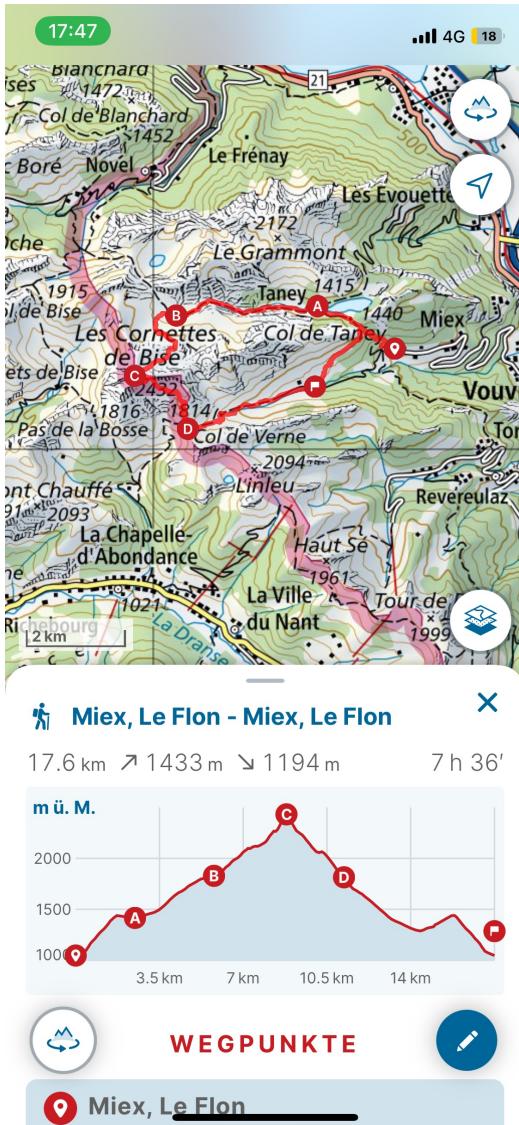
Wir: ~50 kreative Spezialisten (iOS, Android, Web, Backend, Design, UX, Team- & Projektleitung, Administration) in und um Zürich

Unser Motto: Geile Scheiss (oder: Sinnvoll Digitalisieren)

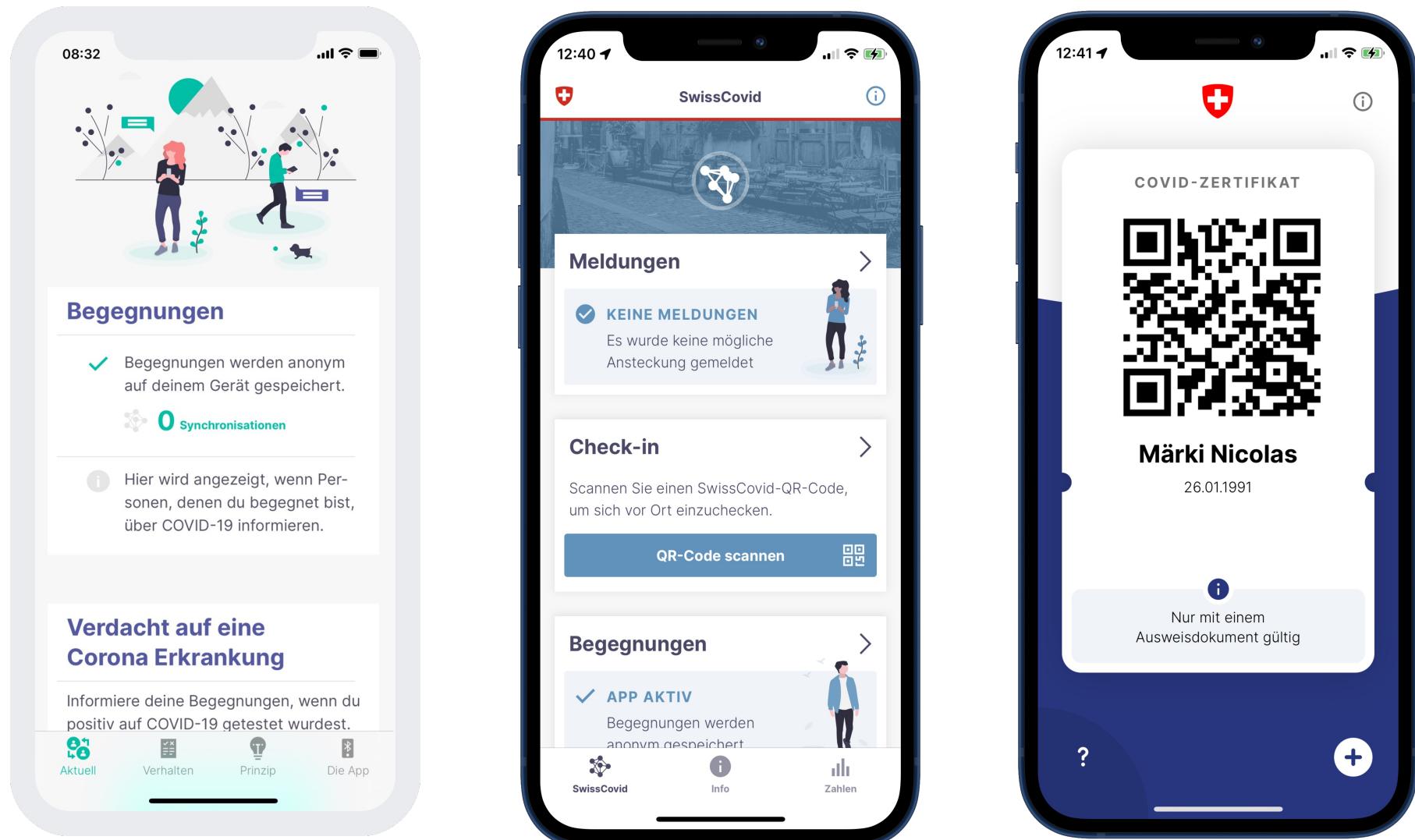
MeteoSchweiz / DWD



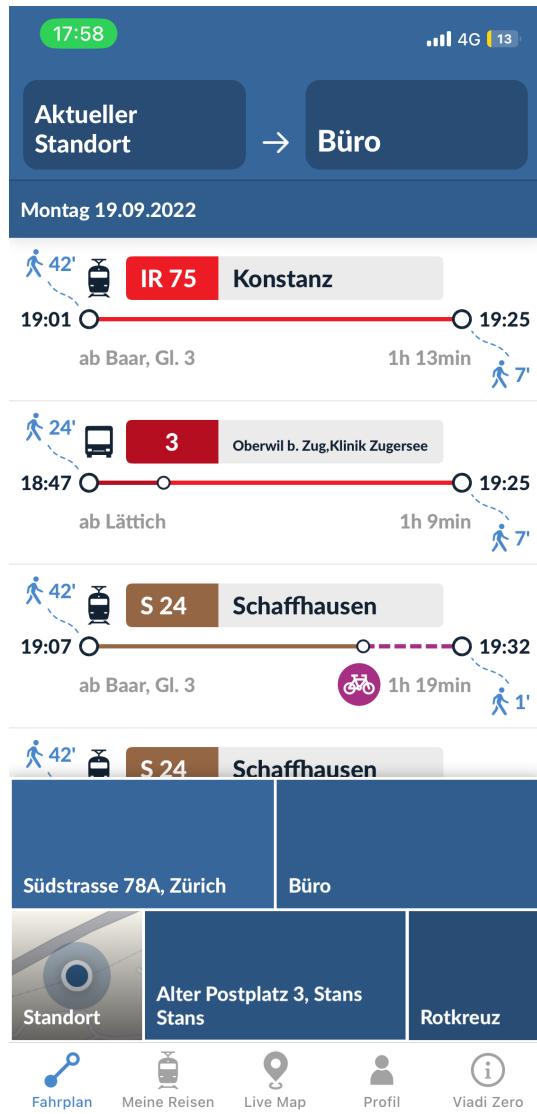
Swisstopo / SchweizMobil / SAC



«Next Step» / SwissCovid / Covid Cert



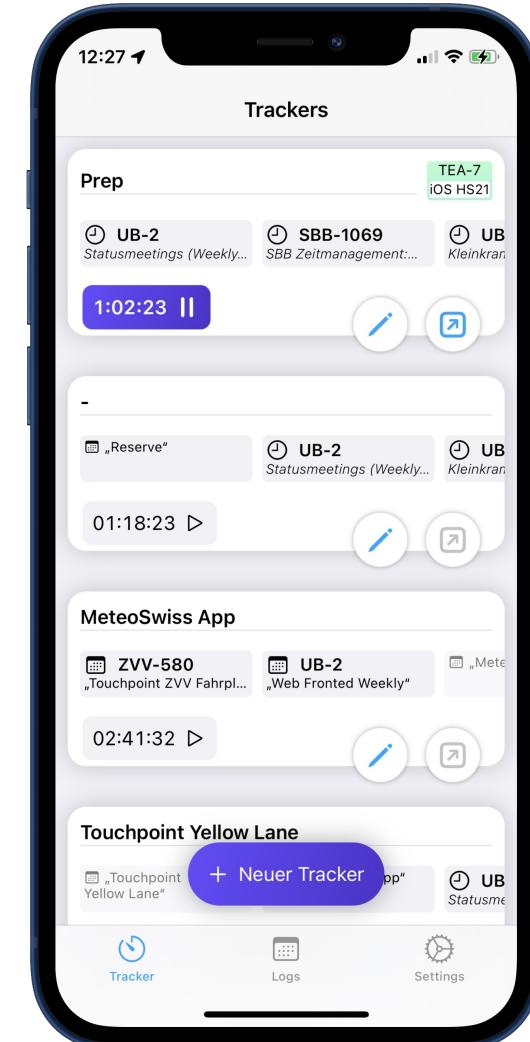
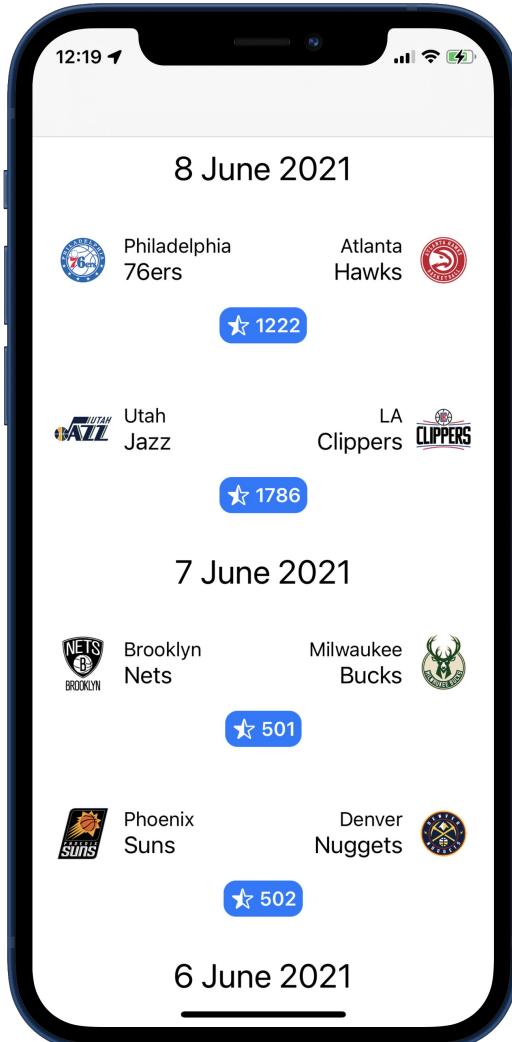
Viadi / SBB



Flesk



“Meine” neusten Apps



...and now:
who are you?



Spontanumfrage!

etc.ch/Sr8w

...and again: Who are you?

- Vorname, Name
Wohnort
- "Programmier-Background"
 - Professionelle Programmiererfahrungen?
 - Jobs/ Firmen/ Technologien/ Plattformen?
- **Motivation für iOS? ☺**



<https://ksassets.timeincuk.net/wp/uploads/sites/55/2018/07/Dave-Grohl-Live-920x584.jpg>



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Modul-Infos

Modul "Programmieren fürs iOS"

- Typ Erweiterung (Intermedidate)
- Credits **3 ECTS (d.h. ~90h)**
- Zeit Montag 18:30 – 20:50
- Ort S1.221
- Dozierende **Nicolas Märki**
(nicolas.maerki@hslu.ch)

Ruedi Arnold (Modulverantwortung)
(ruedi.arnold@hslu.ch)

Modulkurzbeschrieb

Entwicklung eigener Apps, Überblick und Anwendung der Programmiersprache Swift, Grundbausteine in der iOS-Programmarchitektur, Anwendung des iOS-SDK
Fokus: eigene Apps entwickeln!

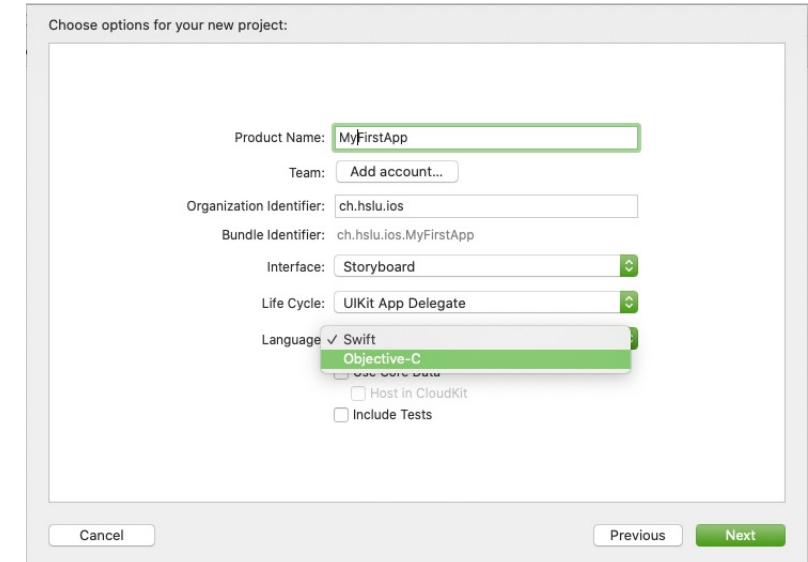
iOS: aktuell zweisprachig!

- ObjC: der alte König...
- Swift: der junge Kronprinz...
 - Zukunftsträchtige "moderne" Sprache
 - Pflicht für SwiftUI
- "Good News": Frameworks, iOS-Architektur, APIs usw. sind für beide Sprachen identisch resp. weitgehend gleich
 - Grosse Teile vom Modul fokussieren darauf ☺



ObjC is not dead

- Aktive iOS Entwicklungssprache
 - Nicht "deprecated" o.ä.
- Mit Swift in beide Richtungen interoperabel
- Grosse Teile von iOS sind/bleiben damit geschrieben
- Es gibt noch sehr viel Legacy Code / Projekte / Doku..
- Apple-Doku und weitere online Quellen (z.B. Stackoverflow) beinhalten weiterhin viel zu ObjC,...
 - Bei iOS-Entwicklung ObjC-Wissen weiterhin nützlich
 - In diesem Modul: Swift (Demos, Übungen, MEP, ...)



Swift vs. Objective C

- Wir fokussieren im Modul auf **Swift** als Entwicklungssprache für iOS
 - Keine Objective C-Kenntnisse notwendig
 - D.h diese ObjC-Syntax bleibt ihnen erspart:

```
[myRef doItWithArg: 7];
```

- Dasselbe in Swift: `myRef.doIt(arg: 7)`

UIKit vs. SwiftUI



- Aktuell zwei UI-Technologien
 - UIKit: der alte **imperative** König
 - ViewControllers, Target-Action-Muster, usw.
 - SwiftUI: der junge **deklarative** Kronprinz...
 - Zukunftsträchtiges "modernes" deklaratives UI-Framework (ähnlich zu Flutter, Jetpack Compose, ...)
 - Aktuell in Transitionsphase...
- Modul behandelt beide, mehr Fokus SwiftUI ☺

Lernziele Fachkompetenzen

- Grundlegende Sprachelemente der Programmiersprachen **Swift** kennen
- Mit der Entwicklungsumgebung **Xcode** arbeiten können
- Selber einfache **Apps** erstellen können.
- Die **iOS-Programmarchitektur** kennen und damit arbeiten können
- Swift-Code verstehen und beurteilen können

Lernziele Methodenkompetenz

- Die Dokumentation von Apple lesen und verstehen können
- Selbstständig eine einfache iOS-App konzipieren, umsetzen und präsentieren können

Lernziele Personalkompetenz

- Sich aktiv und konstruktiv in ein Team einbringen können
- Sich zeitlich organisieren und verbindlich verhalten können

Didaktisches Konzept

- Wochen 1-8: "Vorträge + Übungen"
 - ca. 2 Lektionen geführtes Studium
 - ca. 1 Lektion geführtes Selbststudium
 - Aufgaben lösen
 - Lektüre Folien inkl. Code-Bsp. / online Quellen.
 - Aufgaben vorzeigen & besprechen
- Woche 10-14 "Team-Projekt eigene App"
 - 2er Gruppen, Coachings inkl. Fix-Termine
 - Eigene App-Ideen umsetzen!
 - Abschlusspräsentation im Plenum

Kursinhalt Woche 1-8 (grob)

1. 19.09.: Einführung, Swift Crash Course & Xcode
2. 26.09.: SwiftUI Basics, Layouting, App-Provisioning
3. 03.09.: SwiftUI: States, Bindings, Navigation
4. 10.10.: Kommunikation & Nebenläufigkeit
5. 17.10.: UIKit, ViewControllers, UIKit vs. SwiftUI
6. 24.10.: Fragmentierung, mobile Usability, Widgets
7. 31.10.: Persistenz & Unit-Tests, Property Wrappers
8. 07.11.: Memory-Management, Frameworks

Zulassung und Nachweis

- Zulassung (Testat)
 - Min. 4 Programmier-Übungen (von ca. 8) zufriedenstellend gelöst und Dozierenden präsentiert, Kontrolle hier: <https://bit.ly/list-ios>
 - Aktive Teilnahme an einem individuellen 2er-Team-Projekt, inkl. erfolgreicher Präsentation & Demo der erstellten App im Plenum
- Nachweis
 - 1/3: Bewertung App aus Team-Projekt
 - 2/3: mündliche Modulendprüfung, ca. 15'

Voraussetzung: OOP bestanden

- Bzw. OOP | PLAB | PRG
- Java/C: Basiswissen (Syntax & Semantik)
 - Bedingungen, Schleifen, Variablen, Datentypen, Methoden (Signatur, Parameter, Rückgabewerte), ...
- OOP-Grundkonzepte
 - Klassen und Objekte
 - Instanziierung
 - Vererbung und Polymorphismus

→ Dies ist KEIN Einstiegskurs in (OO-)Programmierung!

Unterlagen

- Folien
- Übungsblätter
- Apple-Doku & Demo-Code
 - Referenzen auf Folien

Ilias: I.BA_IOS.H2201

- [Hier evtl. ganz kurze Ilias-Demo?]
- Forum benutzen!
 - Dozierende beantworten grundsätzlich keine Mails zu technischen/inhaltlichen Fragen zum Modul
 - Gratistipp: Benachrichtigungen einschalten  Aktionen ▾

Benachrichtigung für dieses Forum startenZu Favoriten hinzufügen
- Alles klar?.. ☺

Relevanter Stoff

- Vorlesung: Folien inkl. Ausführungen & Demos
- Übungen
 - Selber lösen!
 - Lösungen später nachvollziehen / erarbeiten
- Links auf Apple-Doku inkl. Demo-Code
- Weitere Quellen (Internet, Bücher, ...)

...in dieser Reihenfolge!

Hinweis zu den Übungen

- Übungspräsentation: 1 Woche nach Ausgabe der Übung
 - z.B.: die Übung der SW04 wird in SW05 präsentiert
- Nicht alle Studierenden zeigen vor, nur zufällige Auswahl
- Falls Übung nicht bzw. ungenügend gelöst bzw. präsentiert wird: Zusätzliche Übung lösen
 - Also falls z.B. die erste Übung ungenügend vorgezeigt wird, ist danach eine Übung mehr, also total 5 (statt 4) Übungen vorzuzeigen
 - Grund: Wir wollen "pokern" nicht attraktiv machen. – Löst also die min. 4 Übungen die ihr vorzeigt zufriedenstellend ☺
- Liste Übungskontrolle: <https://bit.ly/list-ios>

...Übungen präsentieren?

1. Voraussetzung: Ihr habt die aktuelle Übung gelöst und vorzeigbar vorbereitet (Code & Simulator bereit für Demo)
2. Eintragen unter **<https://bit.ly/ios-exercise>** jeweils bis Montag 18:30
3. Dozierende wählen zufällig n Studierende aus, welche im Plenum die Übung vorzeigen
4. Ablauf Vorzeigen: Dozierende wählen die zu präsentierenden (Teil-)Aufgaben aus, Ihr zeigt Eure Lösungen und beantwortet individuelle Fragen
5. Falls Lösungen ok und Fragen zufriedenstellend beantwortet, wird Übung als Teil vom Testat akzeptiert

Vorlesung & Pausen

- ca. 2 Lektionen „geführtes Studium“ (Vorlesung)
- Pausen
 - erste (kurze) Pause 5' nach ca. 45' (d.h. um ca. 19:15)
 - zweite längere Pause flexibel, wenn mit Stoff durch
 - Also z.B. ab 20:00 oder wie lange der zweite Vorlesungsteil eben dauert...
 - Danach Übungsmodus: „open-end“ (bzw. wir mit Fragen durch sind...)

...Hardware?

- Zum iOS-Programmieren braucht's Macs
 - iOS SDK nur für macOS
 - Virtualisiertes macOS ist illegal auf nicht-Apple Hardware...

→ Wir leihen MacBooks aus!

- Leihgebühr: Fr. 150.-
- Macht jemand von diesem Angebot Gebrauch?
 - Möglichst bald Email an empfang.informatik@hslu.ch

...Software?

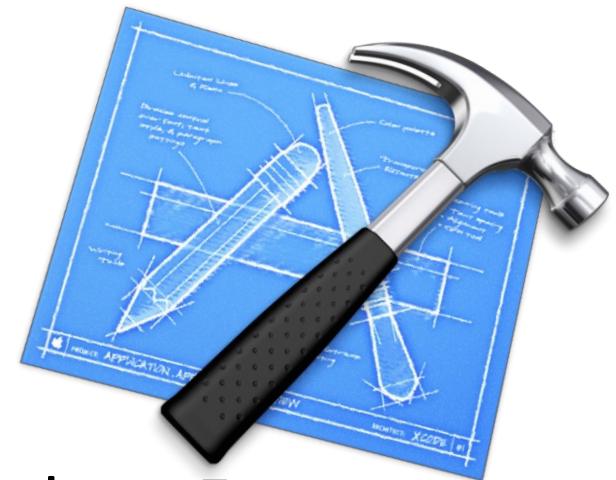
- Zum iOS-Programmieren braucht's Xcode, die von Apple vorgegebene IDE
- Wir verwenden Xcode 14 und Swift 5.7
– Gibt's seit kurzem im macOS App-Store ☺

...Fragen?



Guten Start in's Modul! ☺

- Fragen während der Veranstaltung bitte jederzeit gerne an die Dozierenden!
- Sonstige Fragen bitte ins Ilias-Forum
- **Meine Tipps:** Aktive Teilnahme, mitdenken, Fragen stellen, programmieren, ausprobieren, usw...
→ Programmieren lernt man, indem man es tut! ☺



Programmieren fürs iOS

1. Swift (Crash Course) + Xcode



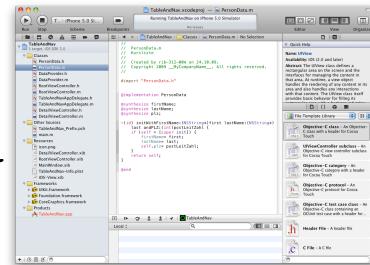
Inhalt

- Swift (Crash Course)
 - Variablen, Konstanten & Typinferenz
 - Datentypen inkl. Optionals & Tupel
 - Properties
 - Kontrollfluss (Bedingung, Schleifen, Auswahl)
 - Funktionen, inkl. benannte Parameter & Default Werte
 - Klassen: Instanziierung & -Initialisierung
- Xcode

"iOS Programmierung"

Tools

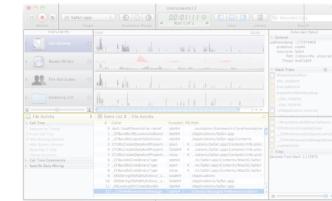
Xcode 14 mit Swift 5.7 😊



Xcode



iOS Simulator



Instruments

Frameworks

Swift Standard Lib

SwiftUI /
UIKit

Languages & Runtime

Swift: myObj.doItWith(aParam)

...zu diesem "Crash Course"

- Rel. viel Stoff in diesem Foliensatz
 - Idee: "Kick-Start: Swift Essentials", damit sie nachher selber loslegen können...
 - Nicht alle Code-Bsp. in Xcode / live Demos
 - Die meisten Dinge werden später im Modul auftauchen
 - Ggf. selber nochmals anschauen & ausprobieren...

→ Denken sie mit, stellen sie Fragen bei Unklarheiten, usw.!

→ So profitieren sie am meisten ☺



A new programming language for iOS and OS X.

- Rel. neue Sprache für macOS und iOS
 - Interoperabel mit Objective-C
 - Unterstützt Cocoa & Cocoa Touch
 - Version 1.0: September 2014
 - d.h.: junge Sprache!
 - OS seit 2.2 (Apache License 2.0) ☺

→ Im Modul: Swift 5.7

- Inbegriffen bei Xcode 14 [beta]

 Swift	Logo
Paradigm	Multi-paradigm: protocol-oriented, object-oriented, functional, imperative, block structured, declarative, concurrent
Designed by	Chris Lattner, Doug Gregor, John McCall, Ted Kremenek, Joe Groff, and Apple Inc. ^[1]
Developer	Apple Inc. and open-source contributors
First appeared	June 2, 2014; 8 years ago ^[2]
Stable release	5.7 ^[3] / 12 September 2022; 1 day ago
Preview release	5.7 branch (5.8 and 6 coming next)
Typing discipline	Static, strong, inferred

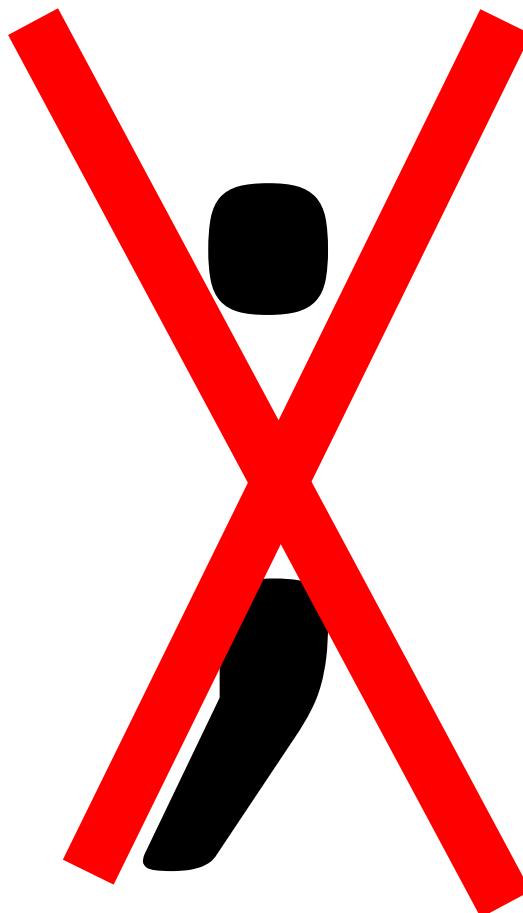
Quellen:

<https://developer.apple.com/swift/resources/>
http://en.wikipedia.org/wiki/Swift_%28programming_language%29

Motivation für Swift

- Apple WWDC 2014: "Objective C ohne C"
 - Sicherer als Objective C
 - Keine Pointer-Typen, Optionals
 - Starke Typisierung, Generics
 - Expressiver als Objective C
 - Ausdrucksstärker, kompakterer Code
 - Weg von der Objective C-Syntax
 - Kein [obj doIt:arg] und @class usw. mehr
- "Moderne", zeitgemäße Sprache

Swift: Semikolon ist optional am Ende von Zeilen ☺



Gute Apple Doku



Swift

The powerful programming language
that is also easy to learn.

- Einstieg: <https://developer.apple.com/swift/>
 - Gute offizielle Quelle: "Language Guide" unter <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>

The screenshot shows the official Swift Language Guide website. The left sidebar includes links for 'THE SWIFT PROGRAMMING LANGUAGE SWIFT 5.1', 'WELCOME TO SWIFT', 'LANGUAGE GUIDE', and 'The Basics'. The main content area features a large title 'The Basics' and a paragraph about Swift's history and its relationship to C and Objective-C. A sidebar on the right lists various language concepts like Constants and Variables, Comments, Semicolons, Integers, Floating-Point Numbers, Type Safety and Type Inference, Numeric Literals, Numeric Type Conversion, Type Aliases, Booleans, Tuples, Optionals, Error Handling, and Assertions and Preconditions. At the bottom, there's a note about variables and their use in Swift.

ON THIS PAGE ×

Constants and Variables
Comments
Semicolons
Integers
Floating-Point Numbers
Type Safety and Type Inference
Numeric Literals
Numeric Type Conversion
Type Aliases
Booleans
Tuples
Optionals
Error Handling
Assertions and Preconditions

Programmierung mit Swift

Swift is a new programming language for iOS, macOS development. Nonetheless, many parts of Swift will be developing in C and Objective-C.

Swift provides its own versions of all fundamental C and for integers, Double and Float for floating-point values String for textual data. Swift also provides powerful collection types, Array, Set, and Dictionary, as des Like C, Swift uses variables to store and refer to values by an identifying name. Swift also



Variablen, Konstanten & Typinferenz

<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Variablen-Deklaration & Typinferenz

- Schlüsselwort `var`
- Typangabe ist optional, wird ggf. abgeleitet vom zugewiesenen Wert
→ Typinferenz (type inference)
- Beispiele:

```
var myInt: Int = 42
var myOtherInt = 43
myInt = 1234
var myString: String = "Hello Swift"
var myDouble = 1.234
```

Hinweis: Semikolon ; generell nicht nötig am Ende von Zeilen (ausser bei mehreren Anweisungen auf einer Zeile)

Konstanten-Deklaration: let

- Schlüsselwort `let`
- Typangabe ist optional, wird ggf. abgeleitet vom zugewiesenen Wert: Typinferenz
- Beispiele:

```
let myConstInt: Int = 77
let myConstDouble = 66.6
let myFixString = "constant"
myFixString = "newValue"           // compile error!!
```

- Randbemerkung: Funktionsargumente sind per Default konstant (d.h. Typ `let`), siehe später

Var. & Konst.: Swift vs. Java

- **Swift:** <name> [: <Typ>]
- **Java (anders rum):** Typ <name>
- **Swift: jede Variable (Konst.) mit var (let) deklariert**
 - Folgende zwei Code-Zeilen kompilieren z.B. nicht:

```
text: String = "hallo"      // compile error!!
name = "Ruedi"              // compile error!!
```
- **Bemerkungen:**
 - Java kennt keine zu var & let analoge Schlüsselworte
 - Java kennt keine derartige Typinferenz (ausser im Kontext von Lambda-Ausdrücken, siehe z.B. PCP-Modul ;-)



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Datentypen

Primitivtypen: Int, Double, Bool

- Swift kennt direkt **keine (native) primitiven Datentypen** (analog zu Java oder C) wie `int`, `float` oder `double`
- Primitive Datentypen sind in Swift in entsprechenden Strukturen (`struct`) verpackt:
`Int`, `Double`, `Float`
 - Hinweis: Strukturen werden in Swift (wie in C) `by value` übergeben (d.h. kopiert)

Bsp.: Int, Double, Bool

- Anwendungsbeispiel zur Illustration (inkl. Fehlermeldung aus der Entwicklungsumgebung Xcode):

```
var i : int = 1           ! Use of undeclared type 'int'  
var j : Int = 2  
var d : double = 3        ! Use of undeclared type 'double'  
var e : Double = 4  
var b : Bool = true|
```

z.B.: Struct Int

Structure

Int

- Beinhaltet
 - Initializers
 - Instanz- und Typ-Methoden
 - adoptierte Protokolle
 - (<https://developer.apple.com/documentation/swift/int>)
- Hinweise
 - Initializiers sind "eine Art" Konstruktoren
 - Structs (und Enums) können Methoden haben
 - Protokoll (Swift/ObjC) = Interface (Java)

Arrays

- In Swift typisiert: Generics
 - Wie in Java
- Beispiel-Typ: String-Array
 - Lange Version: `Array<String>`
 - Kompakte Version: `[String]`
 - Äquivalent, von Apple (und im Modul!) bevorzugt
- Anwendungsbeispiel inkl. Array-Literal:

```
let names : [String] = ["Anna", "Alex"]
```

- Typ-Angabe hier natürlich nicht nötig (Typinferenz!)

Dictionarys

- Enthalten Schlüssel-Werte-Paare
 - Wie Map<K, V> in Java
- Beispiel-Typ: Dictionary mit String & Int
 - Lange Version: Dictionary<String, Int>
 - Kompakte Version: [String: Int]
 - Äquivalent, von Apple (und im Modul!) bevorzugt
- Anwendungsbeispiel inkl. Dictionary-Literal:

```
let ages : [String: Int] = ["Ruedi": 21, "Anna": 23]
```

 - Typ-Angabe hier natürlich auch nicht nötig (Typinferenz!)

Bemerkung zu Collection-Types

- Veränderbarkeit (Mutability): per Default gegeben (Größe & Inhalt)
 - Unveränderbar falls als konstant (let) deklariert
- Neben Arrays & Dictionarys gibt's auch noch Sets (Ungeordnete Menge ohne Duplikate)
 - Inkl. Mengen-Operationen wie `intersect`, `subtract` oder `union` ☺

The Swift Standard Library

The Swift standard library defines a base layer of functionality for writing Swift programs, including:

- Fundamental data types such as `Int`, `Double`, and `String`
- Common data structures such as `Array`, `Dictionary`, and `Set`
- Functions and methods such as `print(_:separator:terminator:)`, `sorted()`, and `abs(_:)`
- Protocols that describe abstractions such as `Collection` and `Equatable`.
- Protocols used to customize operations that are available to all types, such as `CustomDebugStringConvertible` and `CustomReflectable`.
- Protocols used to provide implementations that would otherwise require boilerplate code, such as `OptionSet`.

<https://developer.apple.com/reference/swift>

Bem. zur Standard Library

- Definiert u.a. Basis-Typen wie Int und String und Collection-Typen wie Arrays und Dictionaries
- Die Swift Standard Library ersetzt viele Klassen aus dem Foundation Framework von Objective-C
 - Automatisches "Bridging", z.B. von Array nach NSArray, bzw. Cast möglich mittels as



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Optionals

Datentyp Optional: <DatenTyp>?

- Optional-Typen heisst: es kann sein, dass kein Wert vorhanden ist
- Für Optional-Typen gilt also immer:
 - Es gibt einen Wert und er ist x
 - Es gibt keinen Wert (aber ein "Optional-Objekt")
- **Wichtig: Nicht-optionale Datentypen können nicht nil sein!**
 - d.h. Klassen, structs und enums werden als (weitgehend) gleichwertig betrachtet und müssen immer Wert != nil haben
 - z.B.: `let myInstance : MyClass = nil`
● Nil cannot initialize specified type 'MyClass'

Bsp: Deklaration Int-Optional

- "Konstruktor" `Int(String)` liefert `Int?` zurück,
d.h. einen Optional vom Typ `Int`
 - Sinnvoll: Argument könnte ja keine Zahl sein...
 - Kann also nicht einem `Int` zugewiesen werden:

```
let convNumb : Int = Int("1234")
```

● Value of optional type 'Int?' not unwrapped; ⌘C

- Zuweisung möglich an Var vom Typ "Int-Optional":

```
let convNumb : Int? = Int("1234")
```



Optionals & Unwrapping

- Ausgabe von `let convNumb : Int? = Int("1234")
print(convNumb)`
 - Antwort: `Optional(1234)`
- Q: Zugriff auf Optional-Werte?
- A: Forced-Unwrapping (Operator: `!`)
 - Bsp.: `let convNumb : Int? = Int("1234")
let number : Int = convNumb!`



"Sicheres" Auspacken

- Besser zuerst auf Wert testen beim Zugriff auf Wert von Optional:

```
let convNumb : Int? = Int("1234")
if convNumb != nil {
    let result = convNumb! + 2
    print(result)
}
```

- "Forced Unwrapping": <OptionalTyp>!
 - Holt Wert "heraus", Schlüsselzeichen: !
 - Laufzeitfehler falls Optional keinen Wert hat

Optional Binding: if let

```
if let constantName = someOptional {  
    statements  
}
```

- Temporäre Bindung von einem Optional an eine Variable bzw. Konstante

- z.B.: `let convertedNumber : Int? = Int("1234")
 if let number = convertedNumber {
 print("The number is \(number)")
 } else {
 print("No number available")
 }`

Implicitly Unwrapped Optionals: <Typ>!

- Sind Optionals, die immer einen Wert haben (sollten), Laufzeitfehler falls nicht
 - Können ohne "unwrapping" verwendet werden
 - Zweck: "Bequemlichkeit" (Objective-C & "alte" APIs), siehe später (Viele API-Methoden haben Parameter, die nil sein können)
- Anwendungsbeispiel aus der Swift-Doku von Apple:

```
let possibleString: String? = "An optional string."  
let forcedString: String = possibleString! // requires an exclamation mark  
  
let assumedString: String! = "An implicitly unwrapped optional string."  
let implicitString: String = assumedString // no need for an exclamation mark
```

Hinweis zum Ausrufezeichen

- Achtung: ! hat im Kontext von Optionals also zwei Bedeutungen:
 1. "Forced Unwrap"-Operator von einem Optional:
 - Bsp.: `let value = optional!`
 2. Typendeklaration von einem "implicitly unwrapped" Optional:
 - Bsp.: `var text : String! = nil`

Alternative zu unsicherem Auspacken: Optional Chaining

- Forced Unwrapping (!-Konstrukt) produziert Laufzeitfehler, falls Optional keinen Wert hat
- Alternative: Optional Chaining (Konstrukt **?.**)
 - Idee: Falls Wert von Optional ohne Wert abgefragt wird, kommt `nil` zurück und Anfragen (Property, Methode) darauf liefern wiederum `nil` zurück: "nettes" Verhalten!
 - Wie "Safe Navigation Operator" (`?.`) von Groovy
 - Hinweis: Das war bei Objective-C schon immer so, dass ein Methoden- oder Property-Aufruf auf `nil` keinen Fehler à la `NullPointerException` produziert, mit Optionals und Optional Chaining kann nun dieses Verhalten auch in Swift erreicht werden

Beispiel für Optional Chaining: ?. .

```
class Person {  
    var residence: Residence?          // note: optional type  
}  
  
class Residence {  
    var numberOfRooms = 1  
}
```

- Optional Chaining im Einsatz:

```
let john = Person()  
if let roomCount = john.residence?.numberOfRooms {  
    print("John's residence has \(roomCount) room(s).")  
} else {  
    print("Unable to retrieve the number of rooms.")  
}
```

Nil Coalescing Operator: ??

- Entpackt einen Optional-Typ oder liefert einen Default-Wert zurück, falls der Optional keinen Wert hat
 - Coalescing (EN) = Verschmelzen, Vereinigen
 - Äquivalenter Code:
 - `a ?? b`
 - `a != nil ? a! : b`
 - Analog zum "Elvis Operator" (?:) von Groovy



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Datentyp Tupel

Tupel-Datentypen

- Tupel können beliebig viele Werte von beliebigen Datentypen enthalten
- Beispiel-Deklaration

```
let testTuple = (77, true, "Hi")
```

- Wert-Zugriff mittels Index oder "Tuple Decomposition"

```
var x = testTuple.0 + 1  
var (number, flag, text) = testTuple  
let inc = number + 7  
var (_, justTheFlag, _) = testTuple
```

Index-Zugriff auf Tupel-Werte

Tuple Decomposition

Tuple Decomposition, bei der gewisse Tupel-Werte ignoriert werden (Schlüsselzeichen _)

Tupel: Elemente-Namen & als Rückgabewert von Funktionen

- Tupel-Elemente können benannt sein:

```
let anotherTuple = (id : 66, name : "Ruedi")
print("The id is \(anotherTuple.id)")
```

– Zugriff wie auf Properties, d.h. mittels Dot-Syntax

- Hinweis: Tupel sind beispielsweise praktisch als Rückgabetyp von Funktionen: Funktionen können so beliebig viele Werte zurück liefern! (Siehe später)



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Properties

Properties: Zugriff mit Dot-Syntax

- (Instanz-)Properties assoziieren Werte mit Instanzen einer Klasse, Struktur (`struct`) oder Enumeration (`enum`, nur computed Properties)
 - Analog zu Properties in Objective-C oder C#
- Swift unterscheidet folgende zwei Arten:
 - Stored Properties ("Gespeicherte Werte")
 - Unterart: Lazy stored Properties
 - Computed Properties ("Berechnete Werte")
- Zugriff über Dot-Syntax: `x.property`
 - Analog zu Zugriff auf Instanzvariable in Java

Stored Properties & Lazy

- Stored Property: Variable (`var`) oder Konstante (`let`) ist Teil einer Klasse oder Struktur (`struct`)
 - Benötigt keine spezielle Syntax, kein Schlüsselwort
 - Entspricht Java-Instanzvariablen
 - Einfachster & "typischer" Property-Typ
- Unterart: Lazy Stored Properties, Schlüsselwort `lazy`
 - Wie Stored Property, aber Wert wird erst bei erstem Property-Zugriff ausgewertet und zugewiesen
 - Praktisch wenn Property-Erstellung "teuer" ist und Property evtl. gar nie gebraucht wird

Bsp.: Stored Property & Lazy

- Deklaration in TestClass.swift:

```
class TestClass {  
    var text = "hi"                      // stored property  
    lazy var lazyString = "test"        // lazy stored property  
}
```

- Verwendung: Property-Zugriff mit dot-Syntax

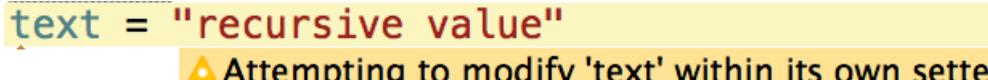
```
var testClass = TestClass()      // create instance  
testClass.text = "bye"          // set new value  
print(testClass.lazyStr)       // first access -> create
```

```
var testClass = TestClass()  
testClass.text = "bye"  
println(testClass.lazyString) < Thread 1: breakpoint 2.1
```



```
▼ L testClass = (SwiftTestApp.TestClass) C  
▶ text = (String) "bye"  
lazyString.storage = (String?) nil ←
```

Computed Properties: Werte "on the fly" berechnen

- Getter- und setter-Methoden werden explizit ausprogrammiert
 - Schlüsselworte `get` + `set`
 - Hinweis: Im `set`-Block darf nicht direkt dieses Computed Property gesetzt werden, da daraus ein nicht-terminierender rekursiver Setter-Aufruf resultieren würde! - Compiler merkt's und warnt:


```
text = "recursive value"
⚠ Attempting to modify 'text' within its own setter
```
- `setter`-Block ist optional
 - Falls nicht vorhanden: Read-Only Computed Property

Beispiel: Computed Property

- Deklaration in TestClass.swift:

```
class TestClass {  
    var message = "hi"                                // stored property (as seen)  
    var text : String {  
        get {  
            return message + " 2"                      // getter implementation  
        }  
        set {  
            message = newValue + " 1"                  // setter implementation  
        }  
    }  
}
```

Achtung: Wir setzen message und nicht text!

newValue = Default Argumentname für den neu zu setzenden Wert (Apple nennt das "Shorthand Setter Declaration")

- Verwendung vom computed Property:

```
var testClass = TestClass()  
testClass.text = "bye"  
print(testClass.text);                                // prints "bye 1 2"
```

Property Observers

- Properties können beobachtet werden
 - Schlüsselworte willSet und didSet
- Anwendungsbeispiel:

```
var value : Int = 7 {
    willSet {
        print("value will be set to \(newValue)")
    }
    didSet {
        print("value was set to \(value)")
    }
}
```

Type Properties (static)

- Die bisher gesehenen (Instanz-)Properties gehören jeweils zu einer Instanz (Analog zu Instanzvariablen, z.B. in Java)
- Es gibt auch "Klassen"-Properties, diese heissen bei Swift **"Type Properties"**
 - Schlüsselwort `static` für "normale" Properties
 - (oder `class` für in Subklassen überschreibbare "computed properties")
 - Allg. Namen "Type-Properties" da auch für enums und structs!
 - Bsp.:

```
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        get {return 42}

    }
    class var overrideableComputedTypeProperty: Int {
        get {return 42}
    }
}
```

Bemerkungen zu Properties

- Properties können Zugriff auf Instanzvariablen kapseln oder Konzept Instanzvariable auch "dynamisch" erweitern (im Fall von computed property)
- Ähnliches Sprachkonstrukt gibt's auch in Objective-C
 - Schlüsselwort dort `@property`
 - Bsp. Deklaration: `@property strong NSString* name;`
 - Unterschied: Instanzvariable hinter einem Property ist bei Objective-C sichtbar, bei Swift nicht
- Ähnliches Konstrukt gibt's auch bei C#
 - ganz ähnliche Syntax für `get { . . . }` und `set { . . . }`



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Kontrollfluss

Bedingungen ohne Klammer

- In Swift gibt's analog zu den meisten anderen imperativen Programmiersprachen u.a. folgende Sprach-Konstrukte zur Steuerung des Kontrollflusses: if, for, while, switch-case
 - Generell sind für die Bedingungen keine runden Klammern notwendig, wie z.B. bei Java
 - Legaler Swift-Code:
- ```
if 4 < a {
 print("smaller")
}
```

# for-in & Range-Operator

- Beispiel:

```
for i in 1...3 {
 print("i = \(i)")
}
```
- Bemerkungen zu obigem Code
  - Verwendet den "range operator" . . .
    - Praktisches Sprach-Konstrukt 😊
  - Ausgabe:  
*i* = 1  
*i* = 2  
*i* = 3

# KEIN "traditionelles" for

- "Old school", C-Style for-Schleifen gibt's in Swift nicht (mehr)



# for-in & Iteration über Arrays

- Code-Beispiel:

```
let names : [String] = ["Anna", "Alex", "Peter"]
for name in names {
 print("Hello, \(name)!")
}
```

- Ausgabe:  
Hello, Anna!  
Hello, Alex!  
Hello, Peter!

# for-in mit Tupeln

- Code-Beispiel mit einem Dictionary:

```
let number0fLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in number0fLegs {
 print("\(animalName)s have \(legCount) legs")
}
```

- Beispiel mit enumerated() auf Array:
  - Hinweis: so gibt's "gratis" index-Variable

```
let names = ["Anna", "Alex", "Luana", "Peter"]
for (index, name) in names.enumerated() {
 print("\(index+1). \(name)")
}
```

# while & repeat-while

- Keine Überraschungen:

- while: `while` `condition` {  
    `statements`  
}

- repeat-while: `repeat` {  
    `statements`  
} `while` `condition`

# switch-case

```
switch some value to consider {
 case value 1:
 respond to value 1
 case value 2,
 value 3:
 respond to value 2 or 3
 default:
 otherwise, do something else
}
```

- Bemerkungen
  - Pro case mehrere Werte möglich
  - Kein impliziertes "Fall-through" (d.h., break i.a. nicht notwendig)

# Control Transfer Statements

- `continue`: nächster Schleifendurchgang
  - auch inkl. Labels (bei verschachtelten Schleifen!)
- `break`: Abbruch bei Schleife oder switch-case
  - auch inkl. Labels (bei Verschachtelung!)
- `fallthrough`: "Durchfallen" von zum nächsten case (erreicht Verhalten von Standard-C-switch-case)
- `return`: Funktionen (siehe später)
- `throw`: Error Handling (siehe später)



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

# Funktionen

# Funktionsdeklaration in Swift

- Muster der Grundsyntax:

```
func <name>(<arg>: <type>) -> <returnType> { ... }
```

- Schlüsselwort `func`
- Argumente nach Funktionsname in `( )`
  - Mehrere Argumente durch Komma getrennt
- Rückgabetyp nach `->`
  - Ist optional, d.h. kein `-> <returnType>` falls Funktion nichts zurückliefert

# Funktion ohne Arg & Rückgabe

- Methode: Impl. in Klasse `SwiftTest`

```
func printHello() {
 println("Hello Swift")
}
```

- Aufruf aus Klasse `SwiftTest`

```
printHello()
var myInstance = SwiftTest()
testSwift.printHello()
```

– Hinweis: Syntax Funktionsaufruf analog zu Java

# Beispielfunktion inkl. Aufruf

- Ein Argument, ein Rückgabetyp (in Klasse SwiftTest)

```
func sayHelloTo(name: String) -> String {
 let greeting = "Hello, " + name
 return greeting
}
```

- Aufruf aus Klasse SwiftTest

```
var result = sayHelloTo(name: "Swift")
var testSwift = SwiftTest()
var value = testSwift.sayHelloTo(name: "Swift")
```

# Funktion mit mehreren Rückgabewerten ☺

- Mit Tupels mehrere Werte zurück liefern:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
 var currentMin = array[0]
 var currentMax = array[0]
 for value in array[1.. if value < currentMin {
 currentMin = value
 } else if value > currentMax {
 currentMax = value
 }
 }
 return (currentMin, currentMax)
}
```

- Bsp.:  

```
let bounds = minMax(array: [22, 4, 6, 42, 17, 1, 34])
print("min is \(bounds.min) and max is \(bounds.max)")
```

# Param: per Default konstant (let)

- "Praktische" Eigenschaft von Swift: Parameter sind per Default konstant (let)
  - Guter Stil in Java?
  - Illustrationsbeispiel:

```
func doStuff(a: Int) {
 a = 7
}
! Cannot assign to value: 'a' is a 'let' constant
```

- Falls ein Parameter doch veränderbar sein soll: mit lokaler var überschreiben
  - Illustrationsbeispiel:

```
func doStuff(a: Int) {
 var a = a // overwrite a
 a = 7 // now ok
```

# Funktionen: Argument Labels & Parameter Names

- Argumente von Funktionen können zwei verschiedene Bezeichnungen haben
  - "extern" (beim Aufruf): Argument Label
  - "intern" (im Funktions-Body): Parameter Name
- Beispiel:

```
func someFunc(extArgumentLabel localParamName: Int)
```

- Funktionsaufruf: someFunc(**extArgumentLabel**: 77)
- Im Body der Funktion ist der übergebene Wert 77 dann in der Konstanten **localParamName** verfügbar

# Aufruf mit Argument Label

- Aufruf mit Argument-Label ist zwingend, sonst gibt's Kompilierfehler

```
func someFunc(extArgumentLabel localParamName: Int)
```

- Bsp. Anwendung (Funktionsaufruf):

```
someFunc(extArgumentLabel: 7) // ok
someFunc(7) // compile error
```

- Hinweis: Die (bei der Deklaration festgelegte) relative Reihenfolge der Argumente muss beim Funktionsaufruf zwingend beibehalten werden

# Bem. zu Argument & Parameter

- Wenn "Argument Label" und "Parameter Name" gleich sind, reicht einmalige Angabe
  - Siehe z.B. vorhin bei `sayHelloTo(name: String)`, da heissen Argument & Parameter name
- Falls Argument Label unerwünscht: `_` (underscore)!

```
func add(_ a: Int, _ b: Int) -> Int {
 return a + b
}
// ...
let c = add(1, 2) // ok without arg labels
```

# Default Parameterwerte

- Parameter können Default-Werte haben 😊
  - Beispielfunktion:

```
func join(s1: String, s2: String, joiner: String = " ") -> String {
 return s1 + joiner + s2
}
```

- Funktionsaufrufe:

```
join(s1: "HSLU", s2: "I", joiner: "-") // provide all 3 args
join(s1: "HSLU", s2: "I") // use default value
join(s1: "HSLU") // compile error
```

# Bem. zu Default-Werten

- Praktisches Konstrukt: Beim Aufruf müssen nur wirklich essentielle Parameter mitgegeben werden, für die andern können Default-Werte gesetzt werden
  - Methoden müssen nicht überladen werden und sich gegenseitig aufrufen
- Praktisch z.B. zur Objekt-Initialisierung
  - Behebt z.B. die Java-"Unschönheit": viele ähnliche Konstruktoren mit je 1 Argument mehr u.ä.



# Klassen: Instanziierung & Initialisierung

# Klassen: Dekl. und Instanziierung

```
class MyClass : MyBaseClass {
 var name = "Test" → Stored Property mit initialem Wert
 var id: Int

 init(id: Int) {
 self.id = id
 }
}
```

Stored Property ohne initialen Wert – muss in einer Initializer-Methode gesetzt werden!

"Initializer"-Methode, einfachste Form mit Schlüsselwort `init`

- Instanziierung eines MyClass-Objekts:

```
var myClass = MyClass(id: 7)
```

- Ruft "automatisch" die `init`-Methode von `MyClass` auf
  - `init`: Initialisierungsmethode, an sich normale Methode
    - ähnlich zu Java-Konstruktor, jedoch ohne spezielle Syntax

# Und noch mehr...

- Neben dem gesehenen bietet / unterstützt Swift noch viel mehr Sprachkonzepte und – konstrukte, z.B.:
  - Closures ("Funktionstyp", siehe später)
  - Zugriffskontrolle: private, internal, public
    - ähnlich wie in Java, "internal = Projekt"
  - Funktionale Programmierung (analog zu Stream@Java8): filter, map & reduce
  - Geschachtelte Funktionen

# ...und viel mehr!

- Subscripts für Klassen, Enums & Structs
  - Ermöglicht Index-Zugriff auf beliebige Objekte
- Extensions
  - Erweitern Klassen dynamisch um neue Methoden (ähnlich zu Kategorien in Objective-C)
- Automatic Reference Counting (ARC)
  - Schlüsselwörter `weak` und `unowned`
  - Übernommen von Objective-C
- Casting, Schlüsselwort: `as`
- Protokolle (= Interface in Java)
  - Übernommen von Objective-C
- ...

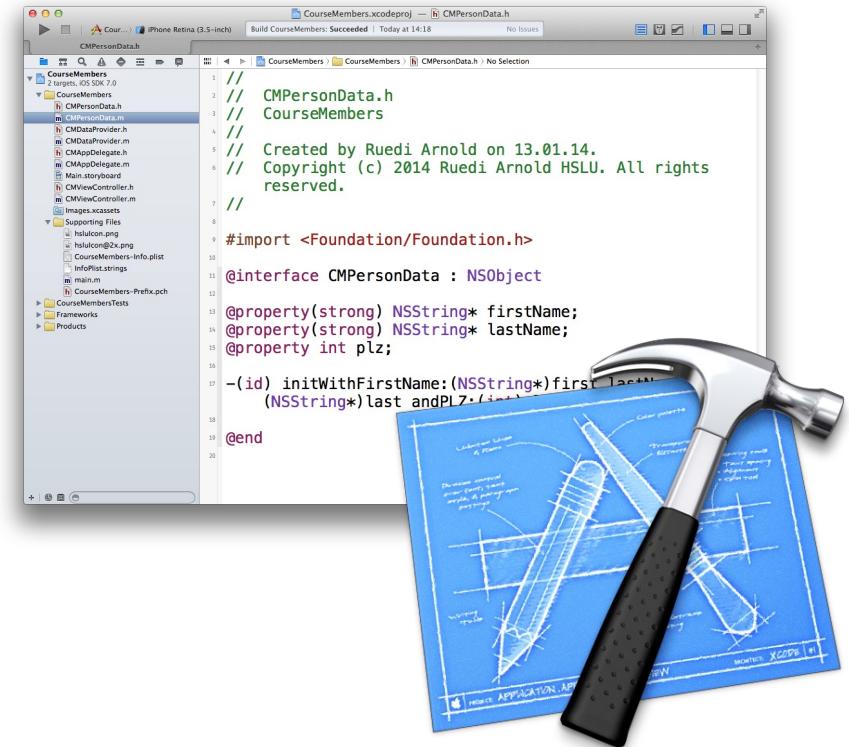


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

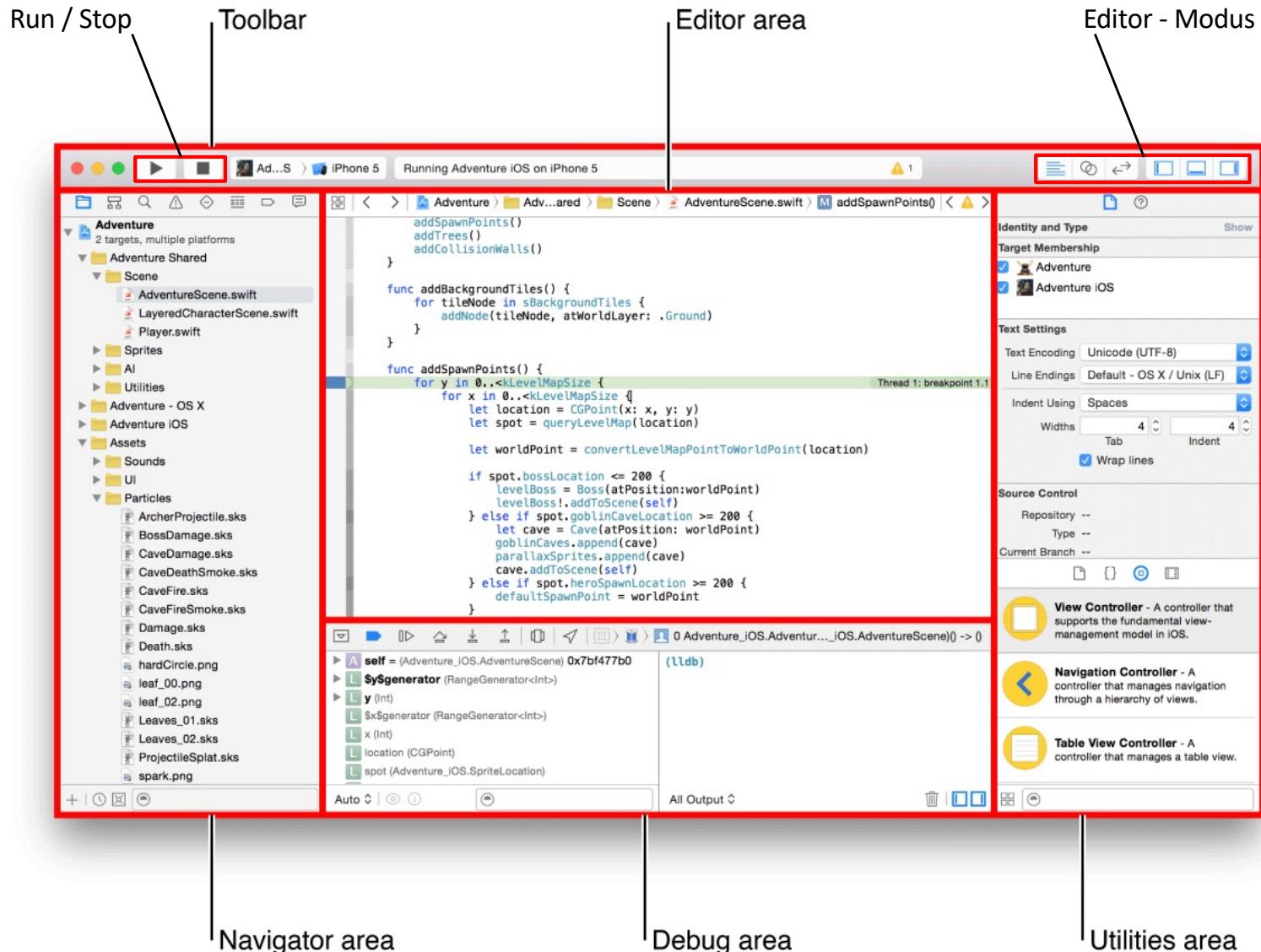
# Xcode

# Xcode IDE

- Code- und Daten-Editor
- Compiler
- iOS-Simulator
- Interface Builder
  - GUI Erstellung
- Debugger
- Instruments
  - Analyse Speicherverbrauch, Funktionsaufrufe, usw.
- Projekt- und Dateiverwaltung
- Versionsverwaltung (Git, SVN)



# Xcode



# Xcode Shortcuts

- esc Autocompletion
- cmd-r Run
- cmd-s Save
- cmd-b Build
- ⌘-Maus-[Code] Springen zur Deklaration
- alt-Maus-[Code] Zugriff Apple Doku

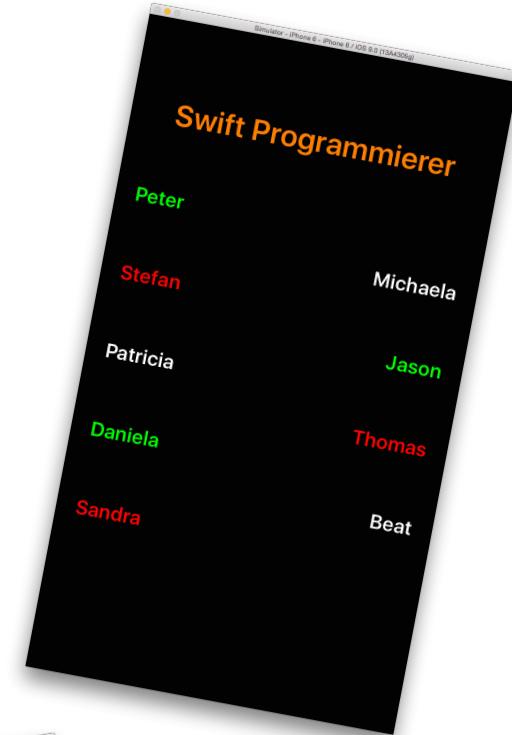


# Apple Resources

- **The Swift Programming Language**
  - <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>
  - Gute Einführung in die Sprache, sehr nützliche Quelle (vor allem der "Language Guide"!)
- **Xcode 14 + Swift 5.7**
  - <https://developer.apple.com/download/>
  - oder (bald) über den macOS App Store
- **Swift Standard Library Reference**
  - [https://developer.apple.com/documentation/swift/swift\\_standard\\_library](https://developer.apple.com/documentation/swift/swift_standard_library)

# Aufgabe 1: iOS-Kursliste

- 1. iPhone-App in Swift! 😊
  - Eigene Klassen & Methoden
  - Selber UIKit-Objekte instanziieren
  - Properties
  - Eigene init-Methode
  - if, for, switch, ...



| Swift Programmierer |           |      |
|---------------------|-----------|------|
| Peter               | Müller    | 8000 |
| Martin              | Walser    | 2345 |
| Quentin             | Tarantino | 3456 |
| Hans                | Stone     | 4567 |
| Otto                | Lotto     | 5678 |
| Dave                | Grohl     | 6789 |
| Johanna             | Obama     | 7890 |
| Stefan              | Test      | 8901 |
| Petra               | Müll      | 1234 |

# Übung 1

Programmieren für iOS



# Übung 1



# Singletons

- Singleton = Klasse, von der genau eine Instanz existiert
- In Swift mit `static let` gelöst
- Initializer sollte `private` sein
- Umstrittene Praxis – Reusability gefährdet
  - Alternative: Injection – Objekte „durchreichen“



```
class LoginManager {
 static let shared = LoginManager()

 private var isLoggedIn = false

 private init() {}
}
```

# Swift-Files (Best Practice)

- I.d.R. eine Swift-Datei pro Klasse/Struct
- Kleine Datenstrukturen / zusammengehörige Models dürfen auch mal in einem File sein
- Komplexe Klassen können mit **extension** auf mehrere Daten aufgeteilt werden

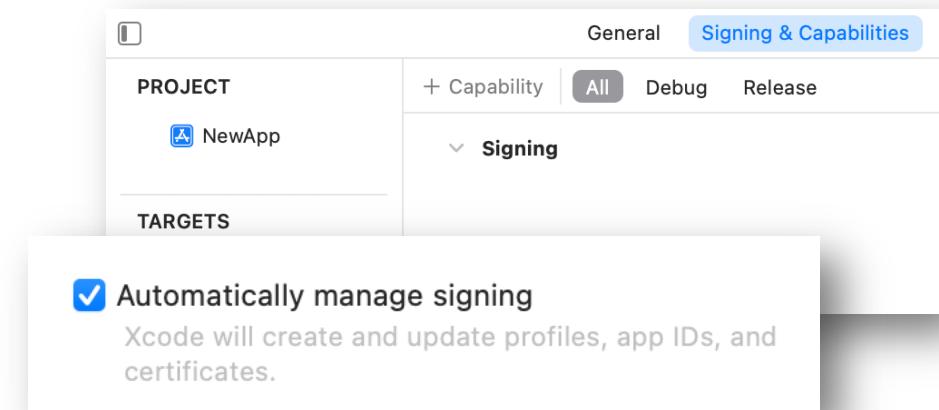
```
// UserManager.swift
class UserManager {
 var isLoggedIn = false
}

// LoginObserver.swift
protocol LoginObserver {
 func loginStateChanged(_ newState: Bool)
}

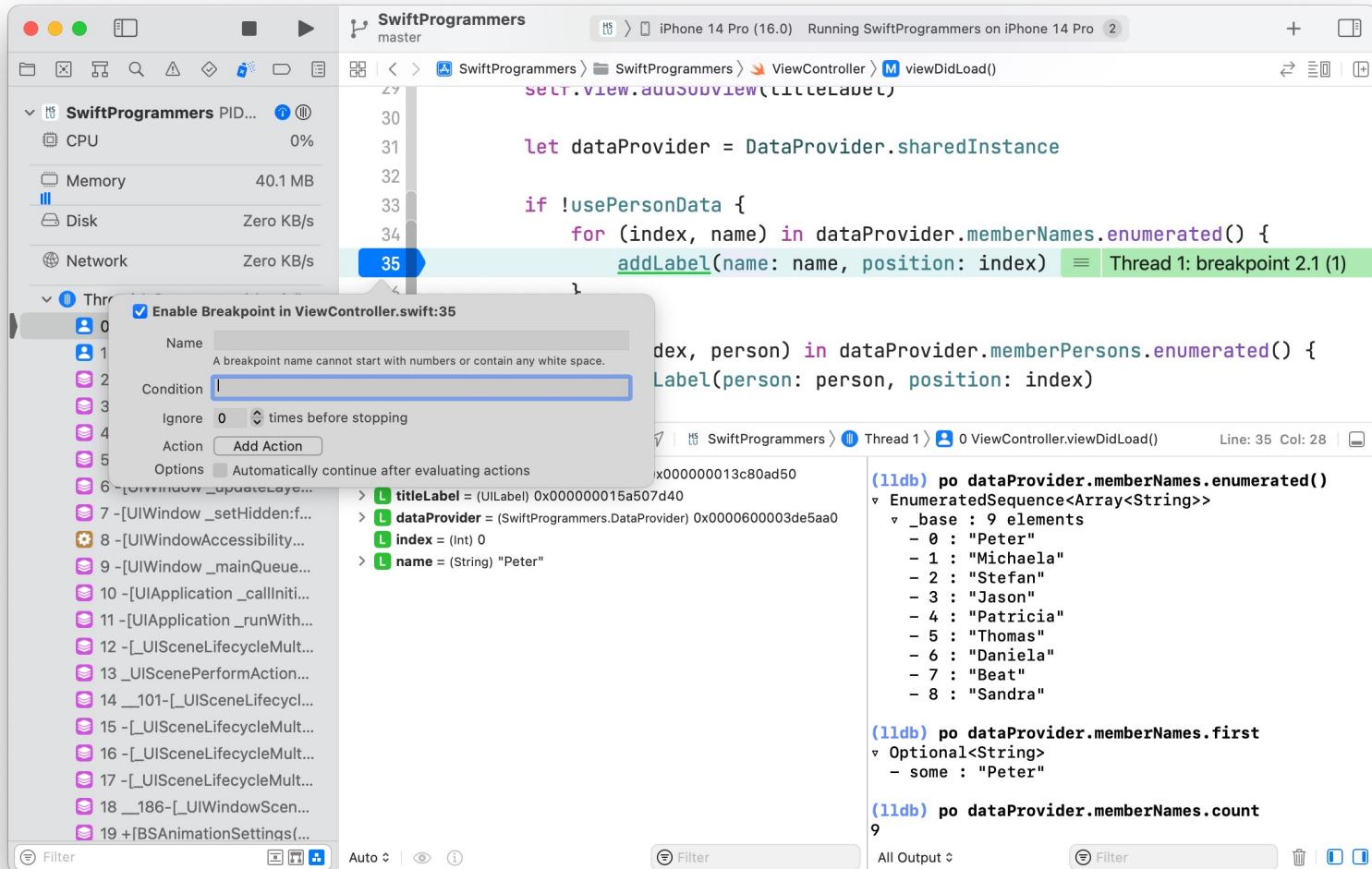
// UserManager+LoginDelegate.swift
extension MyImportantManager: LoginDelegate {
 func loginStateChanged(_ newState: Bool) {
 self.isLoggedIn = newState
 }
}
```

# Signing

- Apple kontrolliert stark, wie Apps auf iOS-Geräten installiert werden können (Monopol-Stellung)
- Nur Apps mit gültiger Signatur können auf registrierten Testgeräten gestartet werden
  - Signatur kann als sogenanntes «Provisioning Profile» von Apple geladen werden
- Xcode macht mittlerweile zum Glück **fast alles automatisch, einfach den Anweisungen folgen ;-)**
- Voraussetzung ist ein Gratis-Account beim Apple Developer Program
  - <https://developer.apple.com/programs/>



# Demo: Breakpoints und Debugger



# SwiftUI 1

Programmieren für iOS





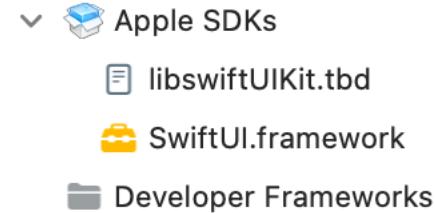
imgflip.com

# Was ist SwiftUI?

- Relativ neues Framework für User Interfaces
  - Von Apple zur Verfügung gestellte Programmzbibliothek zur Entwicklung von native iOS User Interfaces
  - Closed-Source :(
- Basiert auf Swift
  - Code-first Approach, aber mit mächtiger Preview

Choose frameworks and libraries to add:

Q swiftui X



# SwiftUI



Obj-C

iPhone SDK / UIKit

1984

2007

2008

Swift

2014

SwiftUI

2019

2022

## Footnotes

- Die Entwicklung von SwiftUI begann bereits etwa 4 Jahre früher
- Ein paar junge Apple-Engineers wollten einen moderneren Weg, um mit der neuen Sprache Swift Interfaces entwickeln zu können
- Das Projekt nahm sehr schnell Fahrt auf, bis zur WWDC19 haben mehr als 100 Entwickler zu SwiftUI beigetragen

# SwiftUI



Obj-C

iPhone SDK / UIKit

1984

2007

2008

Swift

2014

SwiftUI

2019

2022

#### Footnotes

- Die Entwicklung von SwiftUI begann bereits etwa 4 Jahre früher
- Ein paar junge Apple-Engineers wollten einen moderneren Weg, um mit der neuen Sprache Swift Interfaces entwickeln zu können
- Das Projekt nahm sehr schnell Fahrt auf, bis zur WWDC19 haben mehr als 100 Entwickler zu SwiftUI beigetragen

# SwiftUI vs UIKit

- UIKit
  - Immer noch **mächtiger**
  - Für iOS 12 und früher die einzige Option
  - Wird von Apple immer noch weiterentwickelt
- SwiftUI
  - Oft **einfacher** und eleganter
  - Weniger ausgereift, aber wird schnell weiterentwickelt
  - Wird immer wichtiger (z.B. Widgets)
  - «Under the hood» teilweise UIKit

ubique  Apps & Technology

- Meiste Apps noch mit UIKit
- SwiftUI wenn möglich (iOS 13+) oder nötig (Widgets)

# I learned today

- Heute Morgen im iOS-Chat:

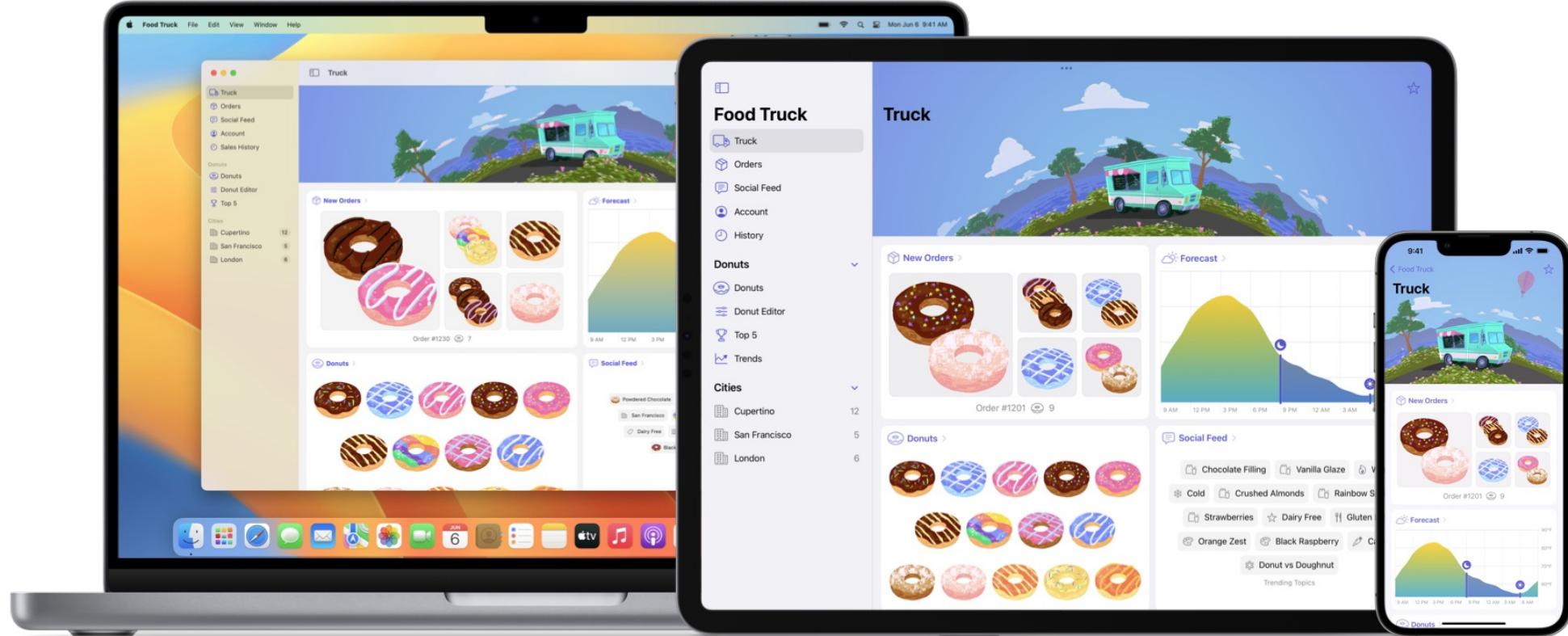


# Wieso SwiftUI lernen?

- State-of-the-art UI Entwicklung
  - Deklarativ, vgl. React, Flutter, Jetpack Compose
- Schnelle Resultate & Iterationen
  - Häufige UI-Ziele oft einfach zu erreichen
- Layout- & Style-Defaults
  - Apps sehen auch ohne Design-Vorgabe öfters gut aus
- Genaue Wysiwg-Preview
  - Einfache Design-Iterationen

# Multi-Plattform

SwiftUI funktioniert auf (fast) allen Apple-Plattformen, das User Interface passt sich automatisch an.



# Beispiel & Demo

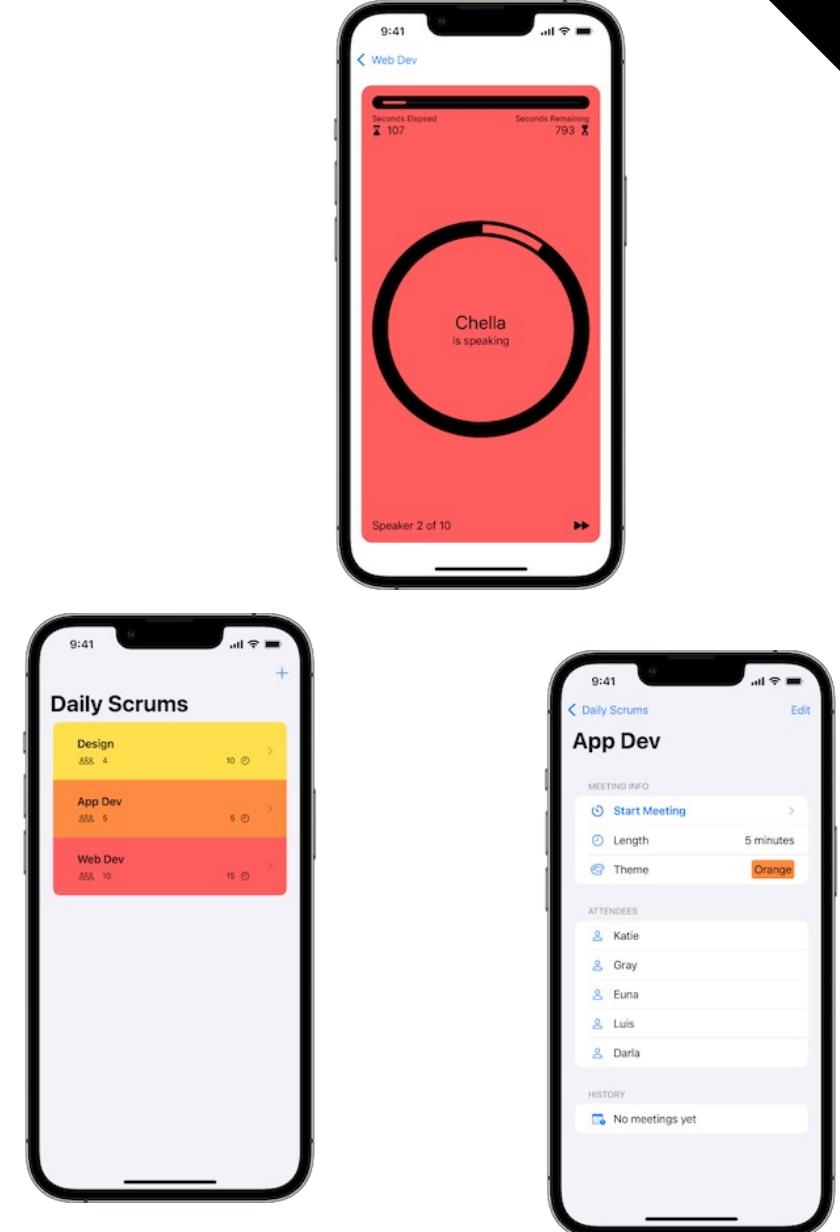


```
struct ContentView: View {
 var body: some View {
 Text("Hello, world!")
 .padding()
 }
}
```



# SwiftUI lernen

- SwiftUI lernt man am besten, in dem man Code schreibt
- Nächste Übungen: Kapitel aus Apple Dev Training „Scrumdinger“
  - Ziel: Einfache, aber komplette App mit SwiftUI umsetzen
- Vorlesung: Resultate der Übungen, Ergänzungen, Tipps, Hintergründe, Diskussionen
  - Annahme: Letzte Übung gelöst



# Farbschema der Slides

- **Grundlagen** – “Daily Business“ der App-Entwicklung
  - **Hintergründe** – tieferes Verständnis fördern
  - **Übungen** – Instruktionen zur nächsten Übung
- 
- **Advanced** – Wegweiser zu weiteren APIs und Techniken
  - **Trivia** – nicht wirklich (prüfungs)relevant
  - Meta – Organisation & Informationen zur Vorlesung

# Inputs zu Übung 2

Programmieren für iOS



# Übung 2

- Kapitel 1-4 von „Scrumdinger“
- Kleine zusätzliche Anpassungen
- Lernziele
  - Standard-UI mit SwiftUI erstellen
  - Einfache Daten anzeigen und bearbeiten
  - Arbeiten mit SwiftUI-Views
  - State und Bindings kennenlernen

# Views

- Alle Teile des User Interfaces sind «Views»
- Vom gesamte Screens über zusammengesetzte Elemente bis zum kleinen Icon



```
struct MyFirstView: View {
 var body: some View {
 Text("Hello HSLU")
 }
}
```

- Jede neue SwiftUI-View
  - ist ein `struct`
  - implementiert das Protokoll `View`
  - hat eine Property `body`, welche wieder eine View zurückgibt

# struct vs. class (1/2)

- Viele Gemeinsamkeiten (Properties, Methoden, Initializer, ...)
- Wichtigster Unterschied: Referenzierung

```
● ● ●
struct Dog {
 var name: String
 var age: Double = 0.0
 init(_ name: String) { self.name = name }
}

var foo = 42
var bar = 10
foo = bar
var myDog = Dog("Cooper")
var other = myDog
```

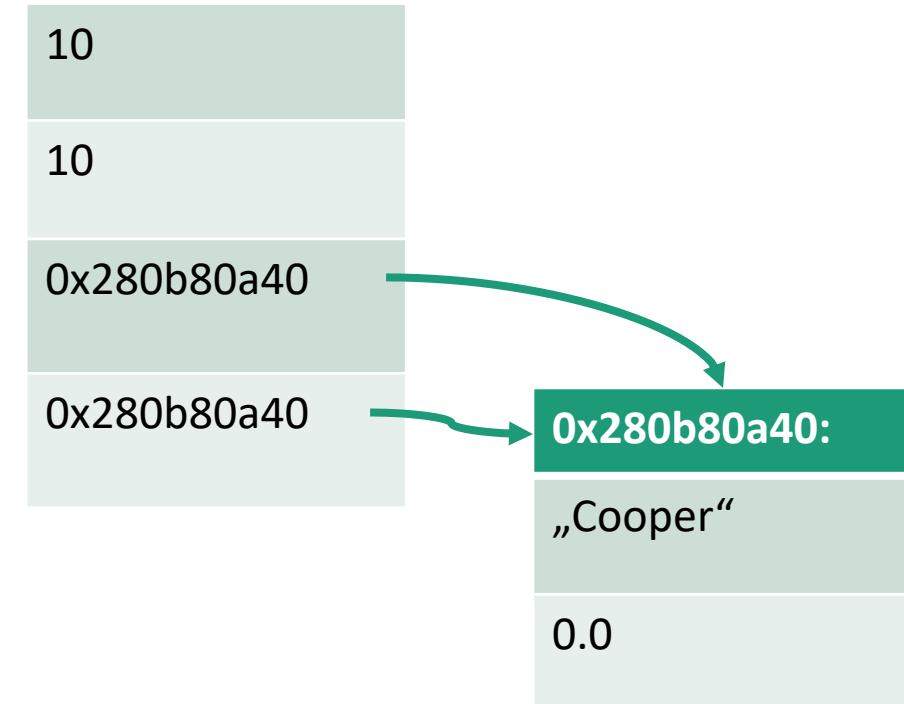
|          |
|----------|
| 10       |
| 10       |
| „Cooper“ |
| 0.0      |
| „Cooper“ |
| 0.0      |

# struct vs. class (2/2)

- Viele Gemeinsamkeiten (Properties, Methoden, Initializer, ...)
- Wichtigster Unterschied: Referenzierung

```
● ○ ●
class Dog {
 var name: String
 var age: Double = 0.0
 init(_ name: String) { self.name = name }

 var foo = 42
 var bar = 10
 foo = bar
 var myDog = Dog("Cooper")
 var other = myDog
```

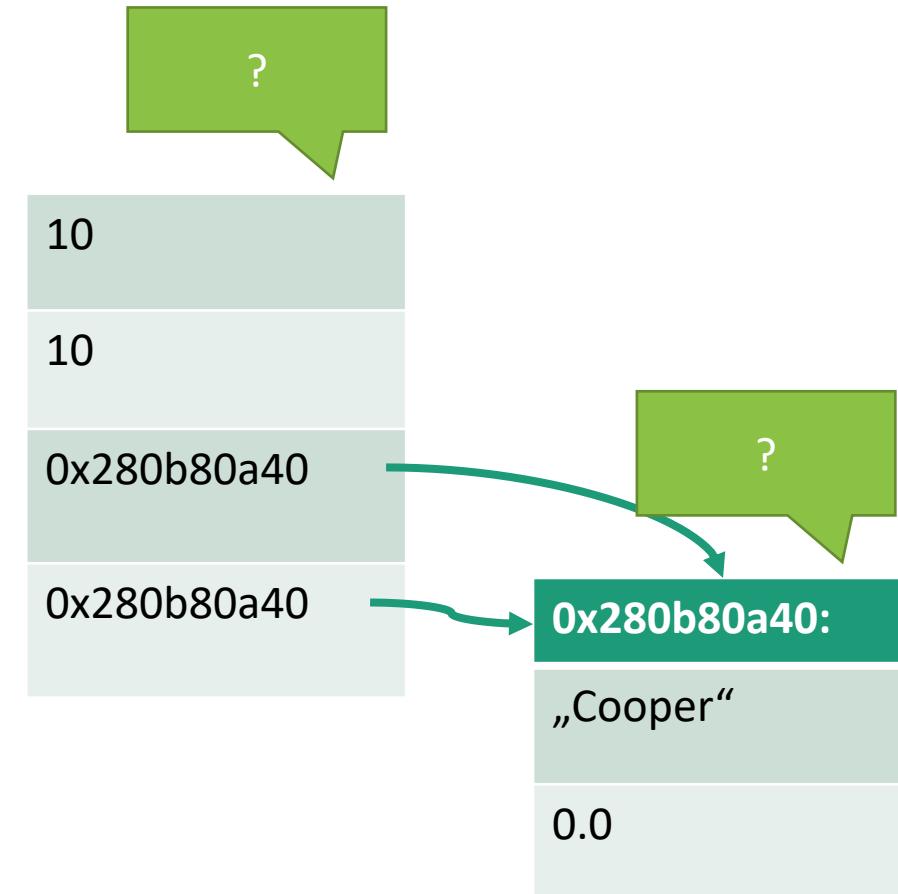


# struct vs. class (2/2)

- Viele Gemeinsamkeiten (Properties, Methoden, Initializer, ...)
- Wichtigster Unterschied: Referenzierung

```
● ● ●
class Dog {
 var name: String
 var age: Double = 0.0
 init(_ name: String) { self.name = name }

 var foo = 42
 var bar = 10
 foo = bar
 var myDog = Dog("Cooper")
 var other = myDog
```

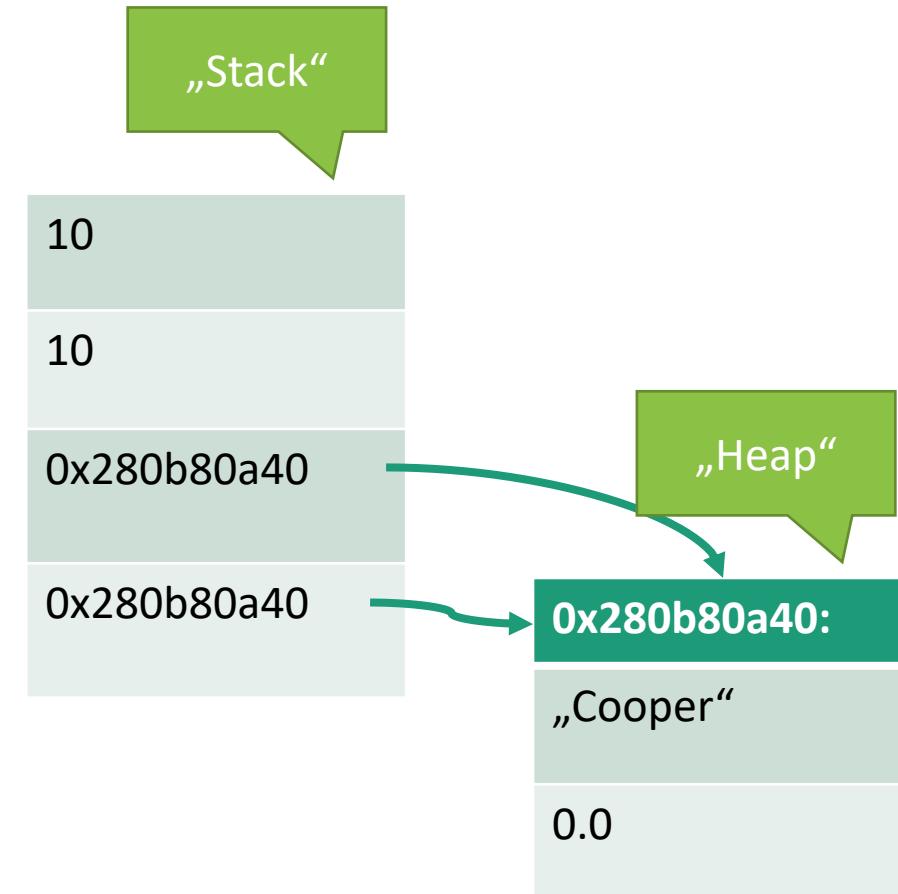


# struct vs. class (2/2)

- Viele Gemeinsamkeiten (Properties, Methoden, Initializer, ...)
- Wichtigster Unterschied: Referenzierung

```
● ● ●
class Dog {
 var name: String
 var age: Double = 0.0
 init(_ name: String) { self.name = name }

 var foo = 42
 var bar = 10
 foo = bar
 var myDog = Dog("Cooper")
 var other = myDog
```



# Swift Structs

- Auch komplexe Daten-Typen sind grundsätzlich **struct**
- Copy-on-write als wichtige Compiler-Optimierung



```
public struct Int { ... }
```

```
public struct Array<Element> { ... }
```

```
public struct String { ... }
```



```
let myArr = Array<Int>(repeating: 42, count: 1_000_000)
var myBrr = myArr // fast
myBrr[999] = 41 // slow
print(myArr[999]) // 42
```

- Wann **class** verwenden?
  - Faustregel: Nur, wenn Referenzierung semantisch Sinn macht

# UIKit: View-Klasse

```
// UIKit
class UILabel : UIView { ... }

// SwiftUI
struct Text : View { ... }
```

Wieso verwendet UIKit *class*  
und SwiftUI *struct*?

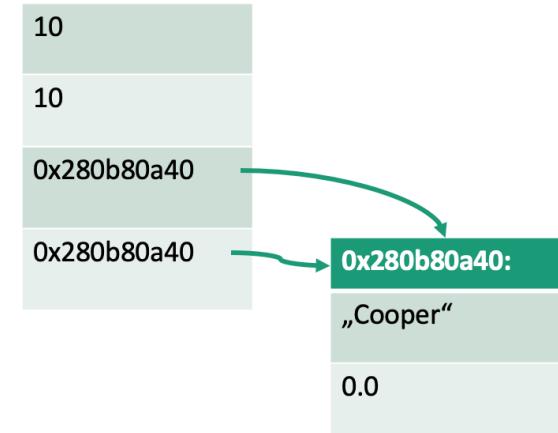
- Jede UIView speichert etwa 200 Properties und Abhängigkeiten (backgroundColor, frame und viele mehr), die nötig sind, um von der Grafikkarte effizient auf das Display projiziert zu werden.
- Das Erstellen neuer Objekte ist deshalb wesentlich teurer als das verändern einzelner Eigenschaften
- Erstellte Objekte werden so lange wie möglich verwendet und müssen deshalb referenziert werden können.

# SwiftUI: View-Struct

- SwiftUI: Die Objekt-Instanzen zur Darstellung sind auch hier nötig
- Die View-Structs dienen als „**Bauplan**“ mit dem das System die Objekte erstellt und verwaltet
  - Legacy: Tatsächlich werden dafür noch viele UIKit-Objekte verwendet
  - Mittelfristig wird SwiftUI wohl von UIKit unabhängig, aber der Entwicklungsvorsprung von UIKit war wichtig
- Der Entwicklungsstil mit UIKit wird als **imperativ** bezeichnet
  - Der App-Code besteht aus expliziten Modifikationen der Eigenschaften von UIView-Objekten, sodass diese das gewünschte User Interface zeichnen
- Der Entwicklungsstil mit SwiftUI wird als **deklarativ** bezeichnet
  - Der App-Code repräsentiert direkt die erhoffte Darstellung

# Wieso also sind Views Structs?

- Performance: Für kleine Datenmengen sind Structs effizienter. Eine SwiftUI-View muss nur die relevanten Eigenschaften repräsentieren
- Korrektheit: Eine `let`-Variabel mit einem Struct lässt keine unerwünschten Veränderungen zu, bei einer `let`-Variabel von einer Klassen-Instanz ist nur die Referenz direkt geschützt
- Semantik: Man wird gezwungen, die View als isolierten Status oder als Daten zu sehen und ist nicht versucht, in (imperativen) Änderungen von bestehenden Objekten zu denken

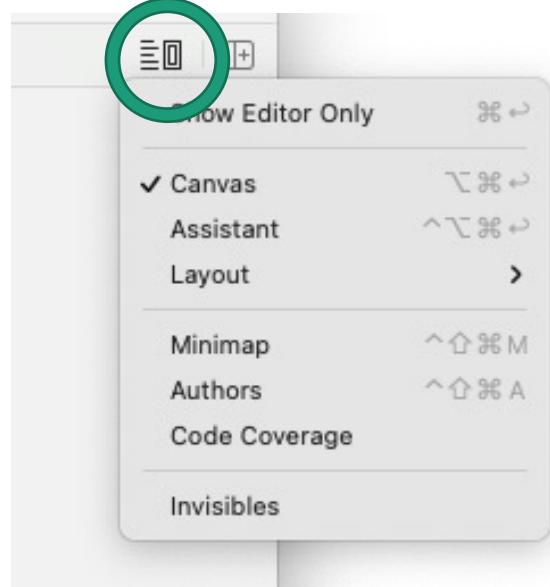


# Weitere Inhalte der Übung

- SF Symbols
- Asset Catalog
- Accessibility

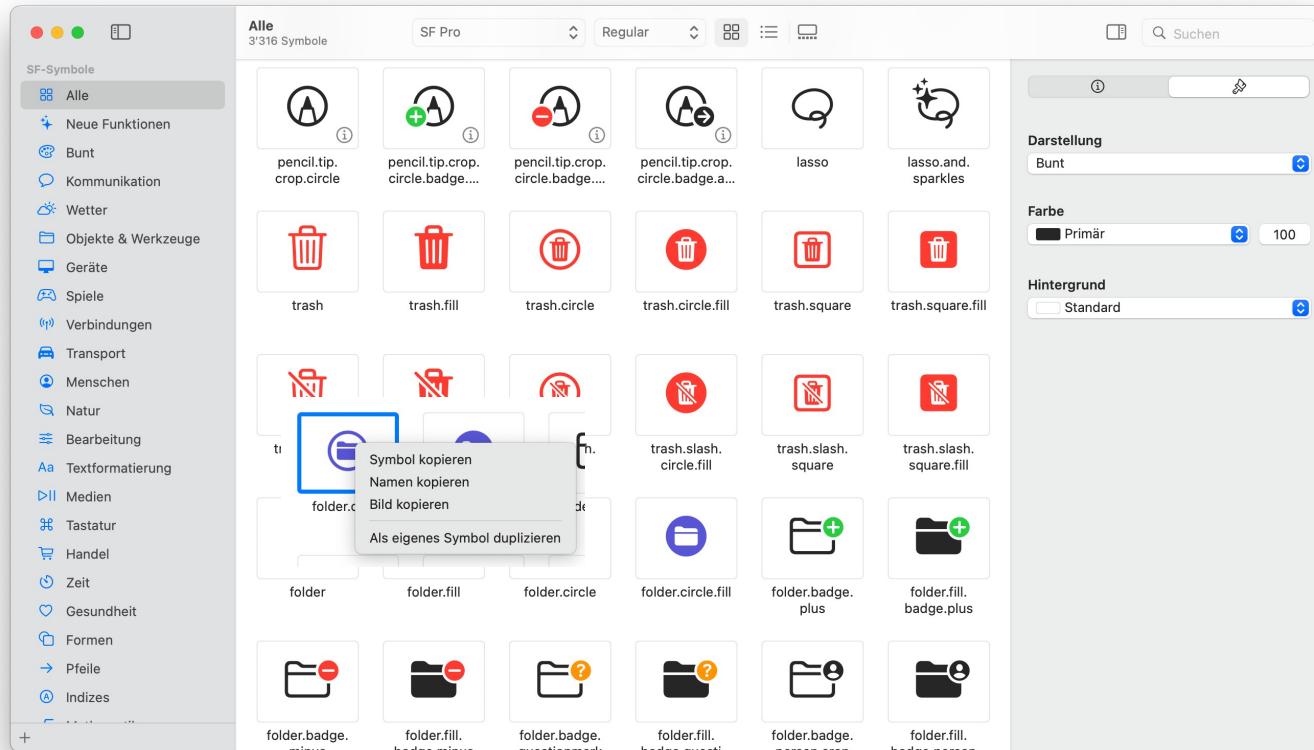
# Editor

- Live-Preview (Canvas)

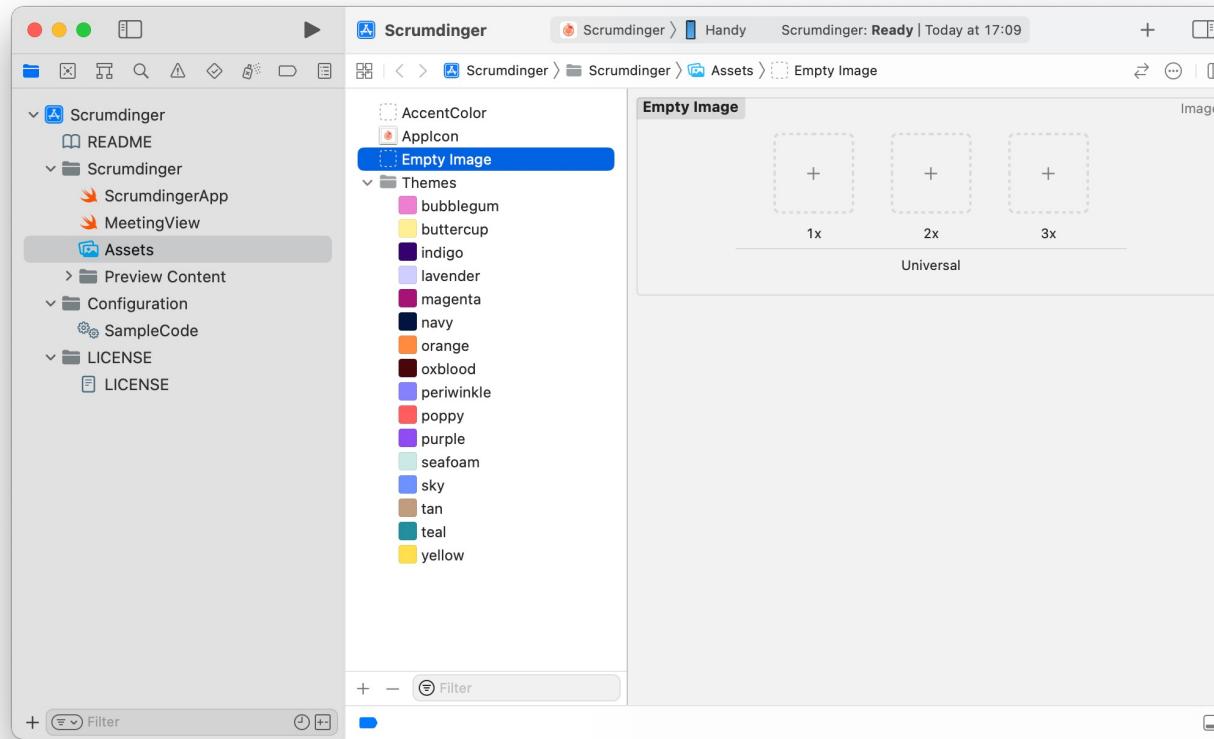
A screenshot of Xcode displaying the ContentView.swift file. The code defines a SwiftUI struct ContentView with a body containing a Text view that displays "Hello, world!". To the right of the editor, a large iPhone simulator window shows the actual application running on the device, displaying the text "Hello, world!".

# SF Symbols

- Sammlung mit 4'000 Icons von Apple
  - (Fast) frei verwendbar
- App Download: <https://developer.apple.com/sf-symbols/>



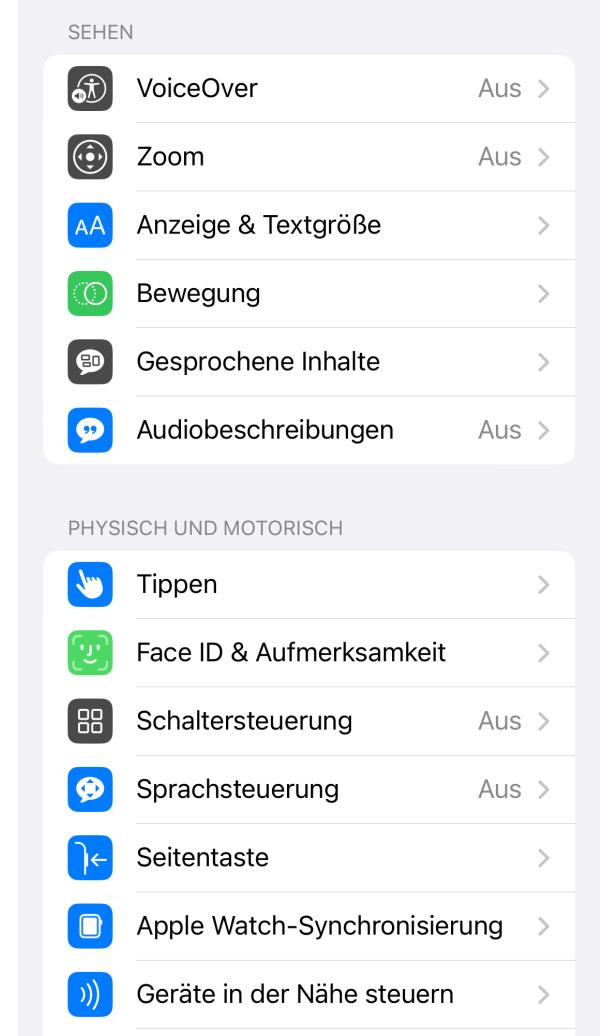
# Asset Catalog



- Datei / Ordner-Struktur, in der „Assets“ (primär Bilder und Farben) abgelegt werden

# Barrierefreiheit

- Apps können und sollten für Nutzer mit Einschränkungen besser zugänglich gemacht werden („Accessibility-Optimierungen“)
- Einige Zahlen aus der Schweiz (2016\*)
  - 100'000 Personen sehschwach oder blind
  - 300'000 mit Problemen mit Armen oder Händen, für 150'000 Umgang mit Gegenständen schwierig
  - 150'000 mit Lernschwierigkeiten, 100'000 mit Kommunikationsproblemen
- Viele betroffene Nutzer verwenden iPhones, da System sehr viele Optionen bieten
- Sehr häufige Optimierung: VoiceOver
  - <https://youtu.be/2IXxJ3W6HD8?t=220>



# Wie löse ich ... in SwiftUI?

- Verhalten und Lösungen sind am Anfang nicht immer intuitiv
- Gesamt-Dokumentation durch generische Syntax und viele Protokolle oft kaum zu gebrauchen :-(
- Viele Lösungen zu konkreten Problemen
  - [www.hackingwithswift.com](http://www.hackingwithswift.com)
  - [www.stackoverflow.com](http://www.stackoverflow.com) – SwiftUI 4!
- Oder: Im Ilias-Forum oder vor Ort Fragen stellen und Lösungen diskutieren



```
init(content: () -> Content)
Creates a list with the given content.
Available when SelectionValue is Never and Content conforms to View.

init<Data, ID, RowContent>(Data, id: KeyPath<Data.Element, ID>,
rowContent: (Data.Element) -> RowContent)
Creates a list that identifies its rows based on a key path to the identifier of the
underlying data.
Available when SelectionValue is Never and Content conforms to View.

init<Data, ID, RowContent>(Data, id: KeyPath<Data.Element, ID>,
selection: Binding<Set<SelectionValue>>?, rowContent:
(Data.Element) -> RowContent)
Creates a list that identifies its rows based on a key path to the identifier of the
underlying data, optionally allowing users to select multiple rows.
Available when SelectionValue conforms to Hashable and Content conforms
to View.

init<Data, ID, RowContent>(Data, id: KeyPath<Data.Element, ID>,
selection: Binding<SelectionValue>?, rowContent: (Data.Element)
-> RowContent)
Creates a list that identifies its rows based on a key path to the identifier of the
underlying data, optionally allowing users to select a single row.
Available when SelectionValue conforms to Hashable and Content conforms
to View.

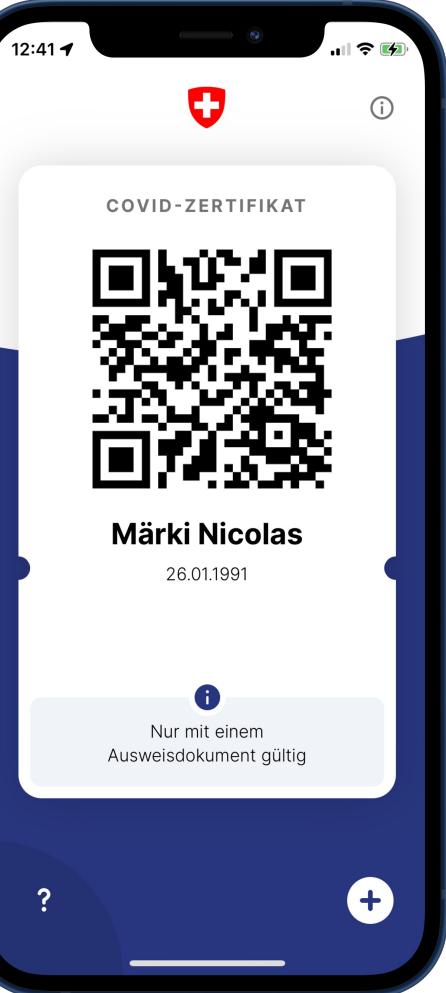
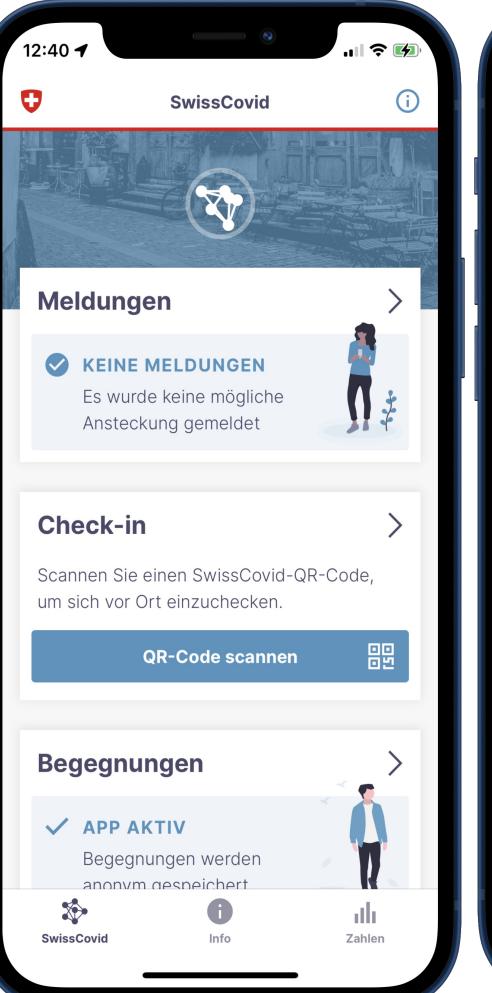
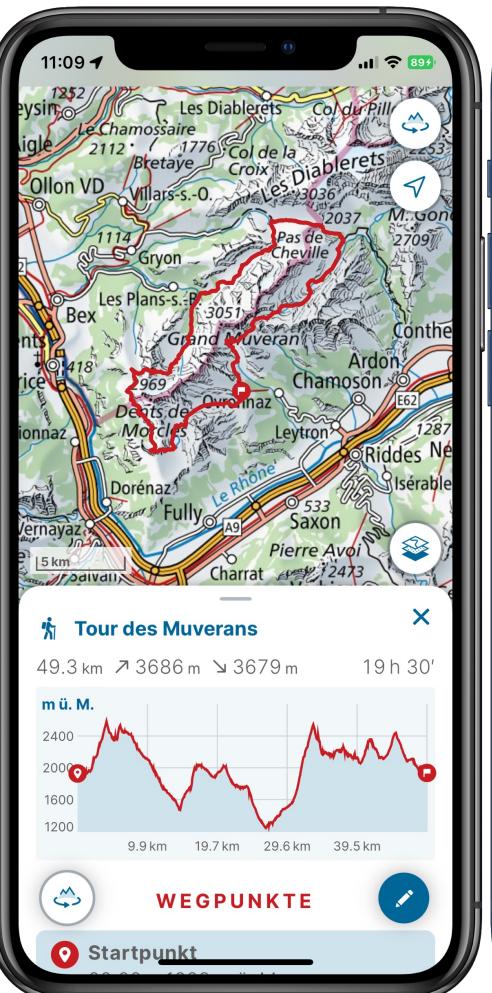
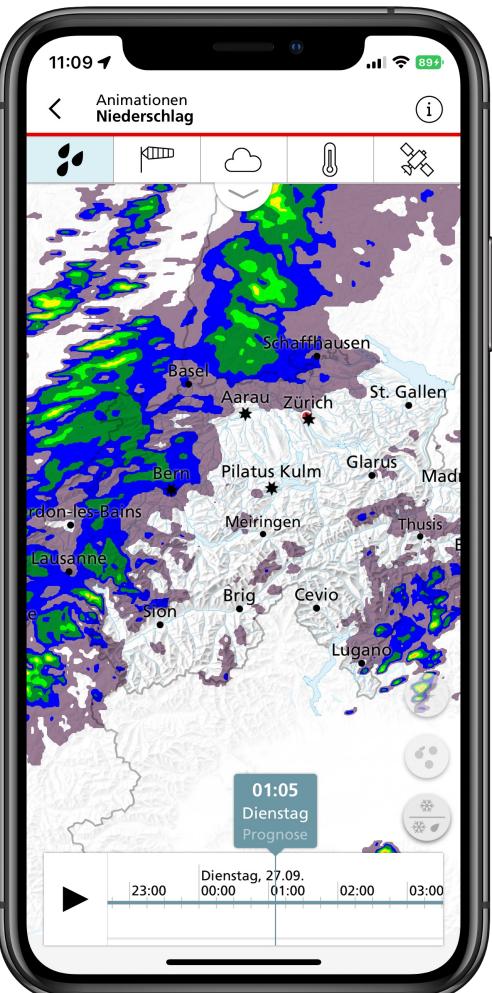
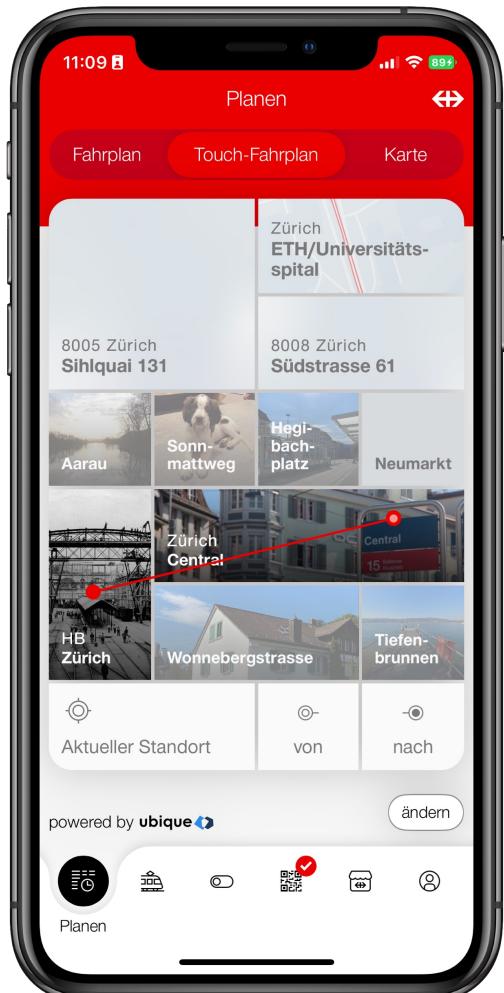
init<Data, ID, RowContent>(Data, rowContent: (Data.Element) -> RowContent)
computes its rows on demand from an underlying collection of
SelectionValue is Never and Content conforms to View.

>(Range<Int>, rowContent: (Int) -> RowContent)
computes its views on demand over a constant range.
SelectionValue is Never and Content conforms to View.

>(Range<Int>, selection: Binding<Set<SelectionValue>>?, content:
(Int) -> RowContent)
computes its views on demand over a constant range.
SelectionValue conforms to Hashable and Content conforms to View.

content>(Data, selection: Binding<Set<SelectionValue>>?, content:
(Data.Element) -> RowContent)
computes its rows on demand from an underlying collection of
SelectionValue conforms to Hashable and Content conforms to View.
```

# Wieso ist das so?



# Danke, das wars!

- Nächste Woche, mehr SwiftUI...
  - Layout-Verhalten
  - States und Data-Flow im Detail
  - Animationen und Transitions

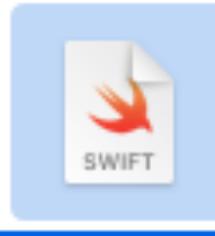
# SwiftUI 2

Programmieren für iOS



# Neue View erstellen

- New File → SwiftUI Template



SwiftUI View

## Wann? Oft!

- Ordnung: Verschachtelte Views werden schnell unübersichtlich
- Struktur: Klar definiert, welche Properties (State, Binding, ...) eine View ausmachen
- Performance: Views helfen dem Framework bei Optimierungen
- Reusability: Einfachere Views können öfters verwendet werden

# Previews

- Aufwand für sinnvolle Previews lohnt sich eigentlich immer!

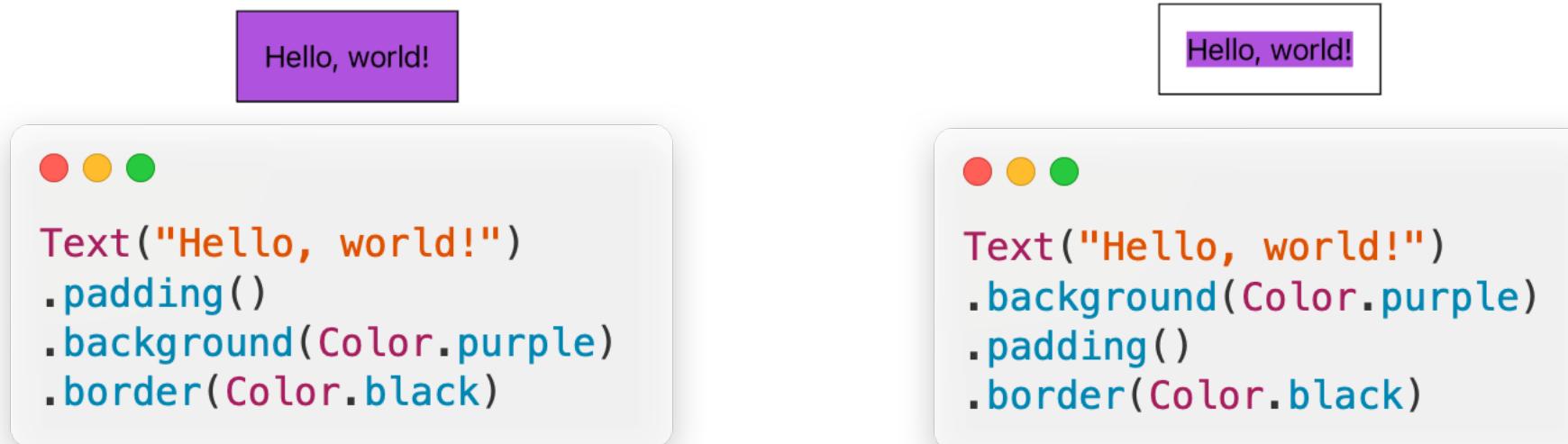
# View Modifiers

- Jeder View Modifier erstellt eine neue View

```
/// - Returns: A view that pads this view using the specified edge insets
/// with specified amount of padding.
```

```
@inlinable public func padding(_ insets: EdgeInsets) → some View
```

- Die **Reihenfolge** der Modifiers ist oft entscheidend!



# Eigene Modifiers

- Mit dem **ViewModifier**-Protokoll können beliebige eigene Modifier definiert werden:

```
● ● ●
struct Sandwich: ViewModifier {
 let base: String
 func body(content: Content) -> some View {
 VStack {
 Text(base)
 content.bold()
 Text(base)
 }
 }
}

extension View {
 func sandwiched(base: String) -> some View {
 modifier(Sandwich(base: base))
 }
}
```

Bread  
Hummus  
Bread

Toast  
Cheese  
Toast

```
● ● ●
struct ContentView: View {
 var body: some View {
 HStack {
 Text("Hummus")
 .modifier(Sandwich(base: "Bread"))
 Divider()
 Text("Cheese")
 .sandwiched(base: "Toast")
 }
 }
}
```

# Environment

Einige Modifiers verändern die View nicht direkt, gelten für die gesamte innerer Hierarchie

Hello  
Word  
Text

```
●●●

struct ContentView: View {
 var body: some View {
 VStack {
 Text("Hello")
 .foregroundColor(.blue)
 Text("Word")
 Text("Text")
 }
 .foregroundColor(.red)
 }
}
```

- Dafür verwendet SwiftUI das Konzept von Environment-Values
  - Mit `@Environment` kann der aktuelle Wert gelesen werden
  - Der `.environment` Modifier überschreibt den Wert für Subviews

```
●●●

struct InnerView: View {
 @Environment(\.font) var font
 var body: some View {
 Text("Test").font(font?.bold())
 }
}

struct ContentView: View {
 var body: some View {
 Text("Test").environment(\.font, .caption)
 }
}
```

# EnvironmentKey

- Bevor eigene Eigenschaften als Environment verwendet werden können, müssen diese als `EnvironmentKey` definiert werden
- In Praxis jedoch selten nötig

```
private struct SecondaryForegroundColor: EnvironmentKey {
 static let defaultValue = Color(.black)
}

extension EnvironmentValues {
 var secondaryForegroundColor: Color {
 get { self[SecondaryForegroundColor.self] }
 set { self[SecondaryForegroundColor.self] = newValue }
 }
}

@Environment(\.secondaryForegroundColor) var secondaryForegroundColor
```

# Text-Tipps

- Text-Views können übrigens mit + sehr einfach kombiniert werden

```
Text("Hallo ") + Text("HSLU").bold().foregroundColor(.red)
```

- Text-Views können auch aus Image-Views erstellt werden

```
Text("Hallo ") + Text(Image(systemName: "house"))
```

Hallo **HSLU**

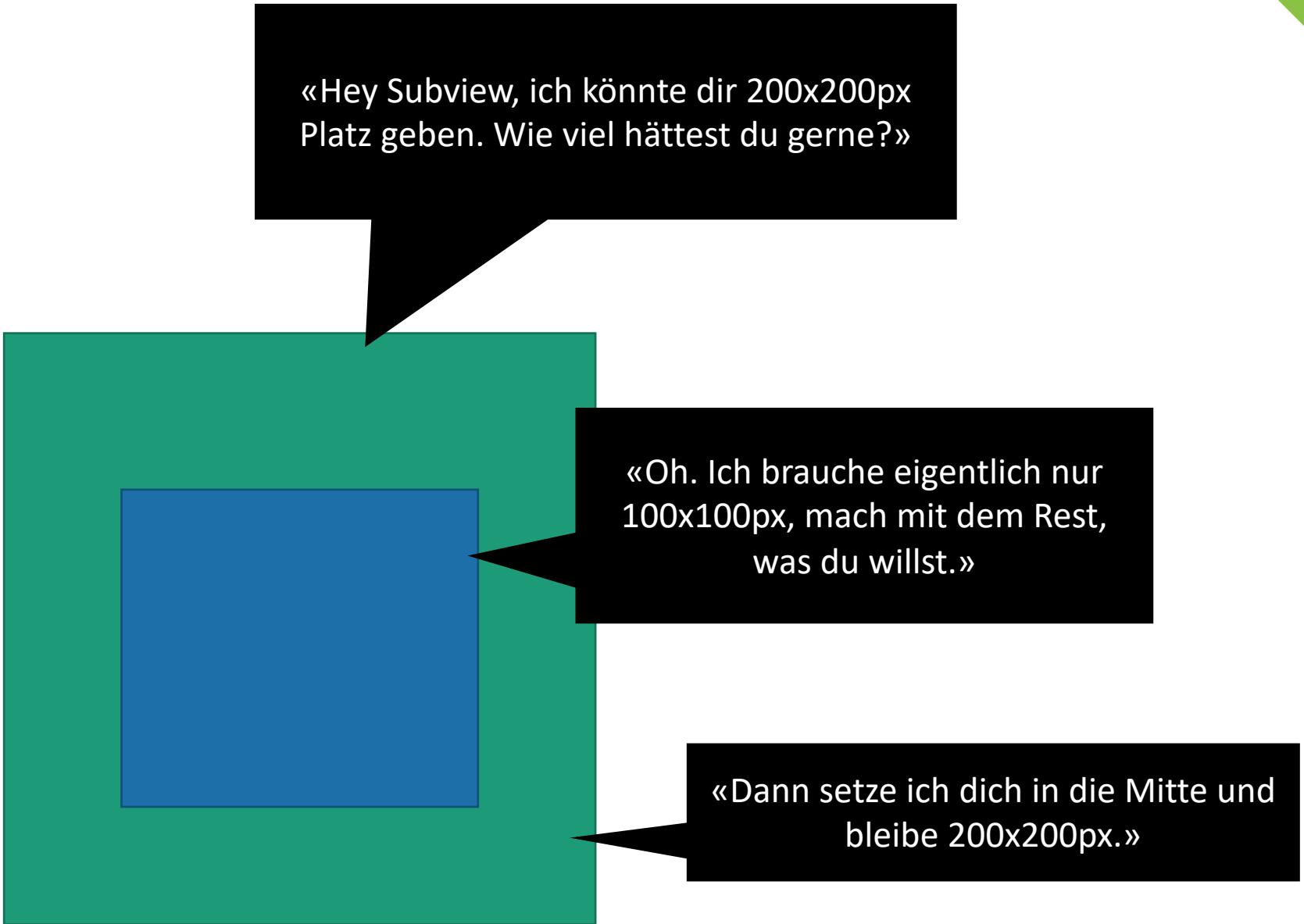
Hallo 

# Layout

Programmieren für iOS



# Layout-Dialog

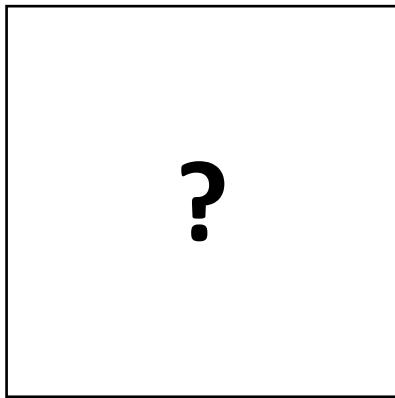


# Layout Mechanismus

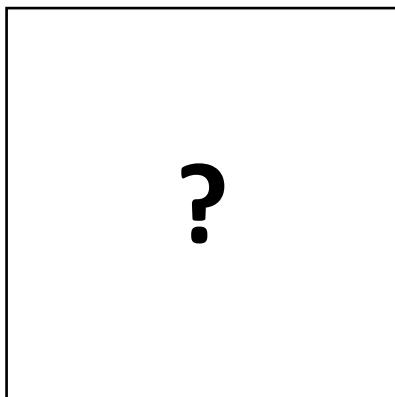
- Alle Layouts in SwiftUI funktionieren mit den gleichen drei Regeln:
  1. Eine View bietet einer Subview eine bestimmte Grösse an
  2. Die Subview kann diese Grösse annehmen oder **sich für eine andere Grösse entscheiden**
  3. Die Container-View positioniert die Subview
- Diese Schritte werden rekursiv auf alle Subviews angewendet
- Layouts entstehen, in dem Größen und Positionen in **unterschiedlicher Reihenfolge** oder mit **unterschiedlichen Prioritäten** berechnet werden. Einige Beispiele...

# Layout Mechanismus

Wie werden diese beiden Views aussehen?



```
VStack(spacing: 0) {
 Color.blue
 Color.red
}.frame(width: 200, height: 200)
```



```
VStack(spacing: 0) {
 Color.blue.frame(height: 150)
 Color.red
}.frame(width: 200, height: 200)
```

# Layout Mechanismus

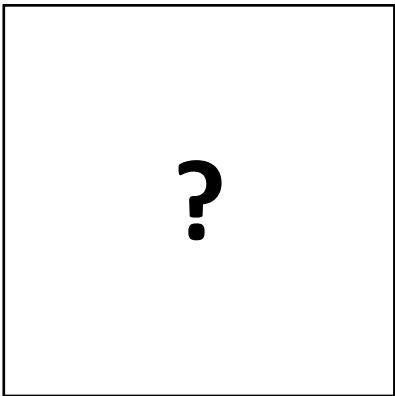


```
VStack(spacing: 0) {
 Color.blue
 Color.red
}.frame(width: 200, height: 200)
```



```
VStack(spacing: 0) {
 Color.blue.frame(height: 150)
 Color.red
}.frame(width: 200, height: 200)
```

# Layout Mechanismus



Wie wird diese View  
aussehen?

```
Image(systemName: "house")
 .background(Color.green)
 .frame(width: 200, height: 200)
 .background(Color.blue)
```

# Layout Mechanismus



Wie wird diese View aussehen?

```
Image(systemName: "house")
 .background(Color.green)
 .frame(width: 200, height: 200)
 .background(Color.blue)
```

# Frame

- *frame()* erstellt eine **neue View** mit der angegebenen Breite und/oder Höhe

```
/// - Returns: A view with fixed dimensions of `width` and `height`, for the
/// parameters that are non-`nil`.
```

```
@inlinable public func frame(width: CGFloat? = nil, height: CGFloat? = nil, alignment: Alignment = .center) → some View
```

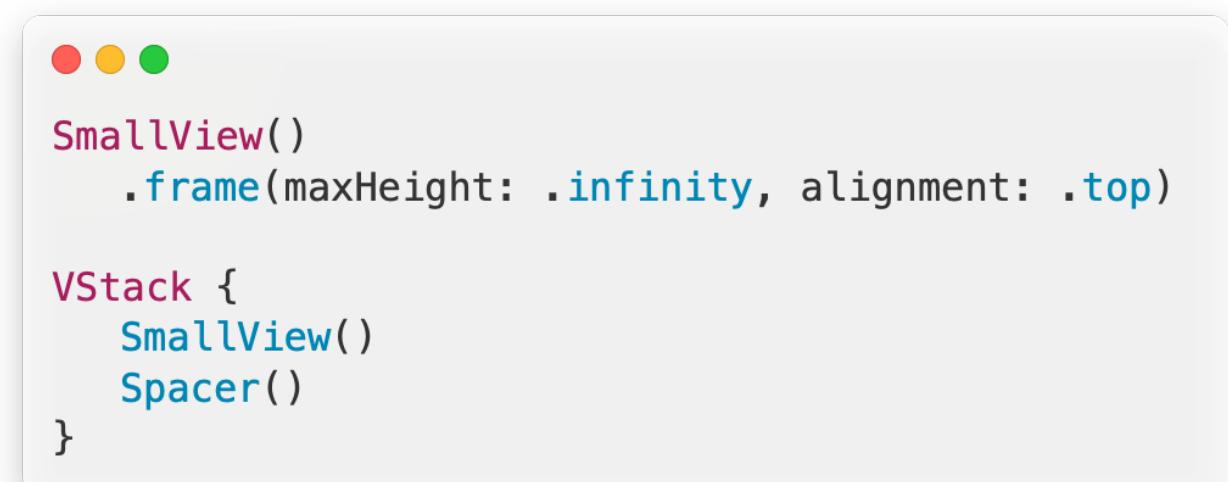
- Die ursprüngliche View **muss die neue Grösse nicht übernehmen**
- Falls die View die Grösse vom neuen Frame ignoriert, wird sie mit dem angegebenen *Alignment* positioniert



# Size Constraints

- Genauer: Im Layout-Dialog wird nicht eine einzelne Grösse diskutiert, sondern minimale, maximale und ideale Breite, rsp. Höhe.
- Nützlich: Frame-Modifier oft ein Ersatz für Stack+Spacer

```
/// - Parameters:
/// - minWidth: The minimum width of the resulting frame.
/// - idealWidth: The ideal width of the resulting frame.
/// - maxWidth: The maximum width of the resulting frame.
/// - minHeight: The minimum height of the resulting frame.
/// - idealHeight: The ideal height of the resulting frame.
/// - maxHeight: The maximum height of the resulting frame.
/// - alignment: The alignment of this view inside the resulting frame.
/// Note that most alignment values have no apparent effect when the
/// size of the frame happens to match that of this view.
///
/// - Returns: A view with flexible dimensions given by the call's non-'nil'
/// parameters.
@inlinable public func frame(minWidth: CGFloat? = nil, idealWidth: CGFloat? = nil, maxWidth:
 CGFloat? = nil, minHeight: CGFloat? = nil, idealHeight: CGFloat? = nil, maxHeight:
 CGFloat? = nil, alignment: Alignment = .center) -> some View
```



# Mehr Beispiele

Wie werden diese beiden Views aussehen?

```
VStack(spacing: 0) {
 Color.blue.frame(height: 50)
 Color.red.frame(height: 70)
}.frame(width: 200, height: 200).border(Color.black)
```

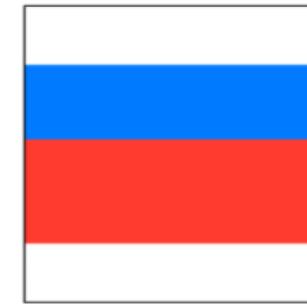
?

```
VStack(spacing: 0) {
 Color.blue.frame(height: 350)
 Color.red.frame(height: 100)
}.frame(width: 200, height: 200).border(Color.black)
```

?

# Mehr Beispiele

```
VStack(spacing: 0) {
 Color.blue.frame(height: 50)
 Color.red.frame(height: 70)
}.frame(width: 200, height: 200).border(Color.black)
```



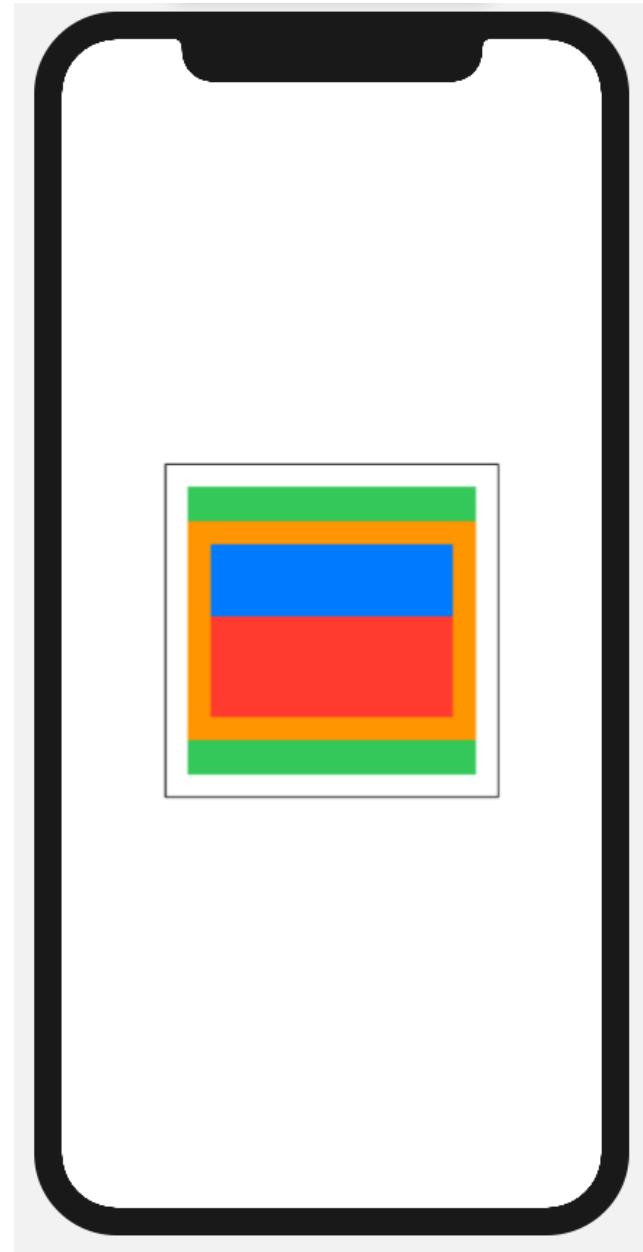
```
VStack(spacing: 0) {
 Color.blue.frame(height: 350)
 Color.red.frame(height: 100)
}.frame(width: 200, height: 200).border(Color.black)
```



# Komplexes Beispiel

```
1 VStack(spacing: 0) {
2 Color.blue.frame(height: 50)
3 Color.red.frame(height: 70)
4 }
5 .padding()
6 .background(Color.orange)
7 .frame(width: 200, height: 200)
8 .background(Color.green)
9 .padding()
10 .border(Color.black)
```

1  
2



# Diskussion: Layouts

- Welche Regeln wenden die Views an, die wir bisher gesehen haben?
  - (H/V/Z)-Stack
  - Text
  - Image
  - Spacer

# Image Layout

- Reminder: Auch `Image`-Views folgen der Layout-Logik und können deshalb nicht einfach mit einem `frame`-Modifier vergrössert oder verkleinert werden
- Der `resizable`-Modifier ändert dieses Verhalten, sodass das Bild die vorgegebene Grösse übernimmt
- Achtung: ohne `aspectRatio` bleibt das Seitenverhältnis nicht erhalten!



```
var body: some View {
 Image("my-image")
 .resizable()
 .aspectRatio(contentMode: .fit)
 .frame(width: 100)
}
```

# Grids, Teil 1

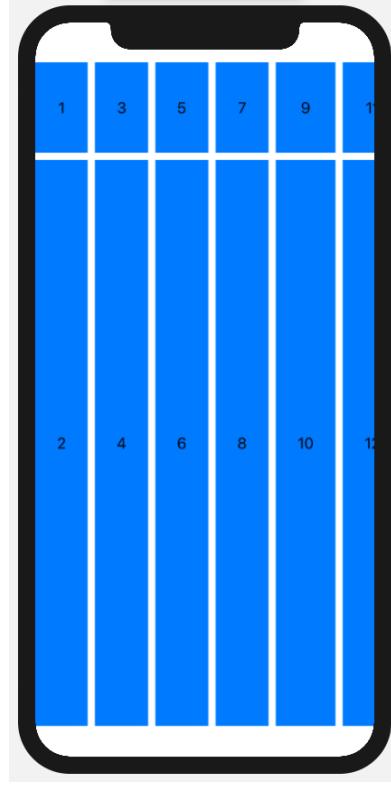
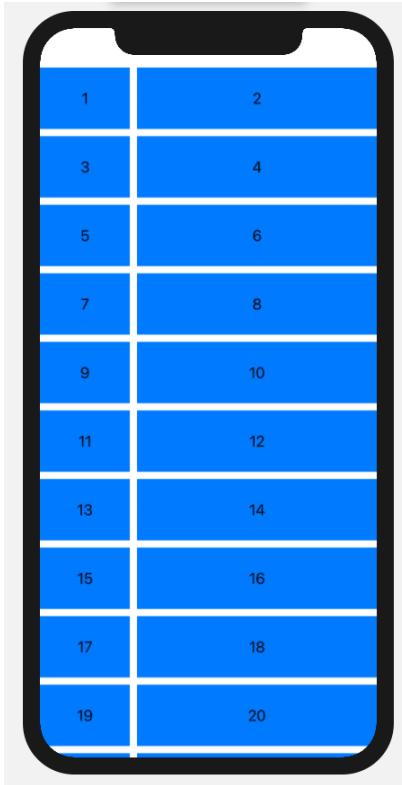
- Mit *LazyVGrid* und *LazyHGrid* können Raster-Layouts beschrieben
- «Lazy»: Es wird nur soviel Inhalt wie nötig geladen
  - *ForEach* wird nur teilweise ausgeführt
  - Andere Views wie *List* laden zwar immer den ganzen Inhalt, sind aber trotzdem sehr effizient
- Layout abhängig von den *GridItems*, siehe nächste Slide...

```
static let data = (1...1_000_000_000)

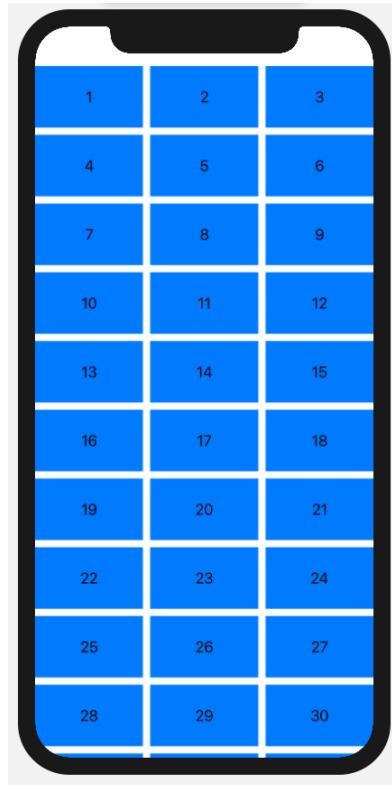
static let items: [GridItem] = [⋮]

static var previews: some View {
 Group {
 ScrollView {
 LazyVGrid(columns: items) {
 ForEach(data, id: \.self) { item in
 Center {
 Text("\(item)")
 }
 .padding().background(Color.blue)
 }
 }
 }
 ScrollView(.horizontal) {
 LazyHGrid(rows: items) {
 ForEach(data, id: \.self) { item in
 Center {
 Text("\(item)")
 }
 .padding().background(Color.blue)
 }
 }
 }
 }
}
```

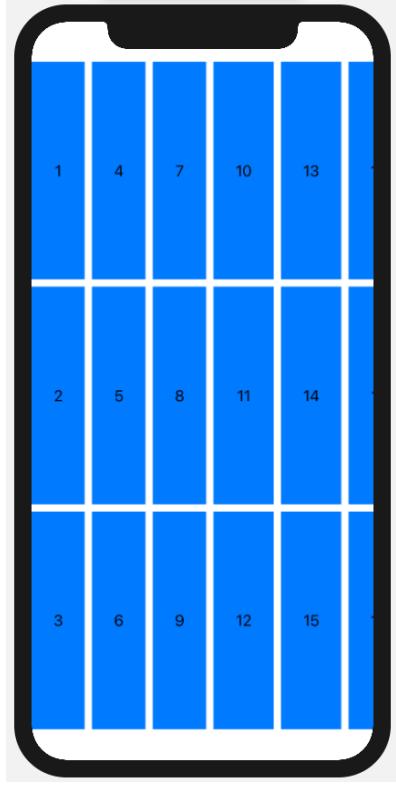
# Grids, Teil 2



```
static let items: [GridItem] = [
 GridItem(.fixed(100)),
 GridItem(.flexible())
]
```



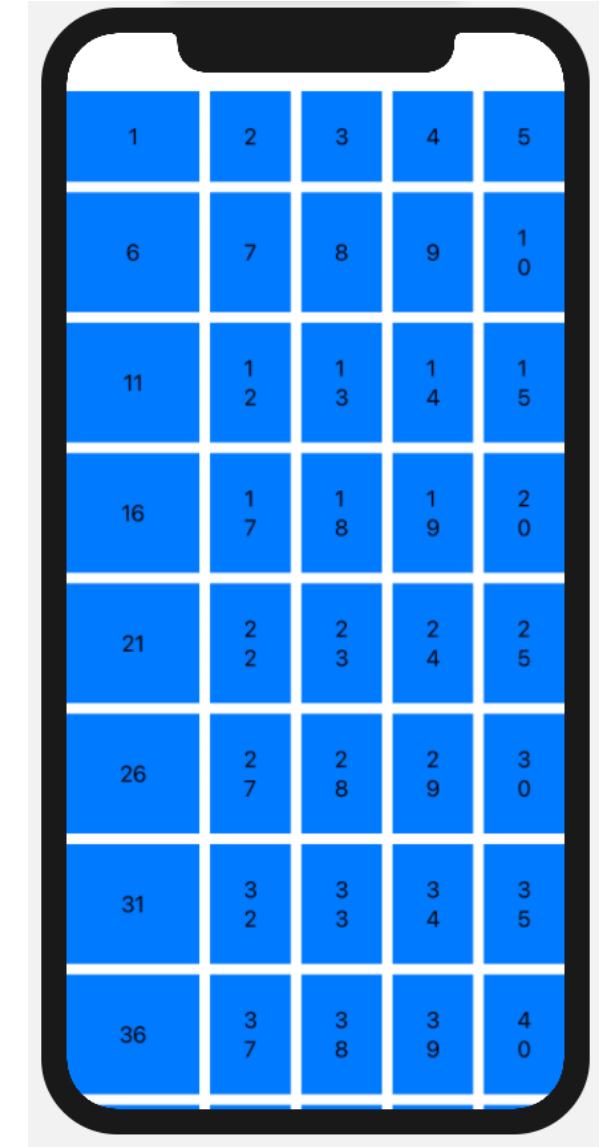
```
static let items: [GridItem] = [
 GridItem(.flexible()),
 GridItem(.flexible()),
 GridItem(.flexible())
]
```



# Grids, Teil 3

- *.adaptive GridItems* werden durch mehrere *.flexible* ersetzt, sodass die Mindestgrösse nicht unterschritten wird

```
static let items: [GridItem] = [
 GridItem(.fixed(100)),
 GridItem(.adaptive(minimum: 50))
]
```



# Layout-Protokoll

- Durch Implementation vom **Layout**-Protokoll können beliebige Layouts erstellt werden (z.B. Subviews in einem Kreis positionieren)

```
struct BasicVStack: Layout {
 func sizeThatFits(
 proposal: ProposedViewSize,
 subviews: Subviews,
 cache: inout ())
) -> CGSize {
 // Calculate and return the size of the layout container.
 }

 func placeSubviews(
 in bounds: CGRect,
 proposal: ProposedViewSize,
 subviews: Subviews,
 cache: inout ())
 {
 // Tell each subview where to appear.
 }
}
```

# State und Bindings

Programmieren für iOS



# Was ist erlaubt? Was ist korrekt?



```
struct ContentView: View {
 var text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```



```
struct ContentView: View {
 @State var text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```



```
struct ContentView: View {
 let text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```



```
struct ContentView: View {
 @State let text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```



# Was ist erlaubt? Was ist korrekt?



```
struct ContentView: View {
 var text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```

Cannot assign to  
property: 'self' is  
immutable



```
struct ContentView: View {
 @State var text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```



```
struct ContentView: View {
 let text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```

Cannot assign to  
property: 'text' is a  
'let' constant



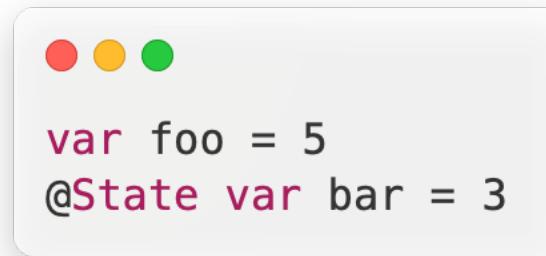
```
struct ContentView: View {
 @State let text = "foo"
 var body: some View {
 Text(text)
 .onAppear {
 text = "bar"
 }
 }
}
```

Property wrapper  
can only be applied  
to a 'var'

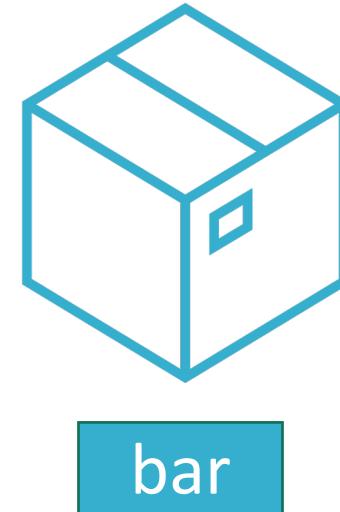
# State

- Alle Variablen, die den Zustand einer View beschreiben, werden mit **@State** annotiert
- Wenn der State ändert, wird die View automatisch aktualisiert → eine wichtige Fehlerquelle fällt weg
- Auch wenn State-Properties normal verwendet werden können, kann es hilfreich sein, sich diese als Box mit Inhalt vorzustellen

5  
foo

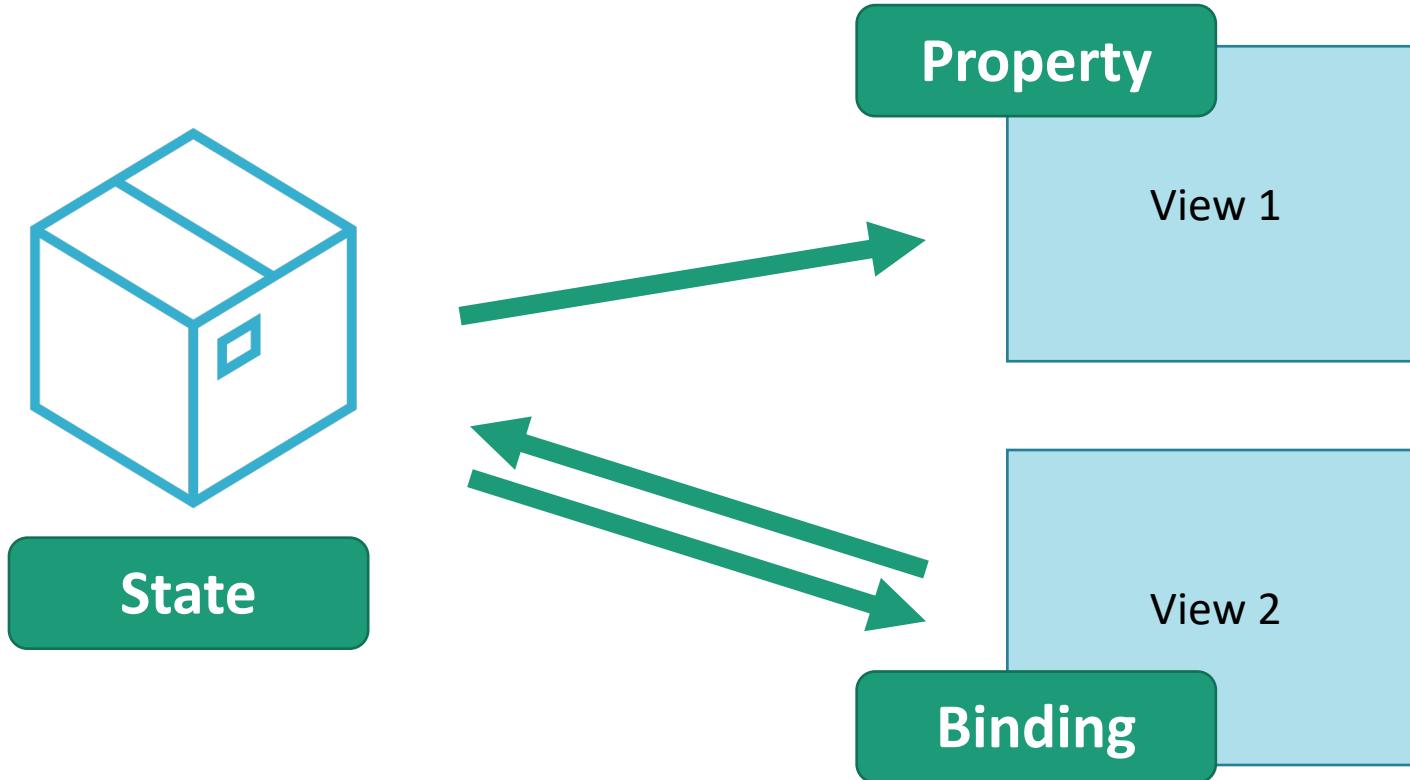


var foo = 5  
@State var bar = 3



# Binding

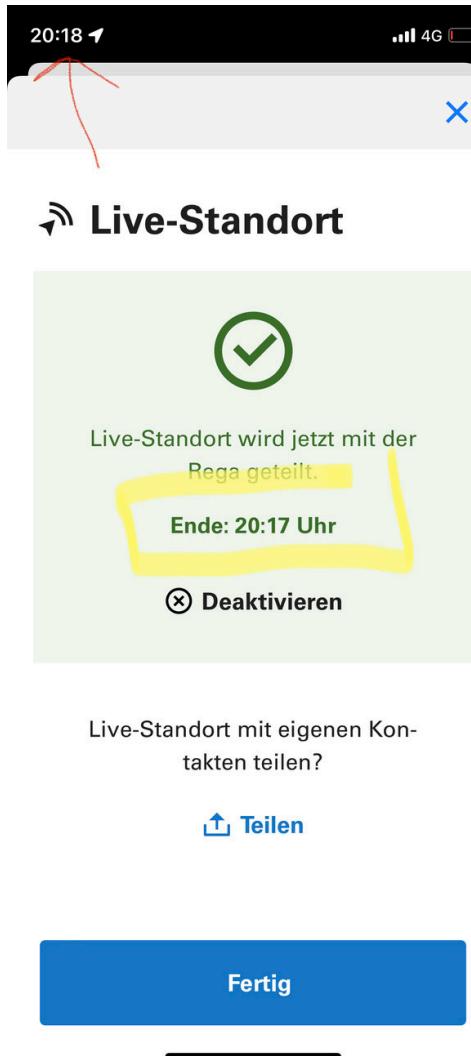
- Eine View kann den aktuellen State einer anderen View als Property übergeben oder mit einem **Binding** den Zugriff auf die Box teilen.



# Bindings vs. Referenzen

- Wieso nicht einfach Klassen verwenden, die kann man ja auch referenzieren?
- System kann Änderungen am Inhalt der Box besser beobachten und darauf reagieren
- View-Structs dürfen komplett neu initialisiert werden, solange alle Boxen wieder mit dem richtigen Inhalt befüllt werden
- Explizites Hervorheben der „**Source of Truth**“

# Praxis-Beispiel: UIKit-Bug



- Beta-Version der Rega App: Standort-Sharing wird automatisch deaktiviert. Falls der User dann gerade auf dem Teilen-Screen ist, wird das UI nicht aktualisiert.
- Ursache: Der UI-Zustand (Teilen-Screen anzeigen) ist nicht direkt vom Logik-Zustand abhängig. In der imperativen UIKit-Welt braucht es expliziten Code, um das UI zu aktualisieren.
- Bei komplexen Projekten ist es oft schwierig, das UI für jedes Szenario aktuell zu halten.
- SwiftUI garantiert automatisch Konsistenz, solange die Source of Truth eindeutig ist.

# Animationen

Programmieren für iOS



# Animationen

- Animationen stellen den bewegten Übergang zwischen zwei States des User Interfaces dar, sowohl aus Nutzersicht (Wahrnehmung) als auch technisch (@State).

ubique  Apps & Technology

- Grundregel: „Es sett nid chlopfe“ – Soweit möglich keine Änderungen am UI ohne Animation.
  - Wenige bewusste Ausnahmen, z.B. Tabs, Touch-States.

# Implizite/explizite Animation

- SwiftUI: Animation von ViewModifiers mit dynamischen Parametern
  - Implizit: **animation()** Modifier animiert eine View, sobald der angegebene "value" ändert
  - Explizit: **withAnimation()** Funktion animiert Views ohne implizite Animation

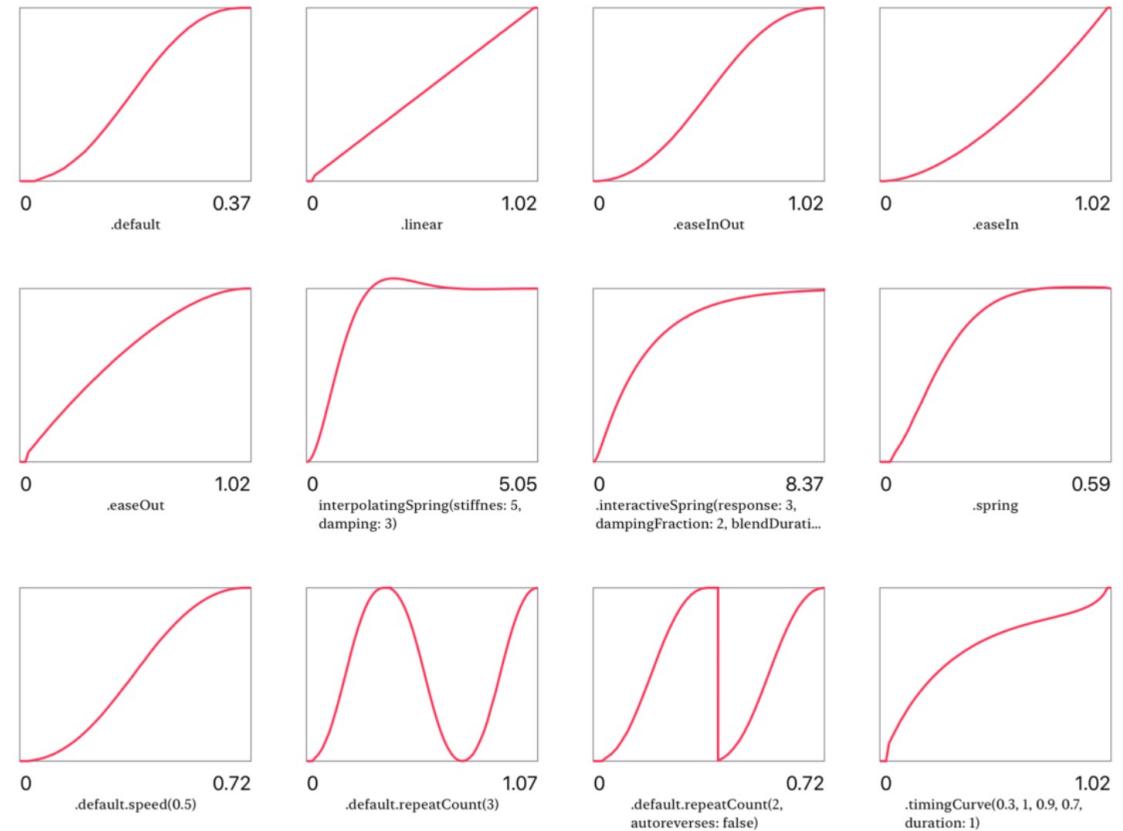
```
@State var scale = 1.0
var body: some View {
 Button("Tap me") {
 scale += 1
 }.scaleEffect(scale)
 .animation(.default, value: scale)
}
```

```
@State var scale = 1.0
var body: some View {
 Button("Tap me") {
 withAnimation(.default) {
 scale += 1
 }
 }.scaleEffect(scale)
}
```

# Das Animation struct

- `.default` kann durch eine anderen Animations-Typ ersetzt werden, um die Interpolations-Funktion zu ändern

`Animation.default.speed(1),`  
`Animation.easeInOut.delay(1),`  
`Animation.linear.repeatCount(2),`  
`Animation.spring().repeatForever(autoreverses: true)`



## Footnotes

- Die Spring-Animation orientiert sich an einer physikalischen Feder und fühlt sich deshalb für den User relativ natürlich an. Die Animation wird deshalb auch gerne verwendet, wenn man nicht das typische Hin-und-her-schwingen Verhalten möchte.

# Nicht alles kann animiert werden

- Einige Modifiers, z.B. `.font` unterstützen (noch?) keine Animationen
- Mit Protokoll `Animatable` trotzdem möglich
  - Getter/setter für Double-Interpolationswert

# Transition

- Problem: Verschwindet oder erscheint eine View, z.B. wegen einem `if` oder `switch`, gibt es keinen Endzustand, zu dem animiert werden könnte
- Lösung: Mit dem Transition-Modifier angeben, was passieren soll
  - Beispiel-Code: Die Text-View soll von unten hochgeschoben und eingeblendet werden

```
struct ContentView: View {
 @State private var showDetails = false

 var body: some View {
 VStack {
 Button("Tap Me") {
 withAnimation {
 showDetails.toggle()
 }
 }

 if showDetails {
 Text("I am details.")
 .transition(.move(edge: .bottom)
 .combined(with: .opacity))
 }
 }
 }
}
```

# Identity

- SwiftUI basiert grundsätzlich nicht auf „Tree Diffing“, d.h. es werden nicht zwei View-Bodies verglichen und Unterschiede erkannt
- Stattdessen wird die View als Konstrukt mit allen Möglichkeiten erfasst. Beim Wechsel im if-Statement wird lediglich ausgetauscht, welche Subview angezeigt wird
  - Beispiel: `_ConditionalContent`

```
struct ContentView: View {
 var body: some View {
 VStack {
 if Bool.random() {
 Text("HSLU").foregroundColor(.red)
 }
 else {
 Text("HSLU").foregroundColor(.blue)
 }
 }.onTapGesture {
 print(type(of: self.body))
 }
 }
 // ModifiedContent<
 // VStack<_ConditionalContent<Text, Text>>,
 // AddGestureModifier<_EndedGesture<TapGesture>>
 // >
```

# Identity und Animationen

- Nur bei gleicher Identity kann SwiftUI die Änderungen richtig animieren

```
var body: some View {
 if large {
 Text("HSLU").scaleEffect(2)
 }
 else {
 Text("HSLU")
 }
}
```

SwiftUI sieht zwei unterschiedliche Text-Views, die eingeblendet werden können.

```
var body: some View {
 Text("HSLU").scaleEffect(large ? 2 : 1)
}
```

SwiftUI sieht eine Text-View und animiert den Scale-Effekt

# TapGesture

- Der Modifier **.gesture()** fügt eine Geste zu einer View hinzu
- Die Closure `onEnded` wird ausgeführt, sobald die Geste erkannt wird

```
struct ContentView: View {
 @State var rotation = 0
 var body: some View {
 Text("Hello, world!")
 .rotationEffect(Angle(degrees: rotation))
 .gesture(TapGesture().onEnded {
 rotation += 45
 })
 }
}
```

Hello, world!

# Viele weitere Gestures

```
let doubleTapGesture = TapGesture(count: 2).onEnded {
 print("Tapped twice")
}

let dragGesture = DragGesture().onChanged { value in
 print("Dragged \(value.translation)")
}

let longPressGesture = LongPressGesture(minimumDuration: 2.0).onEnded { _ in
 print("Pressed 2 seconds")
}

let rotationGesture = RotationGesture().onChanged { angle in
 print("Rotated \(angle)")
}

let magnificationGesture = MagnificationGesture().onChanged { factor in
 print("Magnify \(factor)")
}
```

# ID-Modifier

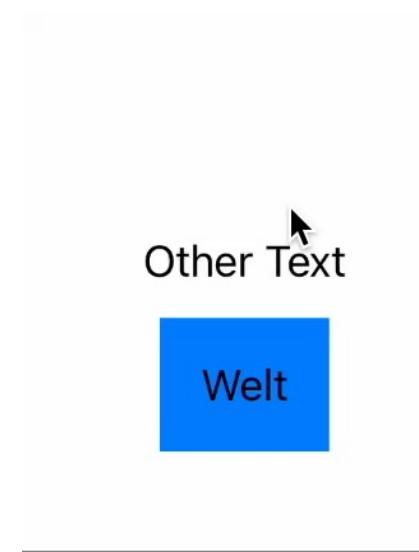
- Es gibt einen Modifier `.id()`, damit kann die Identity einer View explizit neu gesetzt werden
- Zwei unterschiedliche Views mit gleichem ID-Modifier sind aber immer noch zwei unterschiedliche Views
  - Beispiel recht: ConditionalContent-View mit zwei IDView

```
var body: some View {
 if large {
 Text("HSLU").scaleEffect(2)
 .id("hslu-label")
 }
 else {
 Text("HSLU")
 .id("hslu-label")
 }
}
```

# matchedGeometryEffect

- Mit `.matchedGeometryEffect` wird eine View beim Verschwinden so verzerrt, dass das Frame mit einer neuen View mit gleicher ID übereinstimmt
- Das Verhalten entspricht somit eher einer Transition als einer Animation

```
●●●
if large {
 Text("Hallo").padding().background(Circle().foregroundColor(.red))
 .matchedGeometryEffect(id: "label", in: geometry)
}
Text("Other Text")
if !large {
 Text("Welt").padding().background(Color.blue)
 .matchedGeometryEffect(id: "label", in: geometry)
}
```



# SwiftUI & Swift

## Programmieren für iOS



# Swift

- Swift ist Open Source: <https://github.com/apple/swift>
- Community diskutiert Weiterentwicklung aktiv, Apple redet natürlich viel mit
  - <https://github.com/apple/swift-evolution>
  - Einige Proposals auf den nächsten Slides verlinkt
- SwiftUI ist leider Closed Source

# Blocks & Closures: Motivation

- Grundidee: Funktionen als Argument („Code as Data“)
  - Erlauben "funktional[er]e" Programmierung
  - Funktionen sind „first class citizens“, d.h. es gibt einen **Datentyp** dafür
  - Wie Lambda seit Java 8:
    - Bsp.: (int x, int y) -> { return x+y; };
    - Siehe HSLU-Modul PCP (Programming Concepts & Paradigms) ☺
  - In andern Sprachen: Closures oder Blocks (Python, Ruby, Scala, ObjC, ...)
- Entsprechender Datentyp in:
  - Kompaktes Bsp.:

```
let oneFrom = { $0 - 1 }
let result = oneFrom(77)
print("result = \(result)")
```

# Swift: Closures

- Allgemeine Form siehe rechts oben

- Bsp.:

```
let oneFrom: (Int) -> Int = {
 (anInt: Int) -> Int in
 return anInt - 1
}
```

```
{ (parameters) -> return type in
 statements
}
```

- Aufruf: oneFrom(42)
  - D.h. Syntax analog zur Syntax von Funktionen, einfach ohne „func“ ☺
  - Bsp.:

```
func oneFrom(aInt: Int) -> Int {
 return aInt - 1
}
```

# Beispiel: Closures Anwendung

- Deklaration wie „normale“ Variable mit <name>: <Typ>
  - „Code as Data“ ☺

```
let myClosure: (String) -> Bool = {
 (s: String) -> Bool in
 return s.characters.count <= 3
}
```

- Anwendung von unserem Closure-Ausdruck:

```
let text = "swift"
if myClosure(text) {
 print("\(text)' is too short")
} else {
 print("\(text)' is long enough")
} // prints "'swift' is long enough"
```

# Closures: Abkürzungen

- Langer, expliziter Closure-Ausdruck:

```
let c1 : (Int) -> Int = { (i : Int) -> Int in return i - 1 }
```

- Ohne Typ-Angabe (d.h. inkl. Typinferenz):

```
let c2 = { (i : Int) -> Int in return i - 1 }
```

- Implicit return bei nur einer Anweisung:

```
let c3 = { (i : Int) -> Int in i - 1 }
```

- Argumente von Closures können automatisch als \$0, \$1, \$2, usw. („Shorthand“) verwendet werden:

```
let c4 = { $0 - 1 }
```

# Operator Funktionen

- Kompakte Closures wie gesehen mit „Shorthand Argument Names“:  
`names.sorted(by: { $0 > $1 })`
- Q: Geht's noch kürzer?
- A: Ja, mit Operator Funktionen, z.B.:

- Voraussetzung: Entsprechende Operator Funktion (kann auch selber implementiert sein)

```
names.sorted(by: >)
```

# Trailing Closures

- Idee: Closure als letztes Funktionsargument wird bei Funktionsaufruf als Codeblock nachgeliefert
  - Beispiel-Deklaration (ganz normale Closure):

```
func myTrailingClosureFunc(i: Int, closure: () -> Void) {
 closure()
 print(i)
}
```

- Aufruf mit „Trailing Closure“ (Hallo Lesbarkeit?!):

```
myTrailingClosureFunc(i: 7) {
 print("trailing closure impl :-) ")
}
```

# Closures sind „capturing“

Closures in Swift sind „capturing“ (wie Blöcke in ObjC oder Lambdas in Java), d.h. Variablen und Konstanten vom umgebenden Kontext sind bekannt und können verwendet werden

- Werte müssen **nicht** konstant (resp. „effectively final“) sein wie in Java, sondern können verändert werden. Code-Bsp:

```
var c = 77
let myClosure = {
 () -> Void in c = 123 // ok
}
```

- D.h. der Kontext von einer Closure muss in Swift erhalten bleiben, da Variablen verwendet werden können

# Escaping Closures

- Closures als Funktionsargumente sind per Default „noescaping“, d.h. sie können nach dem Verlassen der Funktion NICHT aufgerufen werden
- Deklaration: Escaping Closures
  - Closure darf auch ausserhalb der Funktion aufgerufen werden. Bsp.:

```
var closure: (() -> Void)?
func someFunctionWithEscapingClosure(callback: @escaping () -> Void) {
 closure = callback
}
```

# Trailing Closures und optionale Argumente

- Hierarchischer Aufbau mit { und } :
  - Initializer mit Trailing Closures

```
HStack {
 Text("Placeholder")
}
```

==

```
HStack(content: {
 Text("Placeholder")
})
```

- Sehr viele Argumente haben einen Default-Value

```
init(alignment: VerticalAlignment = .center, spacing: CGFloat? = nil
```

# Generics

- Bei Generics bestimmt der Caller die Typen (mit Unterstützung vom Compiler)

```
func makeEmptyArray<T>() -> T {
 return Array<T>()
}

let arr: [Int] = makeEmptyArray()
```

# Implicit returns

- Funktionen mit einem einzelnen Statement brauchen ab Swift 5.1 kein Return-Statement mehr

```
func double(_ value: Int) → Int {
 value * 2
}
```

```
var constant: Int {
 42
}
```

<https://github.com/apple/swift-evolution/blob/master/proposals/0255-omit-return.md>

# ResultBuilder / ViewBuilder

- Problem: View-Initializer haben oft viele Argumente
- Lösung: Builder-Pattern: Resultat Schritt für Schritt erstellen

```
 VStack
{
 Text("1")
 Text("2")
}

VStack {
 ViewBuilder.buildBlock(Text("1"), Text("2"))
}
```

<https://github.com/apple/swift-evolution/blob/9992cf3c11c2d5e0ea20bee98657d93902d5b174/proposals/XXXX-function-builders.md>

# Eigene Result-Builders

- Mit `@resultBuilder` können auch eigene Builders definiert werden



```
@resultBuilder
struct MapLayersBuilder {
 static func buildBlock() -> [any MapLayer] { [] }

 static func buildBlock(_ layers: [any MapLayer]...) -> [any MapLayer] {
 Array(layers.joined())
 }

 static func buildOptional(_ layer: [any MapLayer]?) -> [any MapLayer] {
 layer ?? []
 }
}

...
```



```
var body: some View {
 return MapView {
 Layer(.luftbild)
 Layer(.wanderrouten)
 }
}
```

# SwiftUI ohne Syntax Sugar

```
struct ContentView: View {
 var body: HStack<TupleView<(Text, Text)>> {
 return HStack(alignment: VerticalAlignment.center, spacing: 0, content: {
 return ViewBuilder.buildBlock(Text("1"), Text("2"))
 })
 }
}
```

statt

```
struct ContentView: View {
 var body: some View {
 HStack {
 Text("1")
 Text("2")
 }
 }
}
```

# Fazit: SwiftUI ist Swift

- Gleicher Compiler verarbeitet Views und „klassischen“ Swift-Code
  - Performance wird bei beidem und für beides gemeinsam optimiert
- Nicht auf Runtime-Parsing ausgelegt
- Breakpoints können wie in normalem Swift-Code gesetzt werden

# Ausblick Übung 3

- Nächste Kapitel aus „Scrumdinger“
- Fokus
  - State Management mit Objekten
  - Persistenz
  - Weitere Framework-APIs verwenden

# SwiftUI & Concurrency

Programmieren für iOS



# Data Flow – viele Möglichkeiten

- Struct
  - Enum
  - Class
  - Source of Truth
  - Read-only
  - Read-write
- 
- let/var
  - @State
  - @Binding
  - @Environment
  - @StateObject
  - @ObservedObject
  - @EnvironmentObject

# SwiftUI & Software-Architektur

- Die Wahl der richtigen Konstrukte ist entscheidend, aber oft schwierig.
- Es gibt keine offiziellen App-Architektur-Patterns zu SwiftUI
  - Aber genügend (unterschiedliche) Meinungen im Internet
- Viel wichtiger: Problem/Lösung verstehen und strukturieren können
  - Wer auf abstrakter Ebene versteht, welche Komponenten wie interagieren müssen, um ein Ziel zu erreichen, findet in der Regel auch eine gute Abbildung davon mit Swift

# POP

- Sehr viele Framework-Methoden von Swift und SwiftUI sind mit Protokollen und teilweise Standard-Implementationen in Extensions definiert
- Diese ersetzen oft die Basis-Klassen und bieten ähnliche Möglichkeiten wie Vererbung mit mehr Flexibilität
- Apple nennt dies **Protocol-Oriented Programming** als Pendant zu Object-Oriented-Programming
- Trotzdem gilt auch hier: Die Aufteilung vom Code in Protokolle soll nicht erzwungen werden

```
protocol Animal {
 func speak()
}

extension Animal {
 func speak() {
 print("I don't know how to")
 }
}

struct Dog: Animal {
}

class Human: Animal {
 func speak() {
 // ...
 }
}
```

# User Defaults

- Letzte Übung: Speichern von Teams als JSON-Datei
- Für kleine Daten-Mengen einfacher: UserDefaults
  - z.B. Benutzereinstellungen, App-Starts
  - Erscheinen nicht bei den App-spezifischen System-Einstellungen
- Meistens genügt das Singleton **standard**
  - Weitere Initializer für geteilte Einstellungen (z.B. App und Widget)



```
var appStarts = UserDefaults.standard.integer(forKey: "app-starts")
appStarts += 1
UserDefaults.standard.set(appStarts, forKey: "app-starts")
```

# AppStorage

- UserDefaults können direkt in SwiftUI verwendet werden, man verliert aber die eindeutige Source of Truth
- **@AppStorage** erlaubt den Zugriff einfach und korrekt

```
struct ContentView: View {
 @AppStorage("counter") var counter = 1
 var body: some View {
 Stepper("Counter", value: $counter)
 }
}
```

# Setters & Getters

- Properties werden manchmal mit gettern und setttern von internem Wert abgeleitet, das kann zu viel Boilerplate-Code führen

Welchen Wert hat value?



```
private var _value: Int?

public var value: Int {
 get {
 if let _value {
 return _value
 }
 else {
 return 42
 }
 }
 set {
 if _value == nil {
 _value = newValue
 }
 }
}
```

# Property Wrapper

- Mit der `@propertyWrapper` Annotation können Getter und Setter abstrakt definiert werden
- Die Variabel kann danach normal verwendet werden, entspricht aber dem `wrappedValue`

```
class MyClass {
 @WriteOnce var myValue: Int
 func foo() {
 let n = myValue + 1
 print(n) // 43
 myValue = 50
 }
}
```

```
@propertyWrapper struct WriteOnce {

 var _value: Int?

 var wrappedValue: Int {
 get {
 if let _value {
 return _value
 }
 else {
 return 42
 }
 }
 set {
 if _value == nil {
 _value = newValue
 }
 }
 }
}
```

# Projected Value

- Mit Underscore kann auf das Konstrukt hinter dem Property Wrapper zugegriffen werden
- Falls eine Property `projectedValue` definiert ist, kann diese mit dem Dollar-Zeichen verwendet werden



```
@propertyWrapper struct WriteOnce {
 var _value: Int?

 var wrappedValue: Int { ... }

 var projectedValue: Int = 42
 var title: String = ""
}

class MyClass {
 @WriteOnce var myValue: Int

 func changeDefault() {
 _myValue.title = "foo"
 $myValue = 100
 }
}
```

# Projected Value

- Mit Underscore kann auf das Konstrukt hinter dem Property Wrapper zugegriffen werden
- Falls eine Property `projectedValue` definiert ist, kann diese mit dem Dollar-Zeichen verwendet werden

Richtig, SwiftUI verwendet  
sehr viele solche  
Property Wrappers!



```
@propertyWrapper struct WriteOnce {

 var _value: Int?

 var wrappedValue: Int { ... }

 var projectedValue: Int = 42
 var title: String = ""
}

class MyClass {
 @WriteOnce var myValue: Int

 func changeDefault() {
 _myValue.title = "foo"
 $myValue = 100
 }
}
```

# SwiftUI – Das wars!

- Es gibt natürlich sehr, sehr viele weitere Views und Modifiers
  - (Z.B. [TextField](#), [Stepper](#), [ColorPicker](#), [ProgressView](#), [blur](#), [blendMode](#), ...)
  - Learning-by-doing: Aktuell gibt es ca. 70 verschiedene Views – es genügt, diese kennen zu lernen, wenn man ein konkretes UI-Problem lösen möchte
- Custom User Interfaces: Genau so möglich
  - Einfache Views kombinieren, mehr Overlays und Stacks, weniger Forms
- Wie geht's weiter?
  - SwiftUI, falls für andere Frameworks relevant
  - Nächste Woche: UIKit
  - Heute: Swift ohne UI (Concurrency)

# Concurrency & Parallelism

Programmieren für iOS

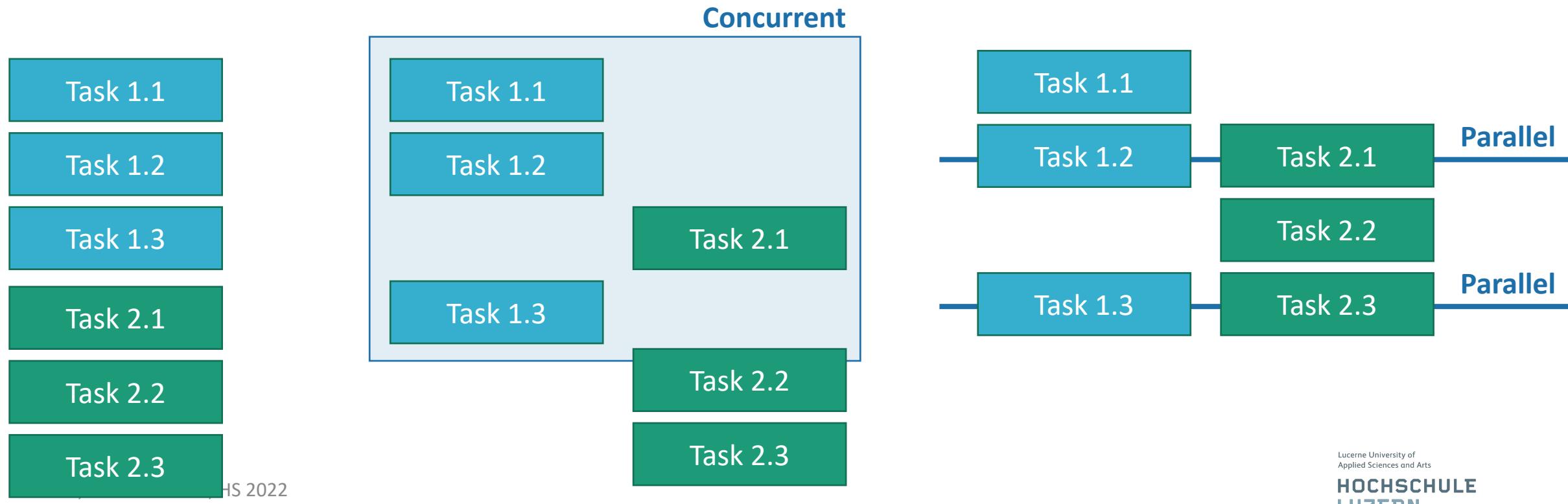


# Definitionen

- Die „**Tasks**“ (Aufgaben) einer iOS-App werden nur selten komplett seriell abgearbeitet. Zwei Begriffe sind für die Diskussion zu unterscheiden:
  - **Concurrency** (Gleichzeitigkeit, Nebenläufigkeit)
  - **Parallelism** (Parallelisierung)
- Viele Definitionen der beiden Begriffe tönen sehr ähnlich:
  - „Concurrency is about dealing with lots of things at once but parallelism is about doing lots of things at once“ (Rob Pike)

# Definitionen 2

- Ein Programm als **nebenläufig/concurrent** bezeichnet, wenn ein Tasks Fortschritte machen kann, bevor ein anderer abgeschlossen wurde.
- Die Ausführung wird als **parallel** bezeichnet, wenn mehrere Tasks im gleichen Moment laufen.
- Letzteres ist nur mit mehreren CPU-Kernen möglich.

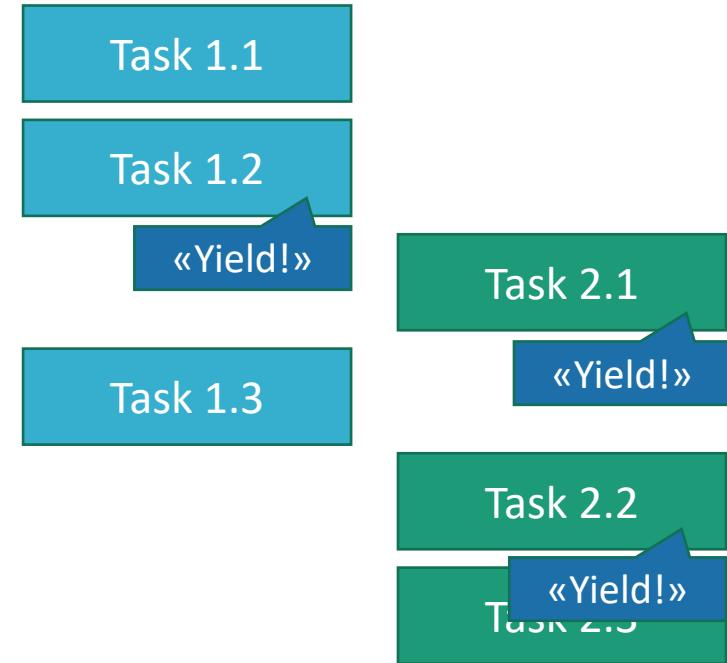


# Concurrency auf iOS

- Nebenläufigkeit ist auf iOS aus mehreren Gründen von Bedeutung:
  1. Die CPU ist nicht immer ausgelastet, das Bottleneck kann oft auch eine andere Hardware-Komponente sein.
    - Beispiele: Netzwerk-Antenne, Disk, GPS
  2. Gewisse Tasks, insbesondere UI-Updates, sollten regelmässig Fortschritte machen, damit das User Interface nicht einfriert.
  3. Ohne Nebenläufigkeit ist auch keine parallele Ausführung möglich, die modernen Multi-Core CPUs können nicht richtig genutzt werden.

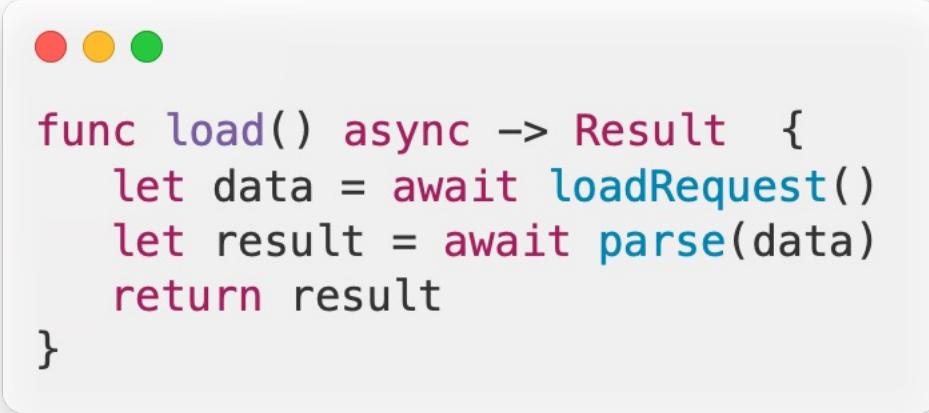
# Asynchron

- Damit Tasks nebenläufig durchgeführt, müssen Methoden asynchron ausgeführt werden können.
- Eine Methode wird als **asynchron** bezeichnet, wenn sie Möglichkeiten zum Aussetzen (**suspend**) bietet. Dies wird auch als aufgeben (**yield**) der Durchführung bezeichnet.



# Async Swift

- In Swift wird eine asynchrone Methode mit dem Keyword `async` markiert.
- Jeder Befehl, bei dem die Ausführung ausgesetzt werden kann, wird mit `await` markiert.



```
func load() async -> Result {
 let data = await loadRequest()
 let result = await parse(data)
 return result
}
```

# Parallelisierung

- Durch mehrere `await` kann eine Methode mehrmals nacheinander aussetzen. Die Reihenfolge der Durchführung bleibt aber gleich.
- In dem mehrere Variablen mit `async let` markiert werden und die Resultate mit einem `await` aggregiert werden, können mehrere Subtasks parallel ausgeführt werden.



```
let firstPhoto = await downloadPhoto(named: photoNames[0])
let secondPhoto = await downloadPhoto(named: photoNames[1])
let thirdPhoto = await downloadPhoto(named: photoNames[2])

let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```



```
async let firstPhoto = downloadPhoto(named: photoNames[0])
async let secondPhoto = downloadPhoto(named: photoNames[1])
async let thirdPhoto = downloadPhoto(named: photoNames[2])

let photos = await [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

# Networking

## Programmieren für iOS



# Networking

- Das Laden von Daten von einem Server ist eines der wichtigsten Beispiele von Nebenläufigkeit.
- Die System-API von `URLSession` bietet dafür (auch) asynchrone Methoden.

```
func loadData() async throws -> Data {
 let (data, _) = try await URLSession.shared.data(from: myUrl)
 return data
}
```

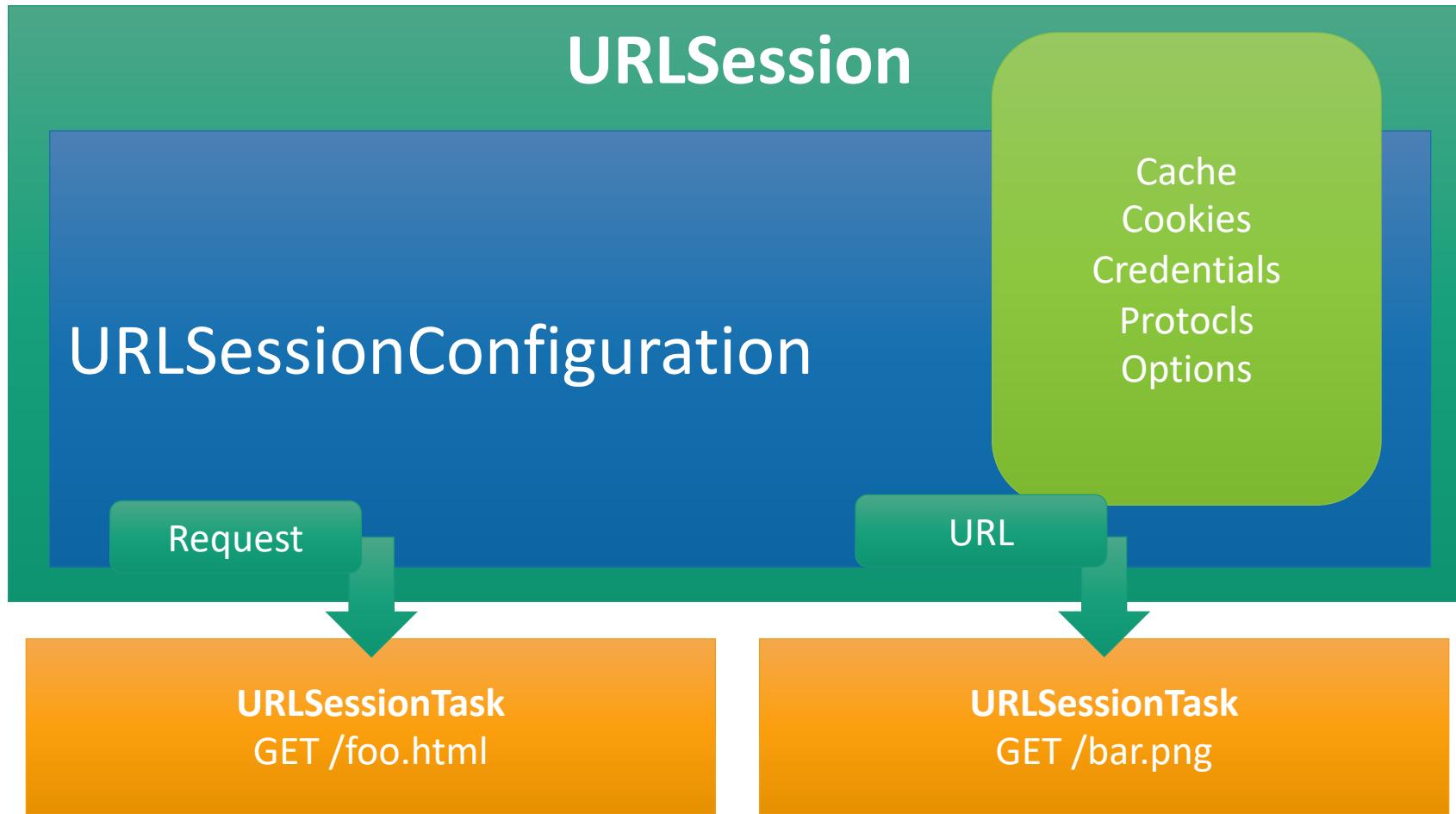
# Kommunikation mittels URLs

- URL = Uniform Resource Locator, z.B.:
  - `http://www.hslu.ch`
  - `ftp://user:password@my.ftp.server.org`
  - `file://private/var/containers/Bundle/Application/Networking.app/file.txt`
- Entsprechende Klassen und Structs in Swift
  - `URL`
  - `URLSession`
  - `URLRequest`
  - `URLResponse`
  - ...

```
let myUrl = URL(string: "https://www.hslu.ch")!

var request = URLRequest(url: myUrl)
request.httpMethod = "POST"
request.addValue("application/json", forHTTPHeaderField: "Accept")
```

# URLSession – Überblick



# JSON-Parsing mit Codable

- Das Codable-Protokoll:
  1. Codable-Typ deklarieren
  2. Mit JSONDecoder direkt Data-Objekt darauf abbilden lassen

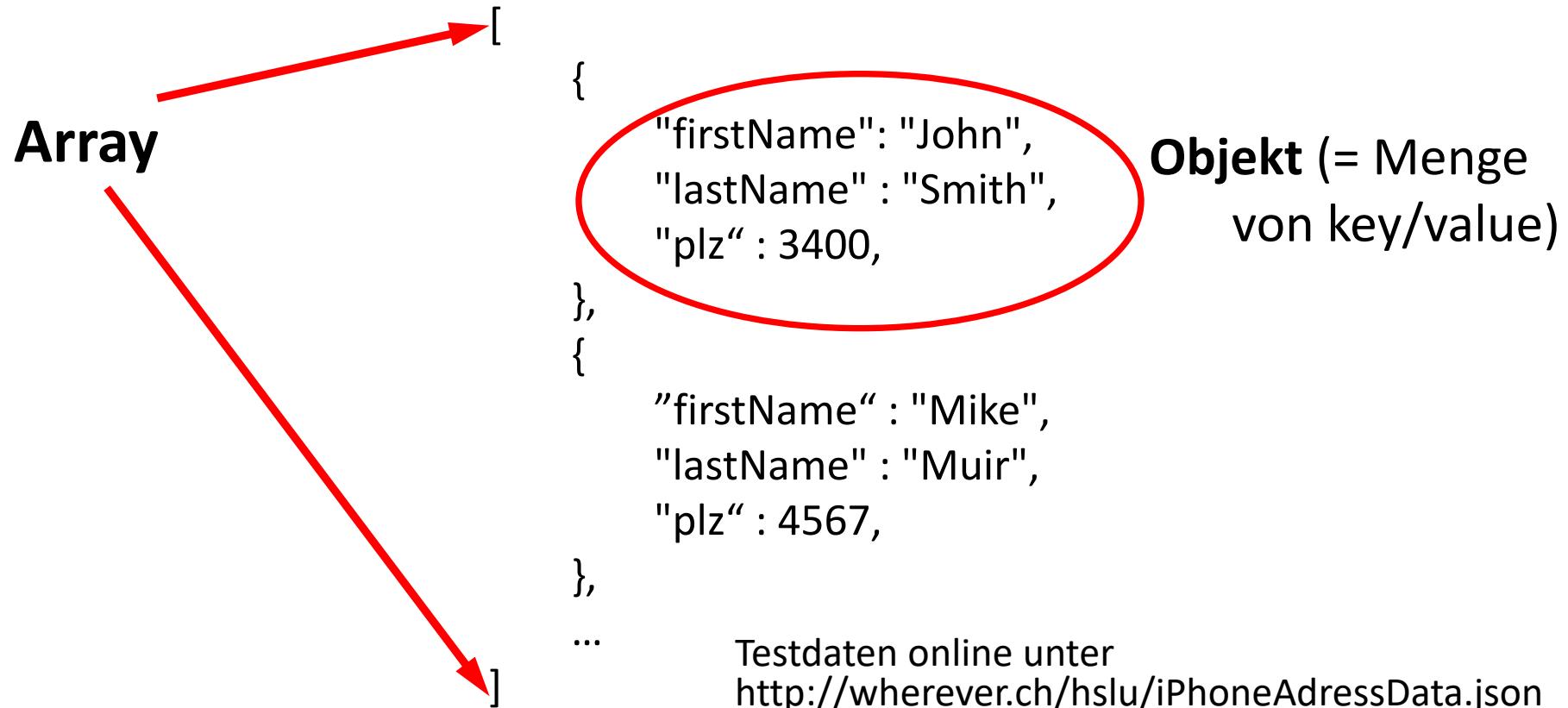
```
struct Person: Codable {
 var firstName: String
 var lastName: String
}

let model = try JSONDecoder().decode([Person].self, from: data)
```

Typ “Array<Person>”

# JSON

- JavaScript Object Notation
  - [http://de.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](http://de.wikipedia.org/wiki/JavaScript_Object_Notation)



# App Transport Security

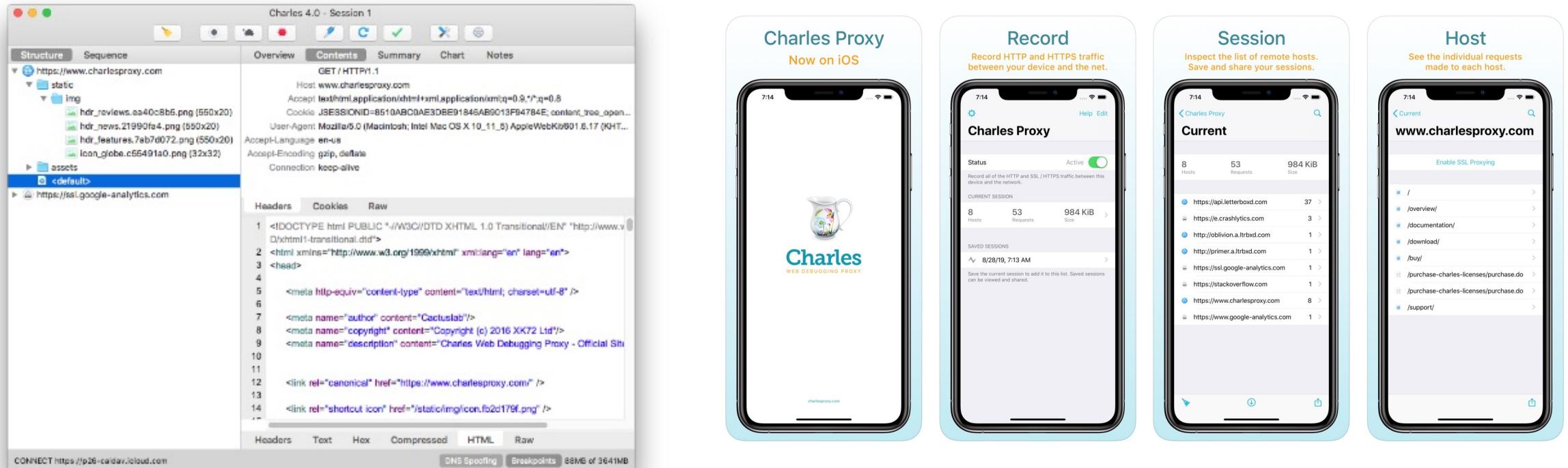
- Seit iOS 9 ist **https mit TLS 1.2 und Forward Secrecy per Default Pflicht!**

```
2015-10-22 21:01:55.079 ComAndCons[69975:5987456] App
Transport Security has blocked a cleartext HTTP (http://)
resource load since it is insecure. Temporary exceptions can
be configured via your app's Info.plist file.
```

- Falls "nur" http gewünscht: URL in Projekt-Info "genehmigen":

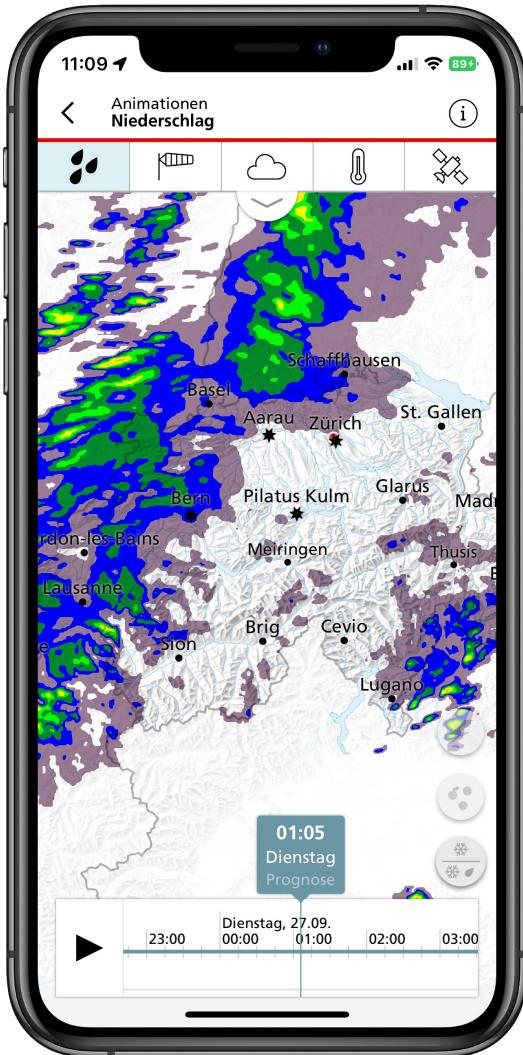
| Info                                |         |            |           |
|-------------------------------------|---------|------------|-----------|
| NSAppTransportSecurity              |         | Dictionary | (2 items) |
| NSAllowsArbitraryLoads              | Boolean | NO         |           |
| NSExceptionDomains                  |         | Dictionary | (1 item)  |
| wherever.ch                         |         | Dictionary | (1 item)  |
| NSEExceptionAllowsInsecureHTTPLoads | Boolean | YES        |           |

# Proxy



- Mit Apps wie Charles oder Proxyman Network können Requests aufzeichnen oder bearbeitet werden
- Super zum Testen von Networking (oder um APIs von anderen Apps kennenzulernen)

# Networking – Case Study



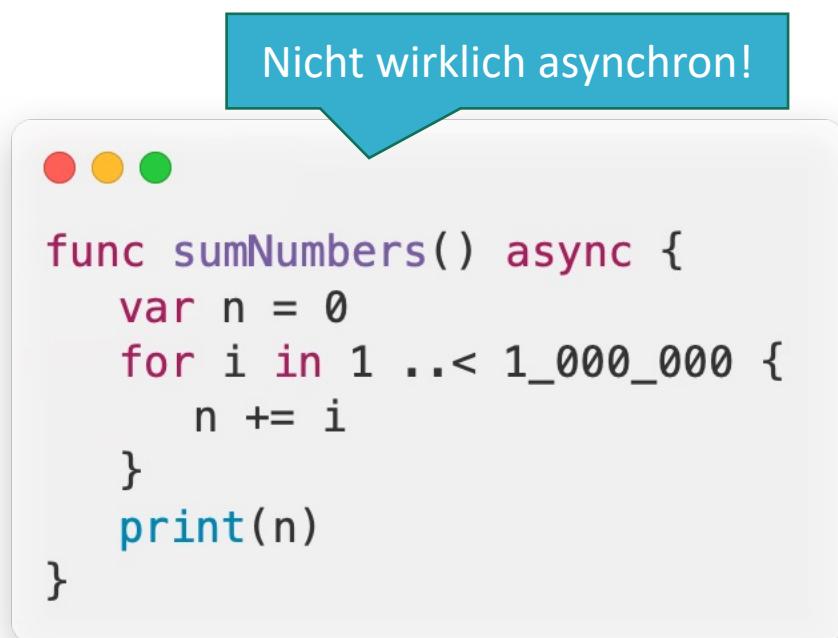
# Concurrency 2

Programmieren für iOS



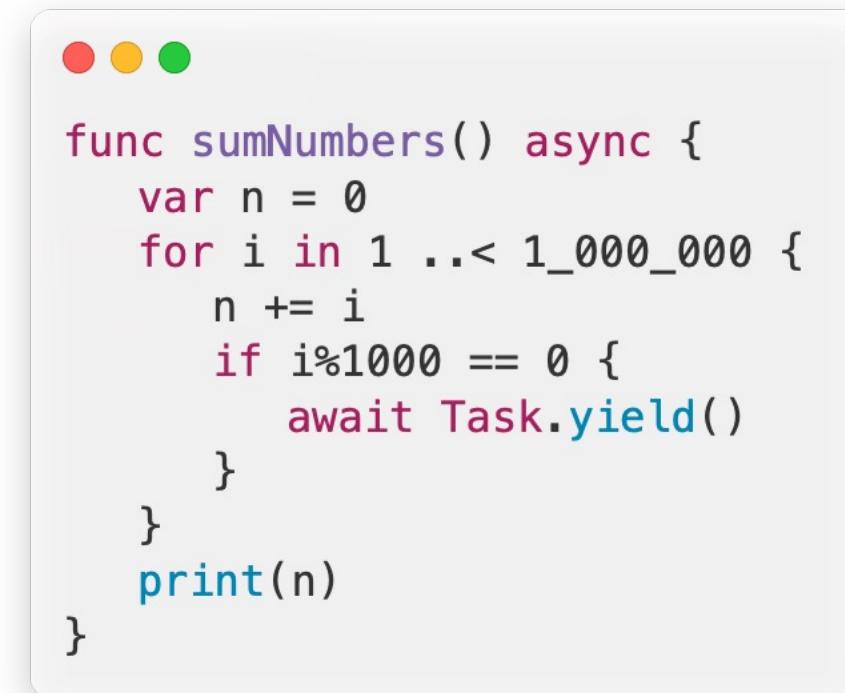
# Yield

- Mit `Task.yield()` kann explizit eine Code-Stelle als möglicher «Suspension Point» markiert werden.



Nicht wirklich asynchron!

```
func sumNumbers() async {
 var n = 0
 for i in 1 ..< 1_000_000 {
 n += i
 }
 print(n)
}
```



```
func sumNumbers() async {
 var n = 0
 for i in 1 ..< 1_000_000 {
 n += i
 if i%1000 == 0 {
 await Task.yield()
 }
 }
 print(n)
}
```

# Structured & Unstructured Concurrency

- Mit `async` und `await` wird implizit eine Hierarchie von Tasks und Subtasks definiert.
- Tasks können auch explizit mit dem Struct `Task` erstellt werden, dessen Initializer eine Closure erwartet.
- Solche Tasks müssen selbst verwaltet werden, z.B. beim Abbruch der Durchführung. Man spricht deshalb von **Unstructured Concurrency**.

```
● ○ ●
let newPhoto = // ... some photo data ...
let handle = Task {
 return await add(newPhoto, toGalleryNamed: "Spring Adventures")
}
let result = await handle.value
```

# Cancellation

- Swift Concurrency setzt auf «**kooperativen**» Abbruch einer asynchronen Methode.
- Ein Subtask erhält lediglich die Information, dass die Task-Hierarchie abgebrochen wurde und muss selbst entscheiden, ob er die Ausführung beenden möchte.



```
func sumNumbers() async {
 var n = 0
 for i in 1 ..< 1_000_000 {
 n += i
 if i%1000 == 0 {
 if Task.isCancelled {
 return
 }
 await Task.yield()
 }
 }
 print(n)
}
```

# TaskGroup

- Swift bietet mit `withTaskGroup` eine API, um viele ähnliche Subtasks zu erstellen

```
let images = await withTaskGroup(of: UIImage.self, returning: [UIImage].self) { taskGroup in
 for title in titles {
 taskGroup.addTask {
 return await self.loadImage(title: title)
 }
 }

 var images: [UIImage] = []
 for await image in taskGroup {
 images.append(image)
 }
 return images
}
```

# TaskGroup

- Dynamische Anzahl an Child Tasks erstellen:

```
let images = await withTaskGroup(of: UIImage.self, returning: [UIImage].self) { taskGroup in
 for title in titles {
 taskGroup.addTask {
 return await self.loadImage(title: title)
 }
 }
}

var images: [UIImage] = []
for await image in taskGroup {
 images.append(image)
}
return images
}
```

Tasks erstellen

# TaskGroup

- Dynamische Anzahl an Child Tasks erstellen:

```
let images = await withTaskGroup(of: UIImage.self, returning: [UIImage].self) { taskGroup in
 for title in titles {
 taskGroup.addTask {
 return await self.loadImage(title: title)
 }
 }
}

var images: [UIImage] = []
for await image in taskGroup {
 images.append(image)
}
return images
}
```

Resultate aggregieren

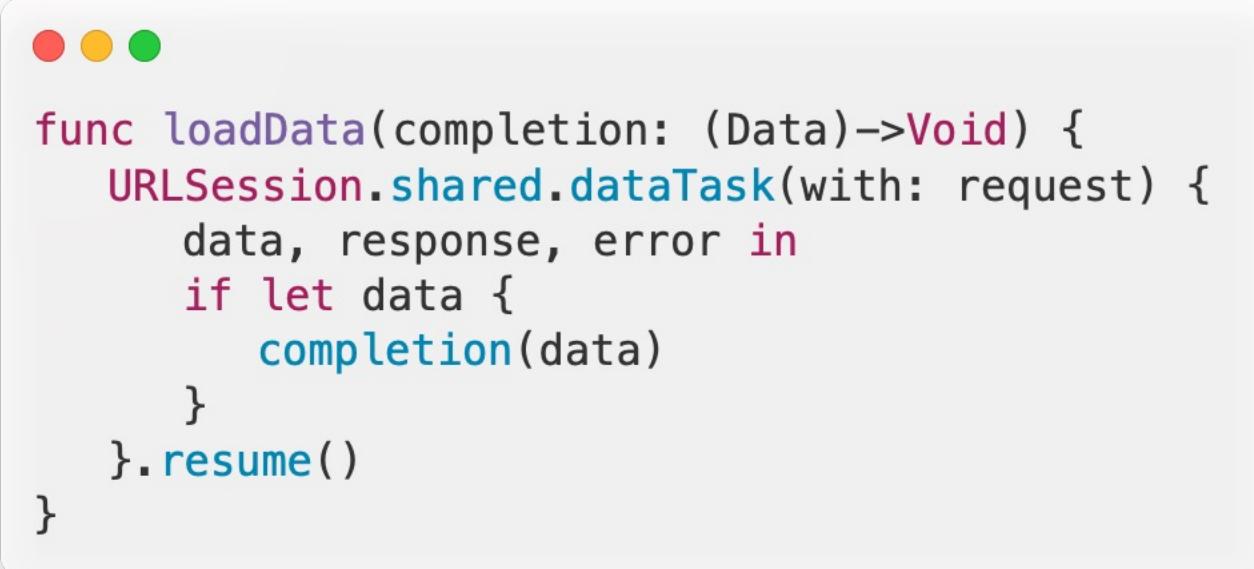
# TaskGroup, zweites Beispiel

- Falls ein Task Fehler werfen kann, muss withThrowingTaskGroup verwendet werden
- TaskGroup ohne Rückgabe möglich

```
await withThrowingTaskGroup(of: URLResponse.self, body: { taskGroup in
 for request in postRequests {
 taskGroup.addTask {
 let (_, response) = try await URLSession.shared.data(for: request)
 return response
 }
 }
})
```

# Ältere APIs

- Letzte Übung: Asynchroner Code ist mit Completion-Closures auch ohne `async/await` möglich.
- Das Ende einer Methode entspricht einem impliziten Yield, andere Tasks können ausgeführt werden, bevor die Completion-Closure läuft.
- Viele APIs wurden entwickelt, bevor Swift mit Concurrency erweitert wurde.



A screenshot of the Xcode code editor showing a completion closure example. The code uses URLSession.shared.dataTask to download data, then handles the result in a completion block. The code is as follows:

```
func loadData(completion: (Data) -> Void) {
 URLSession.shared.dataTask(with: request) {
 data, response, error in
 if let data = data {
 completion(data)
 }
 }.resume()
}
```

# Threads

- Threads bilden die Grundlage zur parallelen Ausführung von Code.
- Swift bietet eine API zum Erstellen oder Beenden von Threads.
- Es ist praktisch unmöglich zu entscheiden, ob weitere Threads die Performance erhöhen, da viele Faktoren wie CPU-Auslastung oder Gerät-Typ einen Einfluss haben.
- Stattdessen bietet iOS verschiedene «**Queues**», denen Code übergeben werden kann. Das System verwaltet eine «**Thread Pool**» mit mehreren Threads, welche die Queues abarbeiten.



```
Thread.detachNewThread {
 print("Run on new Thread")
}
```

# DispatchQueue

- Aus Übung: DispatchQueues sind Objekte, denen mit `sync` oder `async` Closures übergeben werden können.
- Queue: Das System garantiert, dass eine Closure nicht vor einer anderen startet, die später hinzugefügt wurde.



```
let myQueue = DispatchQueue(label: "ch.hslu.ios.MyQueue")
myQueue.async {
 print("Do some work here.")
}
print("The first block may or may not have run.")
myQueue.sync {
 print("Do some more work here.")
}
print("Both blocks have completed.")
```

**Do some work here.**  
**The first block may or may not have run.**  
**Do some more work here.**  
**Both blocks have completed.**

# DispatchQueue 2

- DispatchQueues sind standardmäßig seriell, mit dem Attribut `.concurrent` kann eine Queue erstellt werden, die mehrere Closures gleichzeitig auf mehreren Threads starten kann.
- Trotzdem gilt: Die Reihenfolge der Tasks bleibt erhalten.
- Das System verwaltet mehrere globale Queues, für einfache Nebenläufigkeit werden meistens diese verwendet.



```
let queue1 = DispatchQueue(label: "concurrent-queue", attributes: .concurrent)

let queue2 = DispatchQueue.global()
let queue3 = DispatchQueue.global(qos: .userInitiated)
```

# OperationQueue

- Zusätzliche API basierend auf DispatchQueues. Queues werden nicht mit Closures gefüllt, sondern mit Operation-Objekten.
- Dadurch bieten sich weitere Möglichkeiten wie Abhängigkeiten zwischen Operations, Prioritäten oder Cancellation

```
let queue = OperationQueue()
let op1 = MyCustomOperation()
let op2 = BlockOperation {
 print("op1 done")
}
op2.addDependency(op1)
queue.addOperation(op1)
queue.addOperation(op2)
```

# Race Conditions & Thread Safety

- Als **Race Condition** (Wettlaufsituation) wird der Umstand bezeichnet, dass das Ergebnis einer Operation vom zeitlichen Verhalten von Einzeloperationen abhängt.
- Diese treten insbesondere auf, wenn mehrere Threads den gleichen Speicher bearbeiten.
- Als **Thread Safety** (Threadsicherheit) wird die Eigenschaft einer Softwarekomponente bezeichnet, gemeinsamer Speicher nur so zu verändert, dass keine ungewünschte oder unerwartete Effekte auftreten.
- Swift ist im Allgemeinen **nicht** thread-safe.

# Race Condition Beispiel

Read «counter»

R = 0

Addition R + 1

R = 1

Write «counter»

Counter = 1

Read «counter»

R = 1

Addition R + 1

R = 2

Write «counter»

Counter = 2



```
var counter = 0
for _ in 0 ..< 2 {
 counter += 1
}
print(counter)
```

# Race Condition Beispiel

Read «counter»  
R = 0

Addition R + 1  
R = 1

Write «counter»  
Counter = 1

Read «counter»  
R = 0

Addition R + 1  
R = 1

Write «counter»  
Counter = 1



```
var counter = 0
DispatchQueue.concurrentPerform(iterations: 2) { _ in
 counter += 1
}
print(counter)
```

Mehrere Iteration vom  
DispatchQueues auf den  
Thread-Pool verteilen

# Race Condition Crash

- Race Conditions bei Datenstrukturen wie Arrays oder Maps führen oft nicht nur zu falschen Ergebnissen, sondern zum Crash der App



The screenshot shows a macOS application window with three colored window controls (red, yellow, green) at the top left. Inside the window, there is a code editor displaying Swift code. The code defines a mutable map and performs 1000 concurrent iterations on a dispatch queue. In each iteration, it sets the value of the map at index `i` to `i`. A red rectangular highlight covers the line of code `map[i] = i` and the error message that follows. The error message is: "Thread 12: EXC\_BAD\_ACCESS (code=1, address=0x800000000000...)".

```
var map: [Int: Int] = [:]
DispatchQueue.concurrentPerform(iterations: 1000) { i in
 map[i] = i Thread 12: EXC_BAD_ACCESS (code=1, address=0x800000000000...
}
```

# MainQueue

- SwiftUI (und UIKit) sind ebenfalls nicht thread-safe. Um Inkonsistenzen zu verhindern, wird erwartet, dass das User Interface nur auf dem ersten Thread, der mit der App gestartet wurde, ausgeführt werden.
- Dieser wird als **Main Thread** bezeichnet. DispatchQueue und OperationQueue bieten beide eine globale Instanz, die alle Tasks auf diesem Thread ausführen.



```
DispatchQueue.main.sync {
 print(Thread.isMainThread) // true
}
OperationQueue.main.addOperation {
 print(Thread.isMainThread) // true
}
```

# Actors

- Um Thread Safety auch bei nebenläufiger und paralleler Ausführung zu ermöglichen, setzt Swift auf das **Aktorenmodell**.
  - Dieses wurde erstmals 1973 von Carl Hewitt beschrieben und wird auch in anderen Sprachen wie D oder Scala verwendet.
- In Swift werden Actors mit einem neuen vierten Typ erstellt und sind somit weder Klassen noch Structs.

enum

struct

class

actor

# Actor

- Actors sind wie Klassen **Referenztypen**, und unterstützen alle von Klassen und Structs Elemente wie Properties, Initializers, Methoden.
- Sie erlauben (noch?) keine Vererbung.
- Der Compiler garantiert, dass immer nur ein Task mit veränderbarem Speicher eines Actors interagiert.
- Aufrufe von aussen müssen deshalb mit **await** markiert werden und können somit aussetzen.



```
actor TemperatureLogger {
 let label: String
 var measurements: [Int]
 private(set) var max: Int

 init(label: String, measurement: Int) {
 self.label = label
 self.measurements = [measurement]
 self.max = measurement
 }
}
```



```
let logger = TemperatureLogger(label: "Outdoors", measurement: 25)
print(await logger.max)
```

# MainActor

- Ein **Task** wird in der Regel von einem Actor ausgeführt.
- Der spezielle globale Actor **MainActor** führt alle Tasks auf dem Main Thread aus. Somit können alle Tasks, die von diesem Actor ausgeführt werden, das User Interface ohne Risiko verändern.
- Mit der Annotation **@MainActor** kann eine Klasse, ein Struct oder einzelne Properties oder Methoden dem MainActor zugewiesen werden.



```
@MainActor
class MyModel: ObservableObject {
 @Published var title = "Foo"
 @Published var counter = 0
}

MainActor.run {
 someOtherModel.show = true
}
```

# Tasks & SwiftUI

- SwiftUI bietet einen speziellen Modifier `.task`, der die Closure vor dem Erscheinen asynchron aufruft.
- Der Task wird auch abgebrochen, wenn die View frühzeitig entfernt wird.

```
● ○ ●

class Loader: ObservableObject {
 @Published var title = ""
 func load() async {
 // ...
 }
}

struct TestView: View {
 @StateObject var loader = Loader()
 var body: some View {
 Text(loader.title)
 .task {
 await loader.load()
 }
 }
}
```

# Tasks & SwiftUI 2

- Mit dem Task-Initializer können asynchrone Tasks z.B. nach Interaktionen ausgeführt werden.
- Ein Task wird normalerweise von dem Actor ausgeführt, von dem er auch erstellt wurde
  - Im ersten Beispiel: MainActor
  - Tasks werden somit normalerweise nebenläufig, aber nur auf dem Main Thread ausgeführt
- Mit `Task.detached` wird ein Task unabhängig vom aktuellen Actor erstellt, somit können Subtasks auch parallel ausgeführt werden

```
struct TestView: View {
 @StateObject var loader = Loader()
 var body: some View {
 VStack {
 Button("Reload") {
 Task {
 await loader.load()
 }
 }
 Button("Reload") {
 Task.detached {
 await loader.load()
 }
 }
 }
 }
}
```

# Wofür eigene Actors einsetzen?

- Wie bei der Unterscheidung von Klassen und Structs, gibt es auch zum Einsatz vom Actors keine allgemeingültige Formel
- In Betracht zu ziehen sind Actors dann, wenn Objekte von mehreren Threads referenziert und bearbeitet werden sollen

```
public actor TrackerManager: ObservableObject {
 static let shared = TrackerManager()

 @MainActor @Published
 private(set) var syncError: Error?

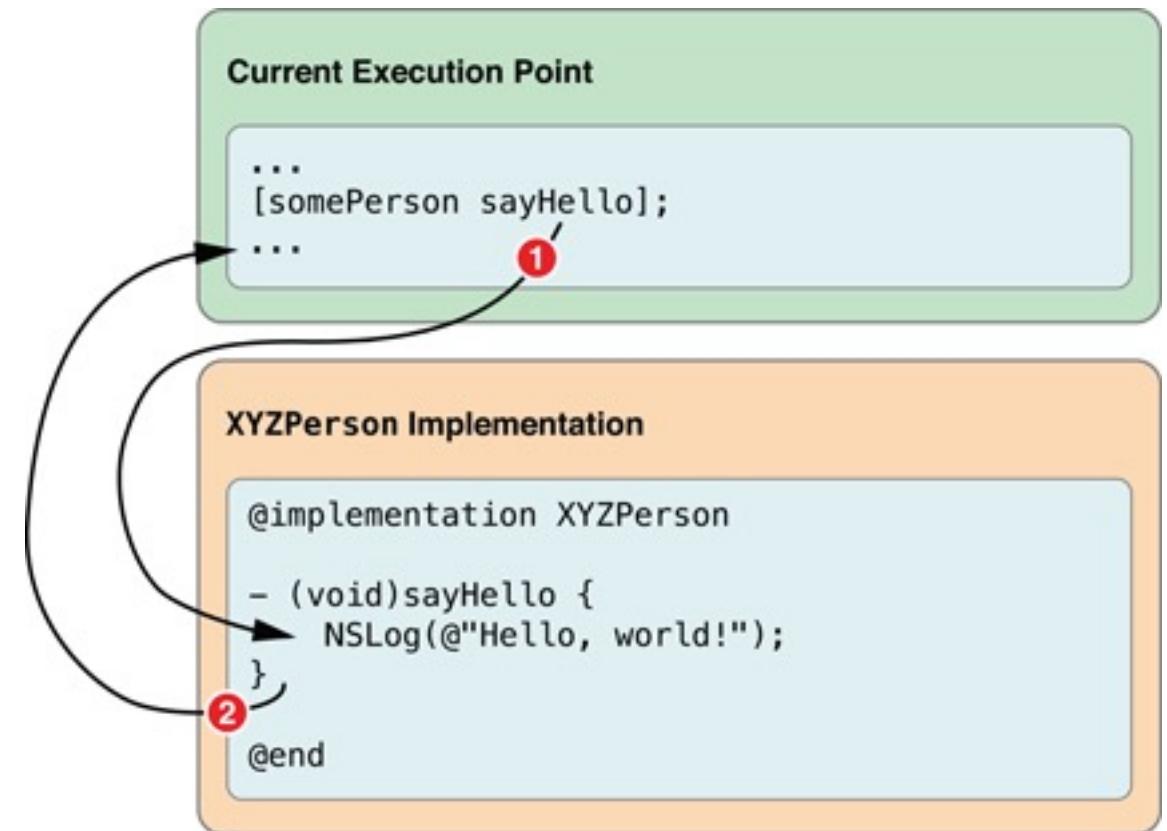
 @MainActor
 private func setSyncError(_ error: Error?) {
 self.syncError = error
 }

 var trackers: [Trackers]

 func syncTracker(_ tracker: Tracker) async {
 // ...
 }
}
```

# Message Passing

- Vor langer Zeit (Objective-C): Jeder Methodenaufruf eine "Nachricht", die an ein anderes Objekt geschickt wird
- Actor Model: Nachrichten werden verschickt («Message Passing») und abgearbeitet, es gibt keinen Shared State



<https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html>

# Sendable

- Nur Objekte, die **sendable** sind, können von einem Actor zu einem anderen übermittelt werden.
- In der Regel trifft das auf alle Structs zu, die aus einfachen Typen kombiniert wurden und auf Klassen, die nur let-Properties enthalten.
- Für komplexe, dynamische Inhalte muss das Sendable-Protokoll erfüllt werden. (schauen wir nicht genauer an.)

# Distributed Actors

- Die Swift-Community arbeitet zur Zeit an einer Erweiterung, mit der Actors auf einem Cluster eingesetzt werden können, um Arbeit auf mehreren Servern korrekt verteilen zu können.
- Mit dem Proposal wird, ein neues Keyword eingeführt, voraussichtlich **distributed actor**.

```
●●●
distributed actor Worker {
 var data: SomeData
 distributed func work(item: String) -> WorkItem.Result {
 // ...
 }
}

let first = ActorSystem("FirstNode") { settings in
 settings.cluster.enable(host: "127.0.0.1", port: 7337)
}
let second = ActorSystem("SecondNode") { settings in
 settings.cluster.enable(host: "127.0.0.1", port: 8228)
}

first.cluster.join(host: "127.0.0.1", port: 8228)
```

# Server-side Swift

- Swift ist eine allgemeine Programmiersprache und kann nicht nur für die App-Entwicklung eingesetzt werden.
- Vorteile von Swift als Server-Sprache
  - Kleiner Memory-Footprint
  - Schneller Startup
  - Gute und deterministische Performance
- SwiftNIO
  - Event-driven Low-Level Framework
  - <https://github.com/apple/swift-nio>
- Vapor / Perfect / Kitura
  - Opensource High-Level Frameworks für HTTP Server

# Vapor

- Verwendet SwiftNIO
- Bietet Ökosystem für Datenbank-Interaktionen, HTML-Output Verschlüsselungen und andere häufige Server-Aufgaben.
  - Kann natürlich (noch?) nicht mit Java/Spring oder anderen populären Server-Frameworks mithalten.
- Versucht stark, neuste Swift-Syntax zu ermöglichen, z.B. `async/await`.
  - Kitura setzt eher auf eine stabilere Entwicklung.



```
import Vapor

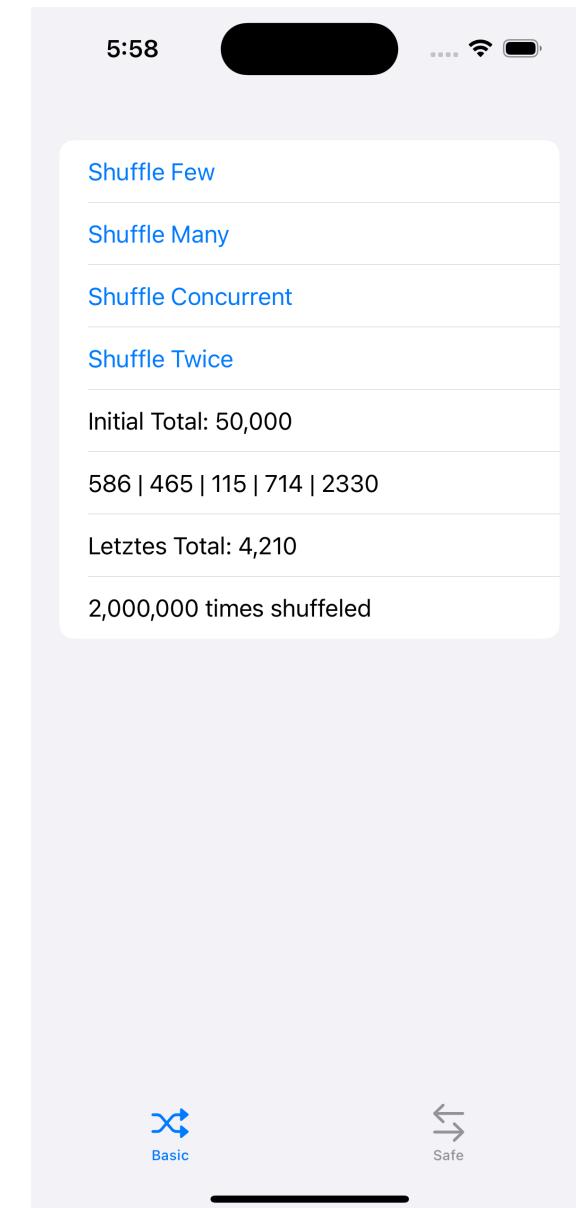
let app = try Application(.detect())
defer { app.shutdown() }

app.get("hello") { req in
 return "Hello, world."
}

try app.run()
```

# Ausblick: Übung 4

- Arbeiten mit Concurrency und Parallelism
- Achtung: Keine Schritt-für-Schritt Anleitung mehr



# Danke!

## Programmieren für iOS



# Programmieren fürs iOS

5. UIKit: Interface Builder,  
ViewControllers & SwiftUI



# Inhalt "UIKit"

- Interface Builder, inkl. Verbindung Layout-Code
- ViewController
  - Content vs. Container
  - UINavigationController
  - Übergänge: modal vs. show (push)
  - Storyboards & Segues
- Interoperabilität UIKit-SwiftUI
  - UIHoistingController
  - UIViewControllerRepresentable



# Intro: UIKit vs. SwiftUI

# SW01: UIKit vs. SwiftUI



- Aktuell zwei UI-Technologien
    - UIKit: der alte **imperative** König
      - ViewControllers, Target-Action-Muster, usw.
    - SwiftUI: der junge **deklarative** Kronprinz...
      - Zukunftsträchtiges "modernes" deklaratives UI-Framework (ähnlich zu Flutter, Jetpack Compose, ...)
  - Aktuell in Transitionsphase...
- Modul behandelt beide, mehr Fokus SwiftUI ☺

# SwiftUI



SwiftUI

- Neuer deklarativer UI-Syntax
  - Seit Xcode 11 / **ab iOS 13**
- (Fast) alles passiert im Code...
  - Preview-Views

- Siehe Intro die letzten beiden Wochen! :-)

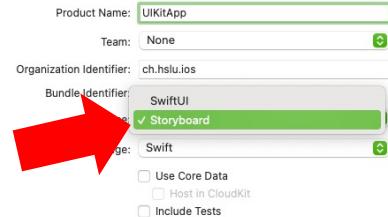
<https://developer.apple.com/xcode/swiftui/>

The screenshot shows an iPhone 12 Pro displaying a SwiftUI application. The app interface includes a navigation bar with the text "iPhone 12 Pro" and "iOS 15.0". Below the navigation bar is a table view showing lap times. The data in the table is as follows:

| Lap    | Time    | Action |
|--------|---------|--------|
| Reset  |         |        |
| Lap    |         |        |
| 4 laps |         |        |
| 0 sec  | 9:01 AM | ↻      |
| 2 sec  | 9:01 AM | ↻      |
| 3 sec  | 9:01 AM | ↻      |
| 5 sec  | 9:01 AM | ↻      |

On the left side of the screen, the corresponding SwiftUI code is visible, showing how the data is being rendered.

# Heute: UIKit / Storyboard



- Massive Auswirkungen in Bezug auf Erzeugung & Verwendung von GUI
- Wir tauchen ein in die "klassische" UIKit-Welt mit Storyboards, ViewControllers, usw.

Zwischenfazit: Die iOS-App-Welt wird umfangreicher & komplizierter!..

Fokus UIKit

# Kursinhalt (grob)

Fokus SwiftUI

1. 19.09.: Einführung, Swift Crash Course & Xcode
2. 26.09.: SwiftUI Basics, Layouting, App-Provisioning
3. 03.09.: SwiftUI: States, Bindings, Navigation
4. 10.10.: Kommunikation & Nebenläufigkeit
5. 17.10.: UIKit, ViewControllers, UIKit vs. SwiftUI
6. 24.10.: Fragmentierung, mobile Usability, Widgets
7. 31.10.: Persistenz & Unit-Tests, Property Wrappers
8. 07.11.: Memory-Management, Frameworks



# Intro Interface Builder

# Interface Builder vs. Coding

- Bisher: Views "programmieren"
  - Übung 1: 

```
let label = UILabel(frame: frame)
self.view.addSubview(label)
```

...
- Neu: Views "zeichnen"
  - Tool: Interface Builder
    - In Xcode integriert (seit Xcode 4.0)
    - Eng verknüpft mit UIKit
      - **NICHT vorgesehen für SwiftUI**

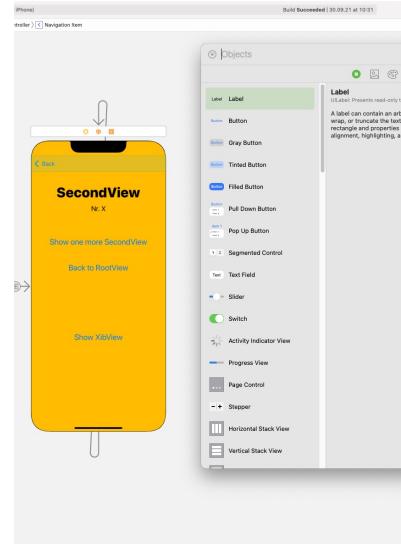
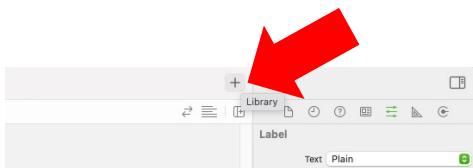


# Interface Builder

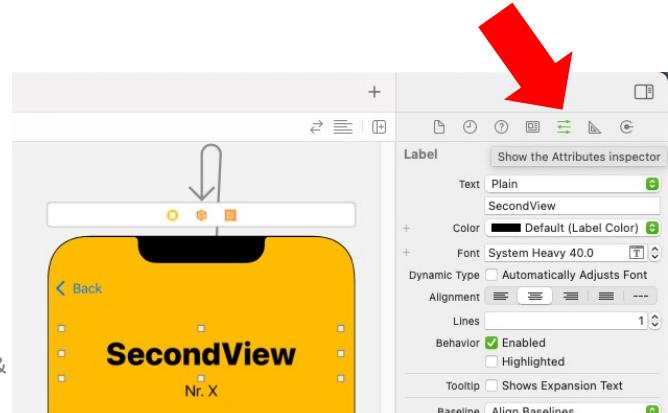
- Interface Builder = Visuelles Tool für GUIs
  - Eng verbunden mit Xcode (liest "automatisch" bestimmte Dinge aus Code, aus einzelnen Klassen)
  - Automatisches Ausrichten: "Apple-Look'n'Feel"
  - Standard-Widgets: UILabel, UIButton, UISlider, ...
  - Dateiformat: .storyboard
    - Früher primär: .nib-Datei (**NextStep Interface Builder**)
      - resp. .xib (x für xml...)
      - Gibt's immer noch, funktioniert immer noch! (Siehe später)
- Relativ einfach zu bedienen, primär graphisch ☺
  - Drag'n'drop, Ziehen, usw.

# Demo: Interface Builder

- (Object) Library
  - Zeigt die im IB verfügbaren Objekte



- Attributes Inspector
  - Attribute inspizieren und setzen



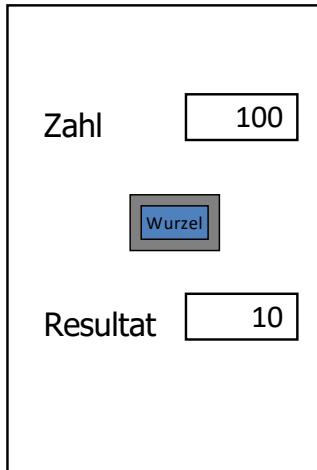


# Interface Builder: Verbindung Layout – Code

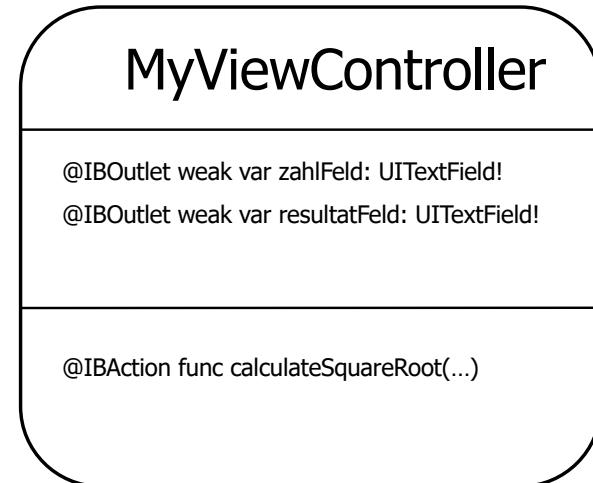
# Verbindung Layout-Code

- Wie Layout (.xib, resp. .storyboard) mit Code zusammenbringen?

\*.storyboard/\*.xib:



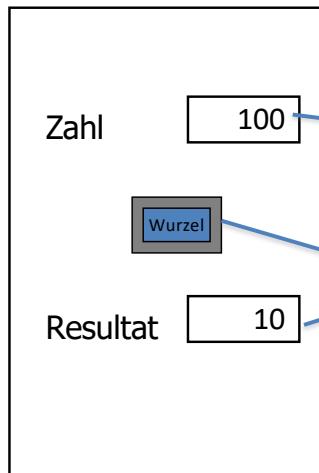
Objekt zur Laufzeit:



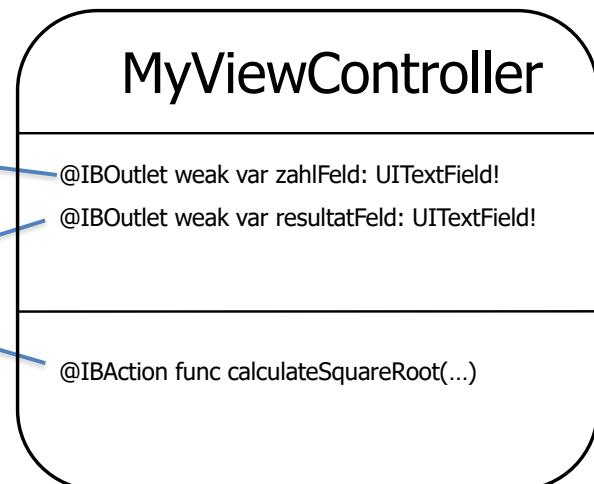
# Verbindung Layout-Code

- Wie Layout (.xib, resp. .storyboard) mit Code zusammenbringen?
- Schlüssel: **IBOutlet & IBAction**

\*.xib / \*.storyboard:



Objekt zur Laufzeit:



# IBOutlet

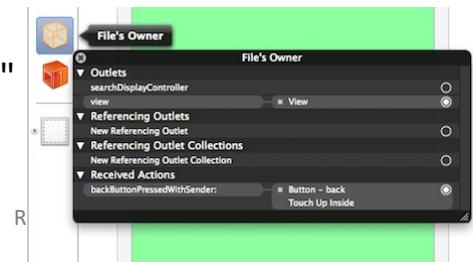
- **IBOutlet** verbinden Layout (.xib/.storyboard) mit Code
  - Verbindung ziehen (mit der Maus!) im Interface Builder
- Schlüsselwort "**IBOutlet**" bei der Deklaration
  - **@IBOutlet weak var myLabel: UILabel!**
- Enge Bindung zwischen IB und Code
  - IB merkt z.B. wenn in Xcode neue **IBOutlets** oder **IBActions** deklariert werden
  - "Mausverbindung" vom IB in den Code

# Deklaration von IBOutlets

- Im Code (`MyViewController.swift`)
  - `@IBOutlet weak var myLabel: UILabel!`
- Verbindung zwischen Code und Layout mit der Maus (Methoden "Rechtsklick" oder "ctrl")
  - Falls nicht vorhanden, wird ggf. IBOulet im Code erzeugt

# Demo: IBOutlet

- Deklaration IBOutlet im Code
- Verschiedene Varianten Maus-Verbindung
  - Maus: Ctrl-Taste oder rechter Mausklick
  - Siehe nächste Folien...
- Referenz auf ViewController?
  - .storyboard: ViewController
  - .xib: File's Owner = Platzhalter für entspr. ViewController  
(mehr zu MVC später...)
    - d.h. +- : "File's Owner = MyViewContorller"



# Mit deklariertem Outlet verbinden



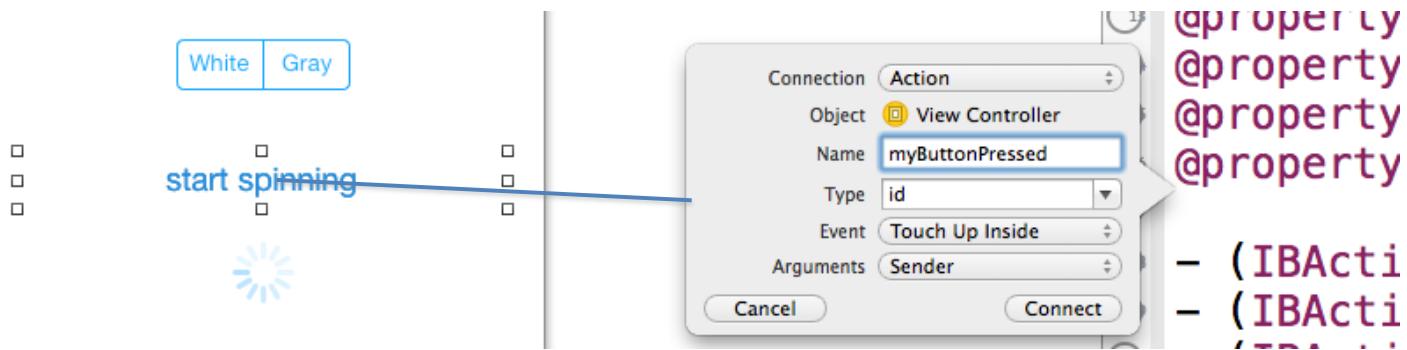
# IBAction

- Target/Action-Muster: Verbindet "Action" mit einem Ziel
  - "Action" z.B. Knopfdruck, oder neuer Wert (Methode)
  - Ziel ist ein Objekt, welches die Nachricht erhält
    - typischerweise ein ViewController (mehr dazu nächste Woche)
- Syntax (MyViewController.swift)
  - `@IBAction func sliderValueChanged(sender: UISlider)`
  - Konvention: Exakt ein Argument "sender"
    - "sender" = Auslöser der Action
- Verbindung Layout-Code mit "Maus" analog zu IBOutlet
  - Auch programmatisch möglich
    - Methode `addTarget:action:forControlEvents:` von UIControl

# Demo: IBAction

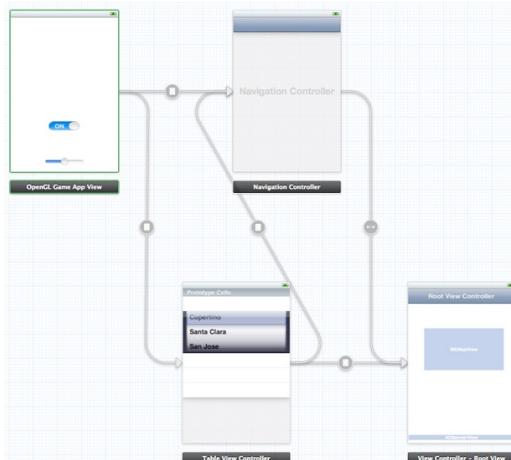
- Deklaration IBAction im Code
- Verschiedene Varianten "Maus-Verbindung"  
wie bei IBOutlet
- Target/Action live...
  - z.B. Action auf Knopfdruck

# IBAction-Methode generieren



# Ausblick: Storyboards

- Mehrere/alle Views in einer Datei, inkl. Modellierung der Übergänge: Storyboard
  - behandeln wir gleich vertiefter



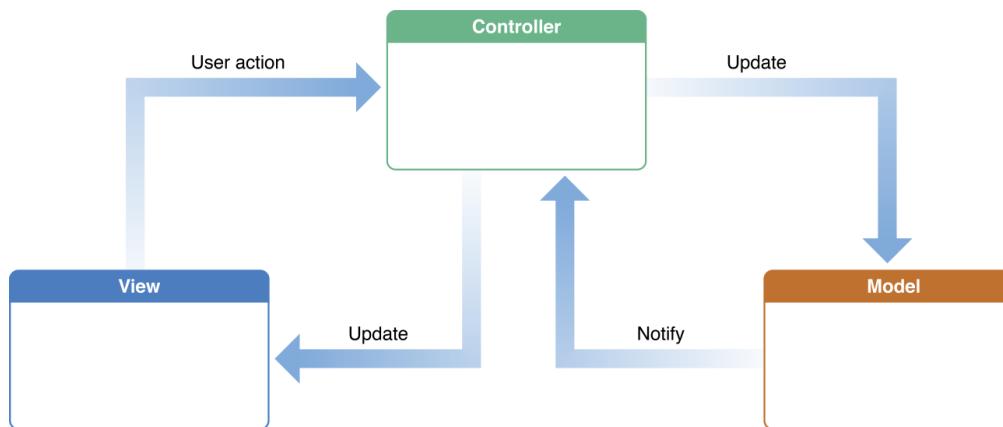
Source: <http://developer.apple.com/>



# UIKit: View- Controller

# MVC: View Controller

- Model-View-Controller Design-Muster
  - Model: "Daten"
  - View: "Ansicht"
  - Controller: "Vermittler"



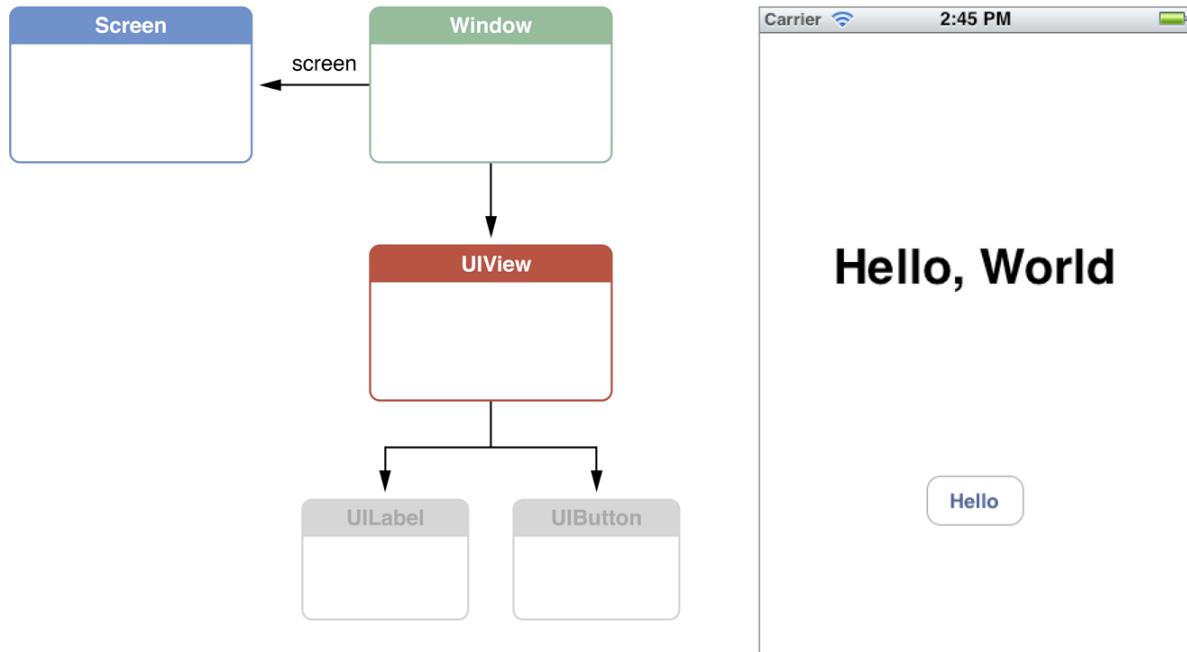
Source: <http://developer.apple.com/>

# ViewControllers sind bei UIKit zentral wichtig...

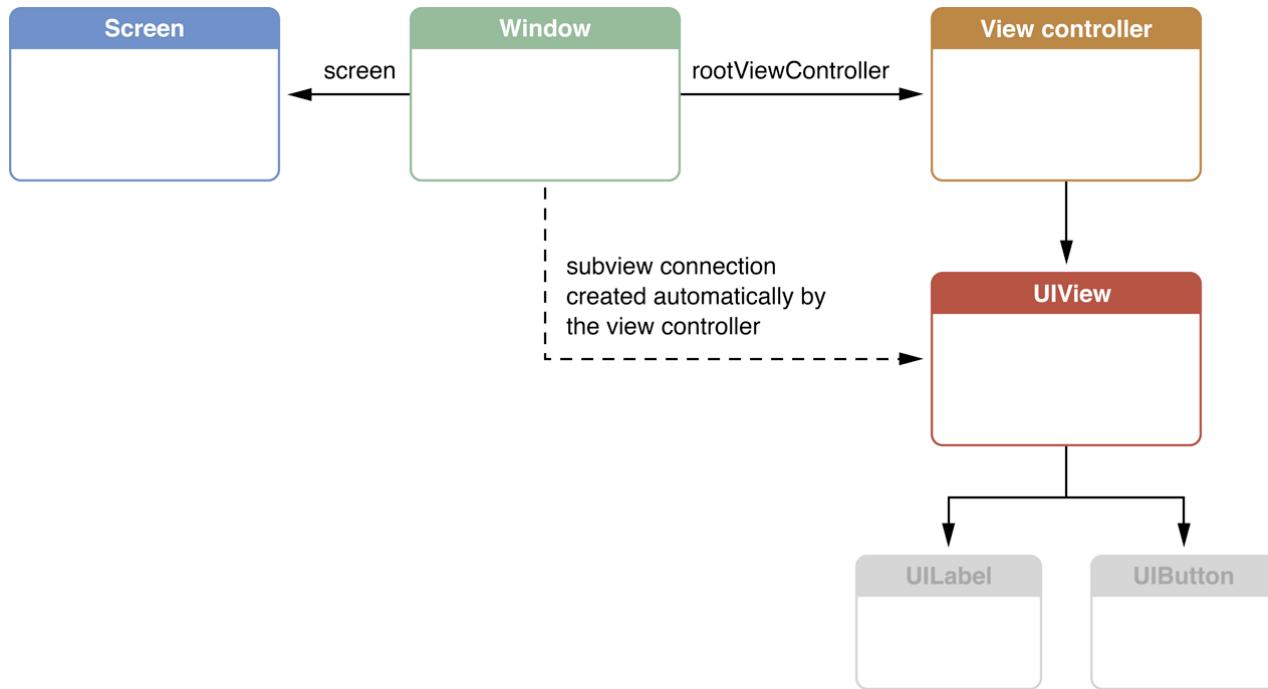
View controllers are the foundation of your app's internal structure. Every app has at least one view controller, and most apps have several. Each view controller manages a portion of your app's user interface as well as the interactions between that interface and the underlying data. View controllers also facilitate transitions between different parts of your user interface.

[https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/index.html#/apple\\_ref/doc/uid/TP40007457-CH2-SW1](https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/index.html#/apple_ref/doc/uid/TP40007457-CH2-SW1)

# Screen, Window & Views



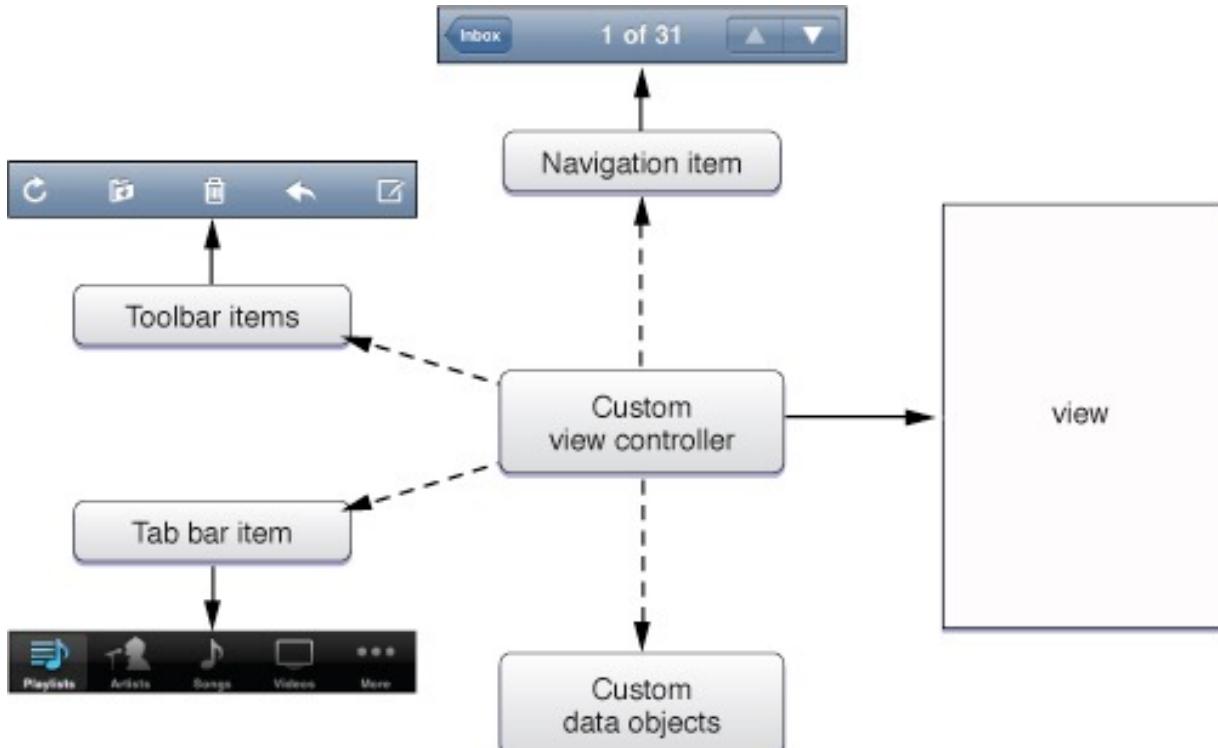
# View Controllers manage Views



# UIViewController

- "DIE" View-Controller-Klasse
  - Controller einer "Bildschirmseite" in iOS (UIKit)
- Basisklasse für alle iOS-ViewController
- Funktionalität für: Modal-View, Navigation, ...

# Anatomie eines View Controllers

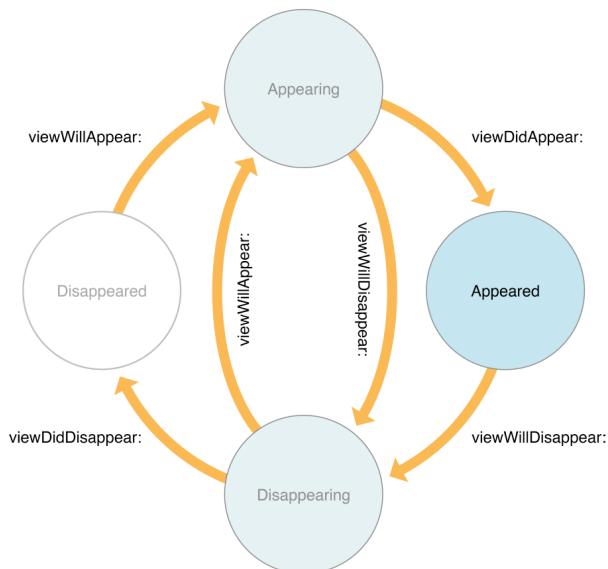


# UIViewController: einige Properties...

- `nibName`
  - kann nil sein (default: nib und Controller heissen gleich)
- `view`
  - DIE view, welcher dieser Controller steuert
- `title`
  - wichtig bei Navigation: Default für Titel NavigationBar & Back-Button
- Für ModalViews
  - `modalViewController`
  - `modalTransitionStyle`
- Für Navigation & Tabs
  - `navigationController`, `navigationItem`, `toolbarItems`, `tabBarController`, `tabBarItem`

# UIViewController: einige Methoden...

- `initWithNibName:bundle:`
  - Initialisierung aus einer Nib-Datei
  - Werden wir kaum brauchen: passiert automatisch, wenn nib und Controller gleich heissen
- `loadView`
  - 3 Optionen, siehe nächste Folie
  - Danach wird `viewDidLoad` aufgerufen
- "Vor- und Nachwarnungen" wenn View angezeigt wird / verschwindet
  - `viewWillAppear: / viewDidAppear:`
  - `viewWillDisappear: / viewDidDisappear:`
- Speicherwarnung
  - `didReceiveMemoryWarning`





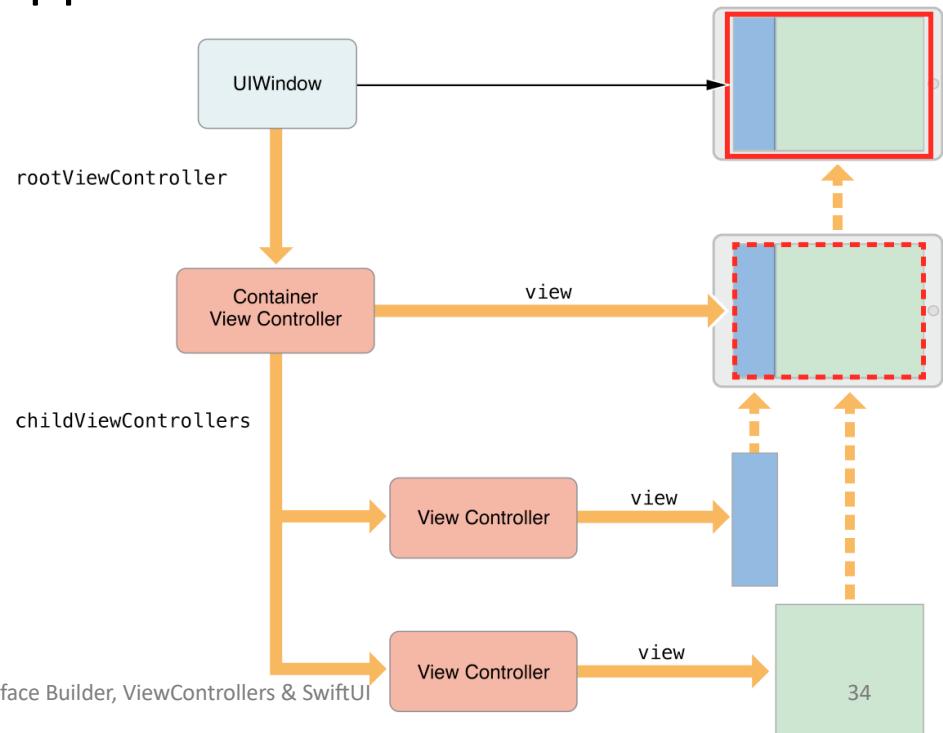
# View-Controller: Content vs. Container

# Content & Container Controller

- Es gibt zwei Arten von View Controllers
  - Content View Controller
    - Stellen Inhalt dar: "Views"
    - z.B.: UIViewController, UITableViewController
  - Container View Controllers
    - Arrangieren Inhalt von anderen ViewControllers (typischerweise Content ViewControllers)
    - z.B.: UINavigationController, UITabViewController, UISplitViewController, UIPageViewController, UIPopoverController

# Container View Controller

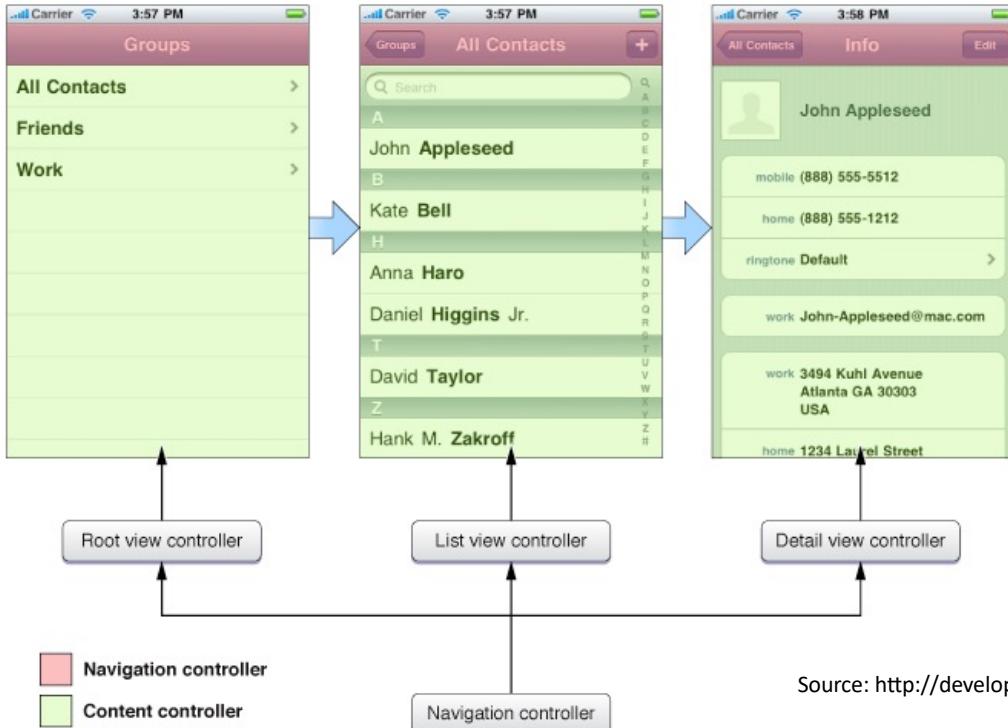
- Bild aus der Apple-Doku



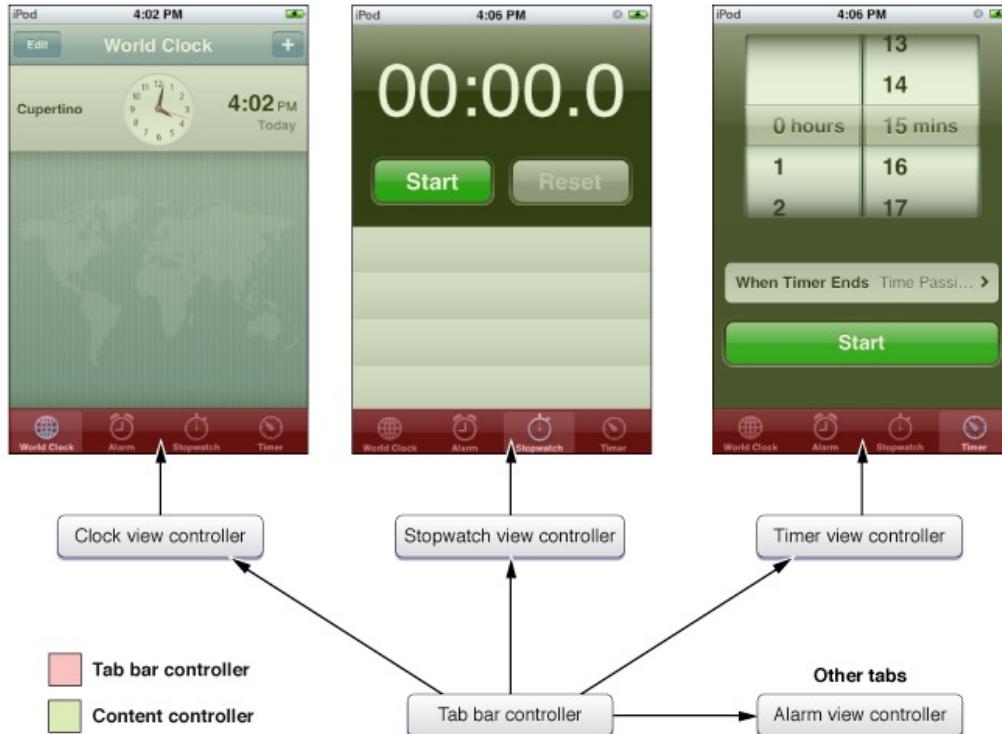
# UITableViewController



# UINavigationController

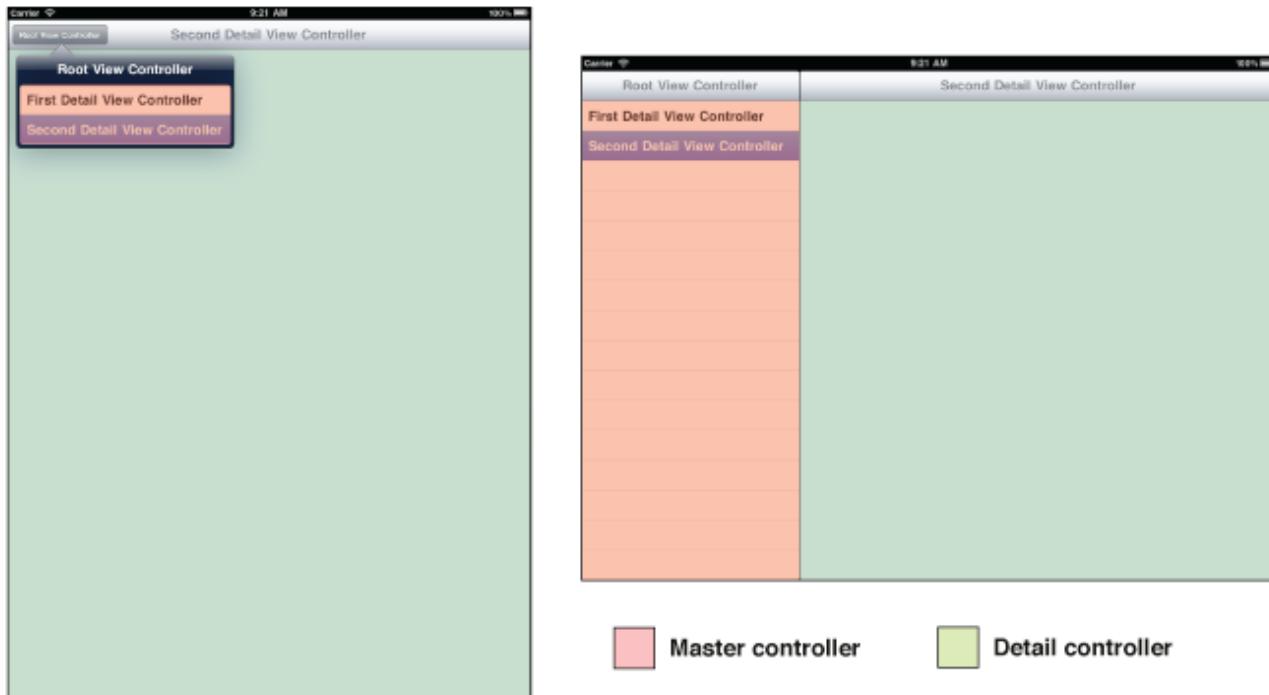


# UITabBarController



Source: <http://developer.apple.com/>

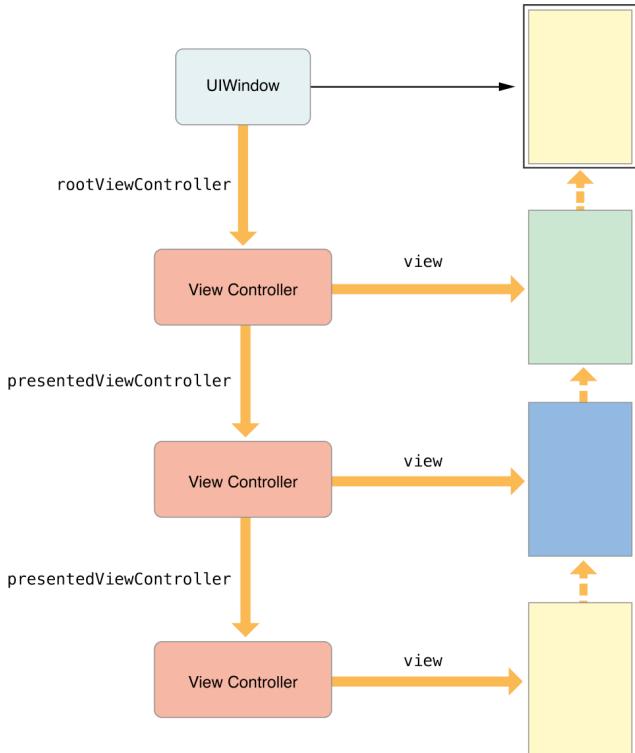
# UISplitViewController (iPad)



Source: <http://developer.apple.com/>

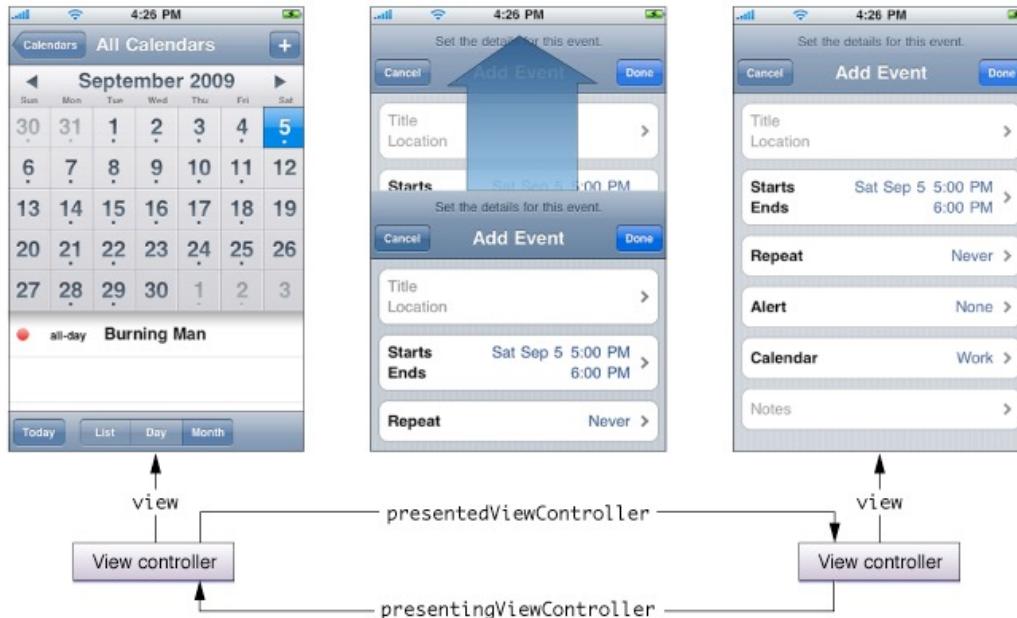
# Modal View Controller

- Teil der Klasse `UIViewController`
  - d.h. `UIViewController` kann auch eine Art "Container View Controller" sein und andere Views (resp. `ViewController`) modal präsentieren



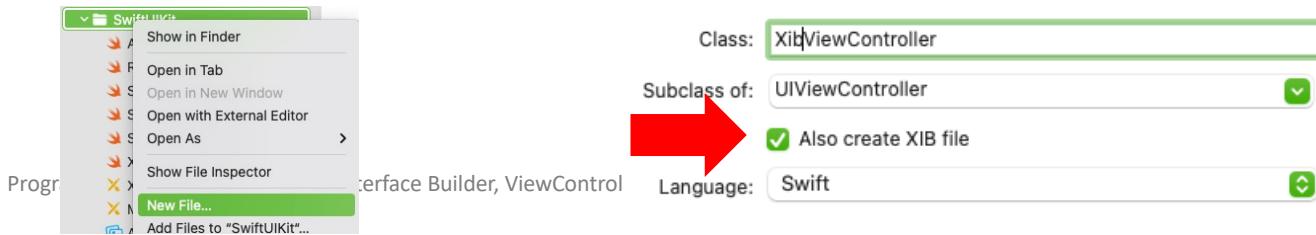
# Modal View Controller

- Altes Bild aus der Apple-Doku:



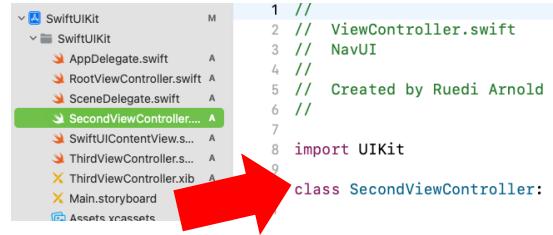
# Demo XibViewController

- Knopf "Show XibView" -> IBAction-Methode
  - UIViewController.present(...)  
`self.present(XibViewController(), animated: true, completion: nil)`
- Klasse XibViewController generieren lassen
  - Unterklasse von UIViewController
  - Häcken "Also create XIB file" unter "New File..." –  
"iOS : Cocoa Touch Class"



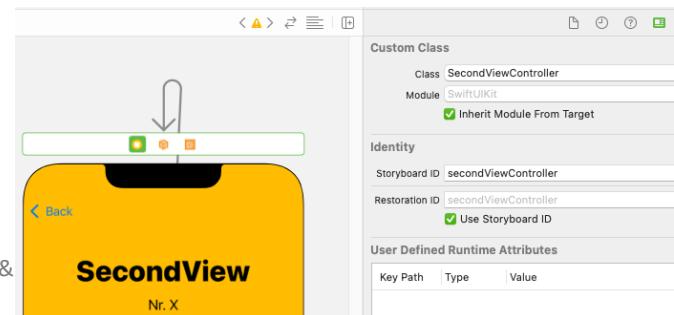
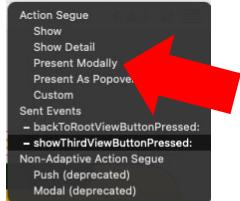
# Demo modaler SecondVC

- Storyboard mit SecondViewController
  - Subklasse von UIViewController
- Recap IBOutlet
- Knopf mit IBAction
  - Segue: Present Modally



```
1 // ViewController.swift
2 // NavUI
3 // Created by Ruedi Arnold
4 //
5 import UIKit
6
7
8 class SecondViewController:
```

A screenshot of the Xcode interface showing the project structure and the code for `SecondViewController.swift`. A red arrow points from the storyboard reference in the code back to the storyboard scene in the Interface Builder preview.





# Storyboard

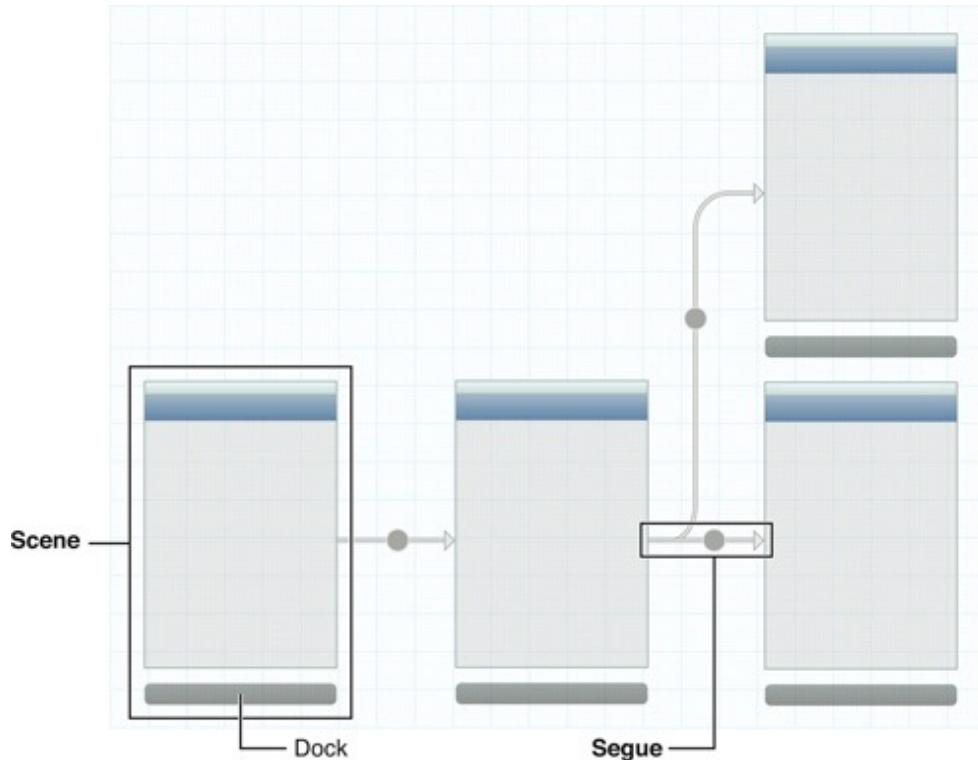
& <http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

# Storyboard: Text...

A storyboard is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens. A storyboard is composed of a sequence of scenes, each of which represents a view controller and its views; scenes are connected by segue objects, which represent a transition between two view controllers.

<https://developer.apple.com/library/ios/#documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html>

# ...und Bild dazu ("Storyboard II")



<https://developer.apple.com/library/ios/documentation/general/conceptual/Devpedia-CocoaApp/StoryboardSegue.html>

# "Scene = 1 Screen"

- iPhone: "1 Szene = 1 voller Bildschirm"
  - "each scene corresponds to a full screen's worth of content"
  - iPad: multiple scenes can appear on screen at once - for example, using popover view controllers
- Hinter jeder Szene steht ein ViewController

# Segue = der Übergang

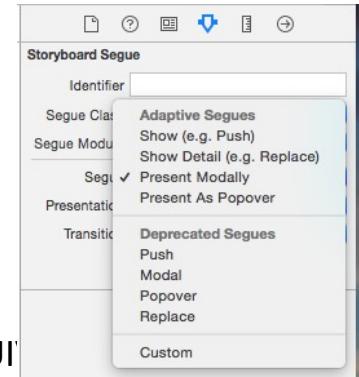
- Ausgesprochen wie "Segway"
  - Hier keine Scooter für Menschen... sondern Übergänge für Szenen! ;-)



<http://www.segway-point.ch/>

# UIStoryboardSegue

- Segue = Übergang zwischen zwei Szenen
  - A UIStoryboardSegue object is responsible for performing the visual transition between two view controllers
  - Verschiedene "Styles", siehe nächste Folie...
- 2 Methoden
  - `(id)initWithIdentifier:(NSString *)identifier source:(UIViewController *)source destination:(UIViewController *)destination`
  - `(void)perform`



# Styles von Segues

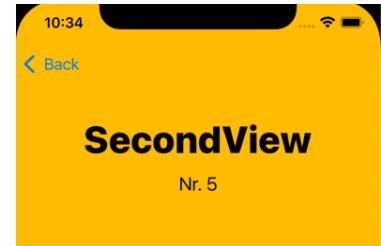
- Show (Push): Modal oder Push
- Show Detail (Replace): innerhalb von SplitViewController (sonst modal)
- Present Modally: Modaler "Overlay"
- Present as Popover: selbstsprechend...

# UIViewController & Segues

- UIViewController: 2 Methoden im Zusammenhang mit Segues:
    - `(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender`
      - Vor Segue-Ausführung, z.B. um Daten zu übergeben ☺
    - `(void)performSegueWithIdentifier:(NSString *)identifier sender:(id)sender`
      - Übergang programmatisch auslösen
      - Verknüpfung typischerweise im Interface Builder (Widget -> Action)
- Wird typischerweise vom System / Storyboard aufgerufen

# Demo Storyboard & Segue

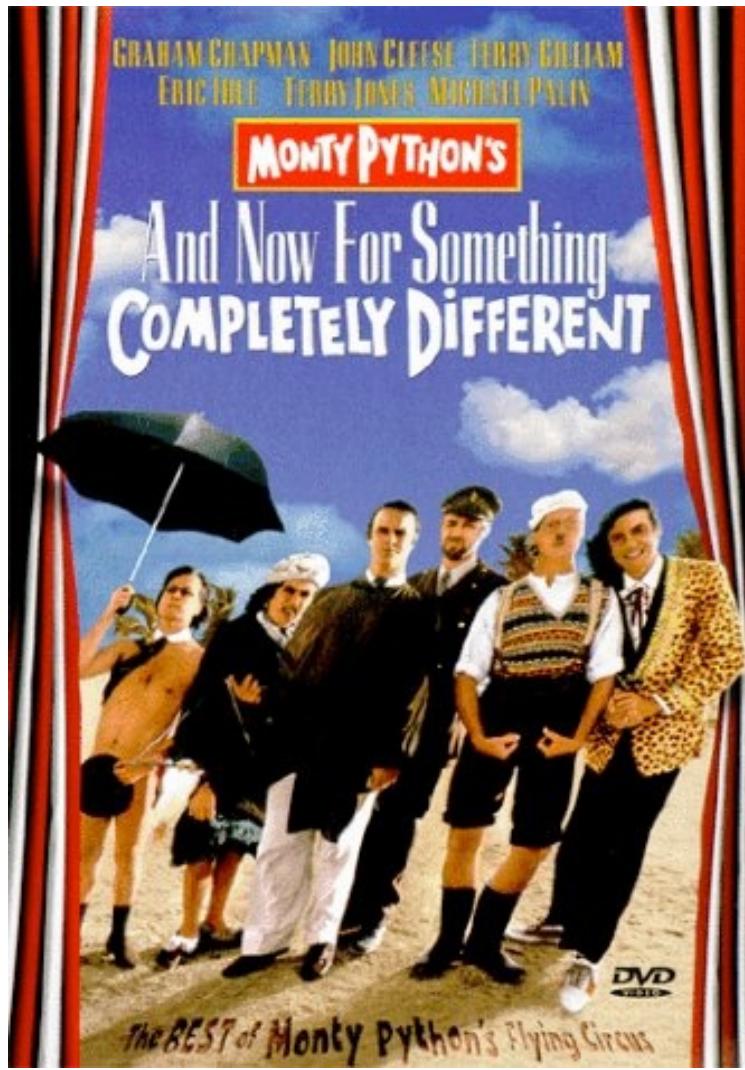
- SecondViewController
  - Subklasse von UIViewController
- Datenübergabe in von MainView zu SecondView in UIViewController-Methode `performSegueWithIdentifier:sender:`
  - Property auf VC-Klasse



```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
 if segue.destination is SecondViewController {
 let secondVC = segue.destination as! SecondViewController
 secondVC.number = self.number + 1
 }
}
```

# Swift: Typcheck (is) & Casting (as)

- **is: Test auf Typ** (Klasse, Protokoll, Enum, Struct)
  - Bsp.: if anyObject is String { ... }
- **as: Cast-Operator**
  - Bsp.: var x = anyObject as String
  - Zwei Varianten: as! und as?
    - **as!**: erzwingt Cast, (falls nicht möglich: Laufzeit-Fehler)
    - **as?**: Liefert nil zurück, falls Cast nicht möglich



# UINavigation- Controller

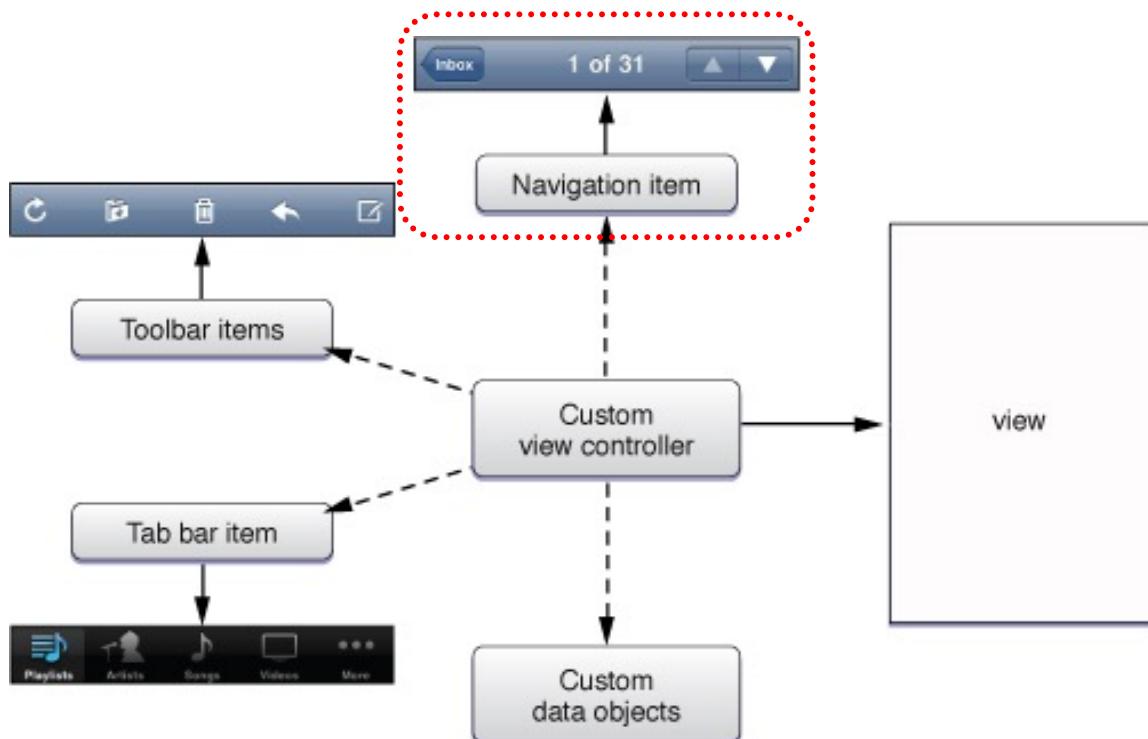
<http://en.wikipedia.org/wik/File:Somethingdifferent.jpg>

& SwiftUI

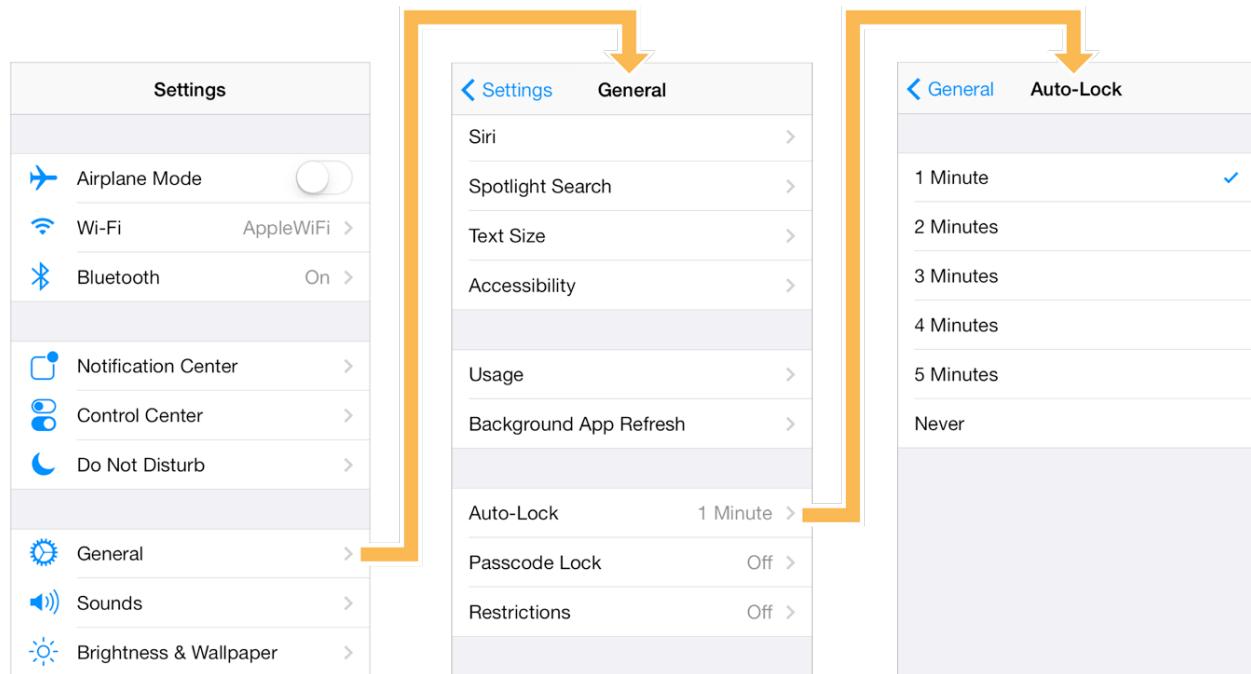
R. Arnold – HS22 V1.0

53

# Anatomie eines View Controllers



# Bsp. Navigation



# UINavigationController

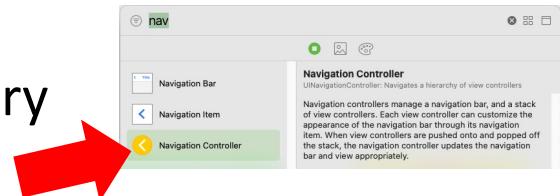
- Container View Controller
- Verwaltet Hierarchie von Views
  - Hat einen Root-View-Controller
- Viele "praktische" Defaults
  - Navigation-Bar
  - Back-Button
- Wichtigste Methoden
  - pushViewController:animated:
  - popViewControllerAnimated:
  - show:

# Ein Stapel von Views...

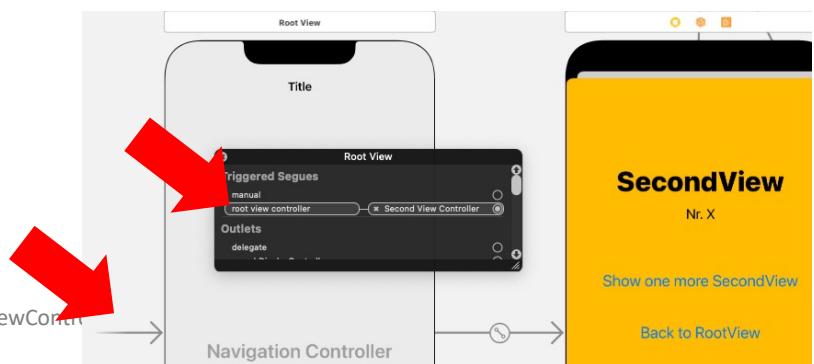
- Grundidee
  - **Push:** Neue View (d.h. entsprechender ViewController) wird auf einen Navigation Controller gedrückt
    - pushViewController:animated:
  - **Pop:** Alte View (d.h. entsprechender ViewController) wird wieder von einem Navigation Controller entfernt
    - 3 Varianten:
      - popViewControllerAnimated:
      - popToRootViewControllerAnimated:
      - popToViewController:animated:

# Demo: Einbau NavigationController

1. Drag'n'Drop auf Object Library

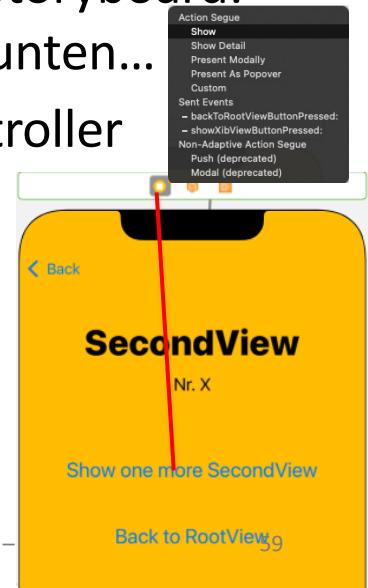


2. Auf NavCon RootViewController setzen  
(mit ctrl-Maus)
3. StartPfeil auf RootView setzen



# Demo NavigationViewController

- SecondViewController mit Knopf "Show one more SecondView"
  - Verbindung Knopf – Segue Action im Storyboard:  
siehe rote Linie im Screenshot rechts unten...
  - Show (= Push) neuer SecondViewController
- Knopf "Back to RootView"
  - Eigene IBAction-Methode in  
SecondViewController-Klasse: Aufruf  
VON `navigationController.popToRootViewController(...)`



# Verschiedene Konzepte für verschiedene Anwendungen 😊



Navigation



Modal



Tabs



# Interoperabilität UIKit - SwiftUI

& SwiftUI  
<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

# UIKit & SwiftUI

- Aktuell also zwei UI-Technologien für iOS
- Q: Lassen sich UIKit und SwiftUI kombinieren?
- A: Ja, in beide Richtungen!
  - Wir schauen je einen einfachen Fall an (ohne komplizierte Navigation, Datenübergabe usw.)
  - Beide Richtungen potentiell sinnvoll
- Frage in die Runde: Typische Szenarien?

# SwiftUI in einem UIViewController

- Klasse UIHostingController
  - Unterklasse von UIViewController
  - init(rootView: Content): "Creates a hosting controller object that wraps the specified SwiftUI view."
- Wichtig: SwiftUI-Framework importieren
  - import SwiftUI

**Summary**  
A UIKit view controller that manages a SwiftUI view hierarchy.

**Declaration**

```
class UIHostingController<Content> where Content : View
```

**Discussion**  
Create a UIHostingController object when you want to integrate SwiftUI views into a UIKit view hierarchy. At creation time, specify the SwiftUI view you want to use as the root view for this view controller; you can change that view later using the `rootView` property. Use the hosting controller like you would any other view controller, by presenting it or embedding it as a child view controller in your interface.

# Demo: UIHostingController

- SwiftUI-View erzeugen
  - z.B. copy/paste ContentView von SwiftUI-Projekt-Template
- Eigene @IBAction
  - HoistingController erzeugen und (modal) anzeigen

```
iPhone 12 Pro
12:01
Hello, world!
```

```
struct ContentView: View {
 var body: some View {
 VStack {
 Text("Hello, world!")
 .padding()
 }
 }
}
```

```
let swiftUIC = UIHostingController(rootView: ContentView())
self.present(swiftUIC, animated: true)
```

- Show (push) ginge natürlich auch, siehe Übung ☺

<https://developer.apple.com/documentation/swiftui/uihostingcontroller>

# UIViewController in SwiftUI

## Summary

A view that represents a UIKit view controller.

## Declaration

```
protocol UIViewControllerRepresentable :
Never
```

- UIViewController aus SwiftUI anzeigen:  
UIViewControllerRepresentable = Protokoll  
mit zwei relevante Funktionen:

```
func makeUIView(context: Self.Context) -> Self.UIViewType
```

Creates the view object and configures its initial state.

Required.

```
func updateUIView(_ uiView: Self.UIViewType, context: Self.Context)
```

Updates the state of the specified view with new information from SwiftUI.

Required.

<https://developer.apple.com/documentation/swiftui/uiviewrepresentable>

# Demo: UIViewControllerRepresentable

- Eigene struct erzeugen, welche gewünschte Subklasse von UIViewController zurück gibt:

```
struct UIKitVC: UIViewControllerRepresentable {
 func updateUIViewController(_ uiViewController: XibViewController,
 // not used in this simple example.
)
 func makeUIViewController(context: Context) -> XibViewController {
 return XibViewController()
 }
}
```

- Und dann z.B. `Button("Show XibView") {  
 modal = true  
}.sheet(isPresented: $modal,  
 onDismiss: { modal = false },  
 content: { UIKitVC() })`  
modal anzeigen:



# Kurs-Ausblick & Übung 5

<http://en.wikipedia.org/wik/File:Somethingdifferent.jpg>

& SwiftUI

R. Arnold – HS22 V1.0

67

# Kurs-Ausblick

- SW06: Fragmentierung, mobile Usability, Widgets
- SW07: Persistenz, Testen, Property Wrappers
- SW08: Memory Management, Frameworks
- SW10-14: Teamprojekt (+ ggf. Gastvorträge)
  - Zu zweit je eine eigene App umsetzen!
  - Idee - Papier-Prototyp - Umsetzung - Präsentation - Demo ☺

# Zwischenfazit nach 5 Wochen...

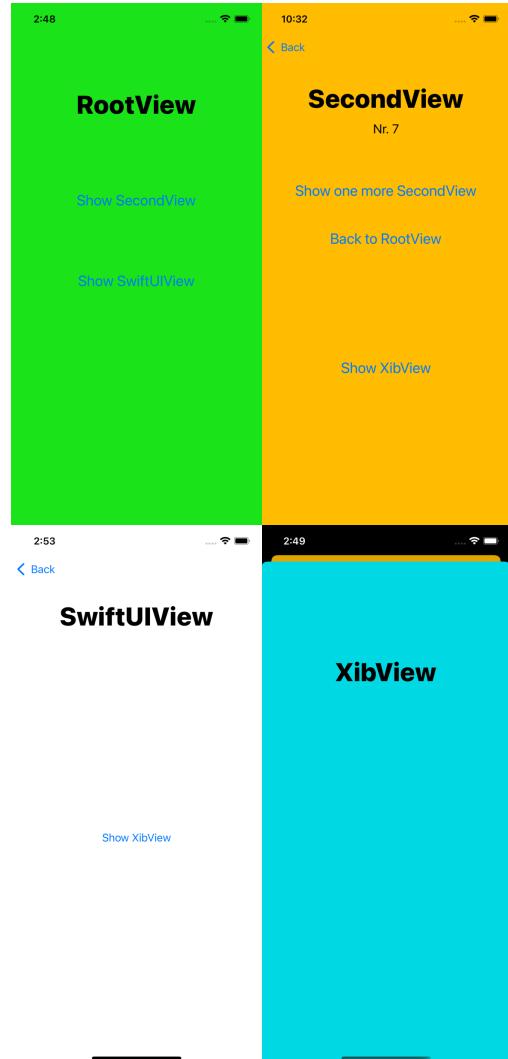
- Fragen / Anmerkungen?
- Inhalt soweit ok?
- Modul-Modus inkl. Präsenzunterricht passt?
- Übungen?
- Sonstwas?..

Thx4feedback... spontan jetzt oder später!

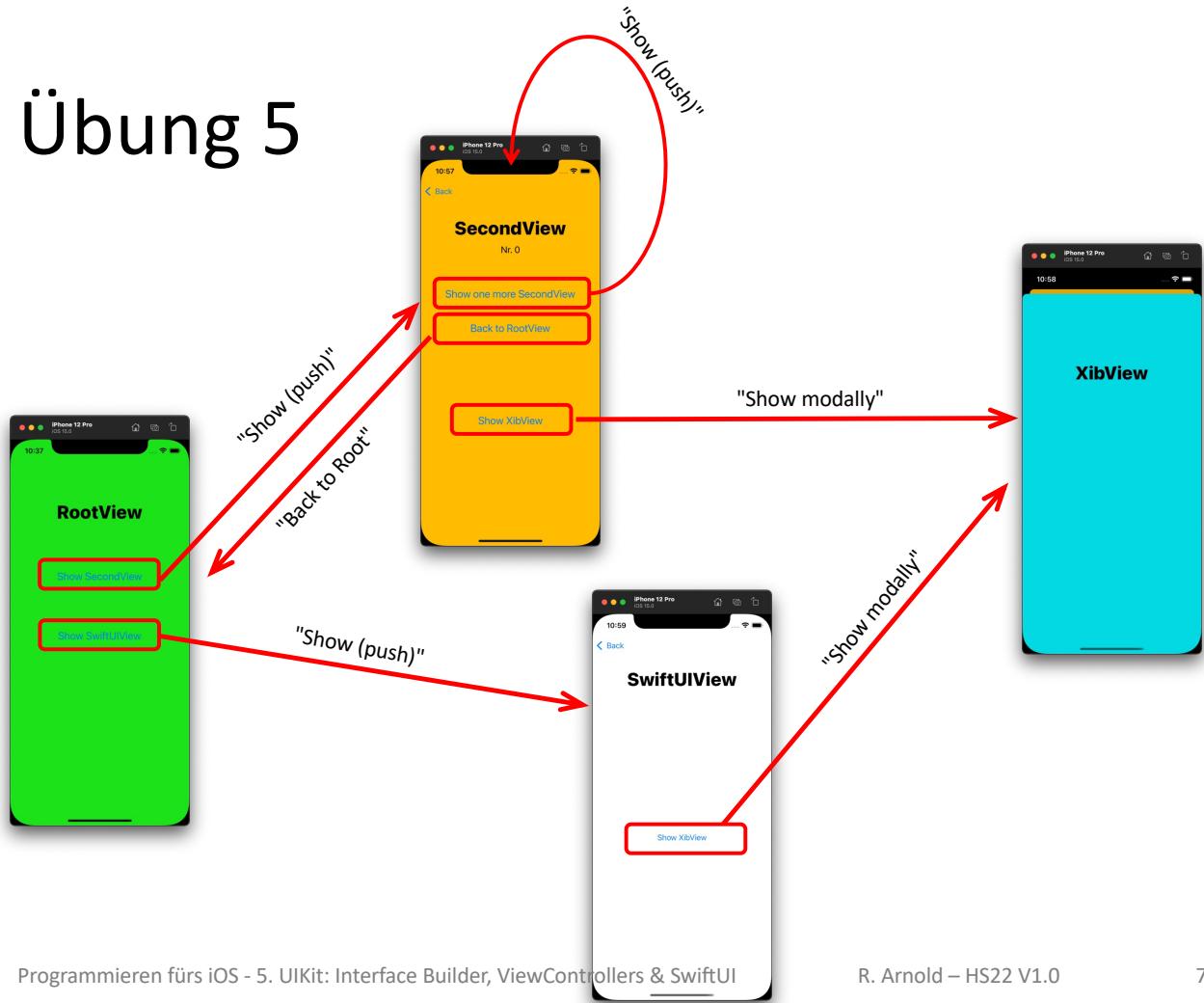
...dieses Modul findet  
zum 11. Mal statt und wir  
geben uns dabei Mühe und  
möchten es gerne weiter verbessern! 😊

# Übung 5

- Interface Builder & Widgets
  - IBOutlets & IBAction
- Mehrere ViewControllers
  - mit .xib, resp. im .storyboard
- Übergänge
  - Modal & Show (Push)
  - Programmatisch & im Storyboard
- Interop SwiftUI
  - UIViewController@SwiftUI
  - SwiftUI@ViewController



# Übung 5



# Ü5: Setup Storyboard



- Dazu je 1 Scene aus xib- und SwiftUI-Datei 😊

# Fragmentierung, Mobile Usability, HIG, Lokalisierung, Widgets

Programmieren für iOS



# Inhalt

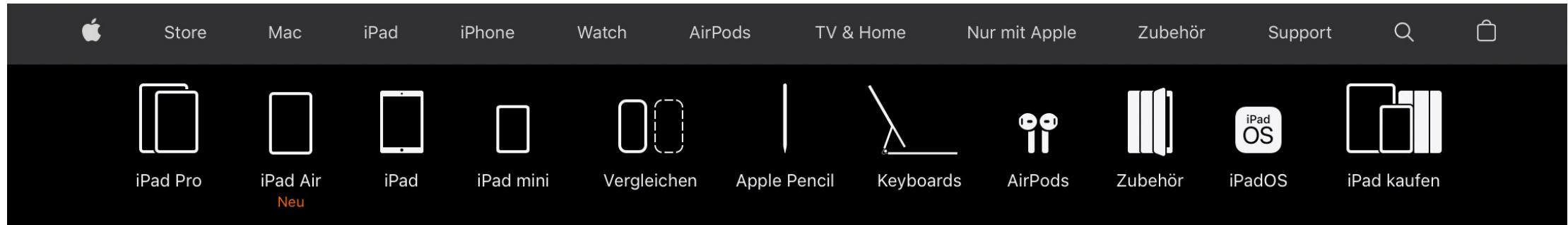
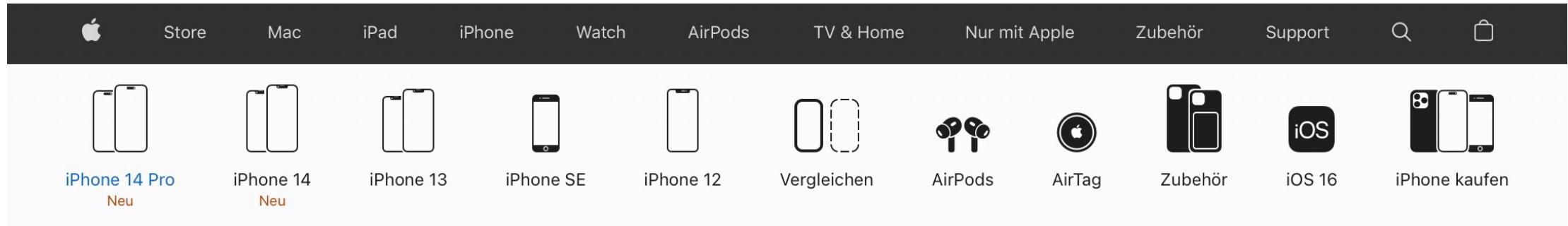
- Fragmentierung: iOS-Geräte und Eigenschaften
- App-Design & Human Interface Guidelines (HIG)
- Mobile Usability und User Experience (UX)
- Internationalisierung & Lokalisierung
- Widgets
- App-Veröffentlichung

# Die iOS-Gerätefamilie



# Die iOS-Gerätefamilie

... wächst jedes Jahr:



# Geräte-Eigenschaften

- Touch-Bildschirm (verschiedene Größen)
- Kamera(s) mit verschiedenen Spezifikationen
  - Front & Back
- Audio
- (Satelliten-)Telefonie
- 4G/5G, GPS, WiFi, Bluetooth, ...
- Viele Sensoren: Kompass, Beschleunigung, TouchID/FaceID, Helligkeit, Gyroskop, NFC, ...
- Weitere spezifische Chips: GPU, U1, Lidar, ...

# Bildschirm-Auflösungen

- <https://iosref.com/res>

## Resolution by iOS device

[iPhone](#) [iPad](#) [Apple Watch](#) [iPod touch](#) [Further reading](#)

All tables are ordered by release date, then logical resolution height, with some exceptions.

### iPhone

| Device                                             | Diagonal size | Logical resolution | Scale factor          | Actual resolution | Aspect ratio | PPI |
|----------------------------------------------------|---------------|--------------------|-----------------------|-------------------|--------------|-----|
| iPhone 14 Pro                                      | 6.1"          | 393 × 852          | @3x                   | 1179 × 2556       | 9 : 19.5     | 460 |
| iPhone 14 Pro Max                                  | 6.7"          | 430 × 932          |                       | 1290 × 2796       |              |     |
| iPhone 14 Plus<br>and 13 Pro Max<br>and 12 Pro Max |               | 428 × 926          |                       | 1284 × 2778       |              |     |
| iPhone 14<br>and 13 / 13 Pro<br>and 12 / 12 Pro    | 6.1"          | 390 × 844          |                       | 1170 × 2532       |              |     |
| iPhone 13 mini<br>and 12 mini                      | 5.4"          | 375 × 812          | @2.88–3x <sup>1</sup> | 1080 × 2340       | 476          | 458 |
| iPhone 11 Pro Max                                  | 6.5"          | 414 × 896          | @3x                   | 1242 × 2688       |              |     |

# Bildschirm-Auflösungen

- 3 “Versionen” von Pixels:
  1. “Points” → Im Code verwendete Einheit. Abstraktion von physikalischen Pixels.
  2. “Rendered Pixels” → Bilder werden hochskaliert mit bestimmtem Faktor, meistens 1x / 2x / 3x.
  3. “Physical Pixels” → Rendered Pixels werden auf physische Pixels im Display gemapped. Meistens 1:1, kann aber auch != 1 sein (siehe Tabelle).
- Dazu: PPI (Pixels-per-Inch) → Sagt aus, wieviele Pixels in einem Inch Platz haben und somit, wie gross das Display am Schluss in der echten Welt ist

<https://www.paintcodeapp.com/news/iphone-6-screens-demystified>

# Bildschirm-Auflösungen

- Im Code: Wir haben fast immer nur mit **Points** zu tun
- Falls nötig: Scale Factors können ausgelesen und verwendet werden.

```
/// The natural scale factor associated with the screen.
```

```
let scale = UIScreen.main.scale
```

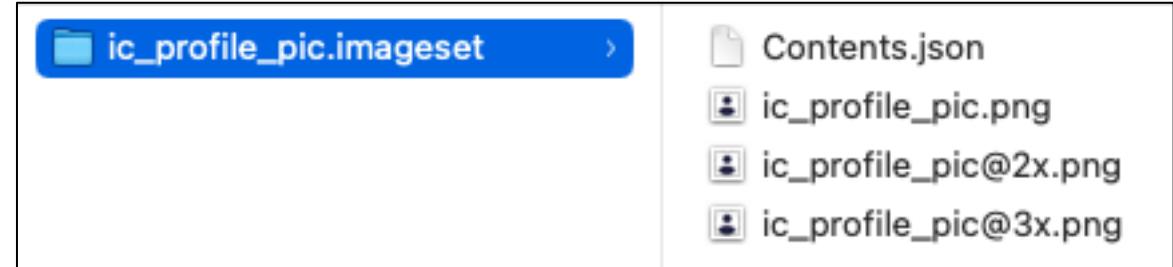
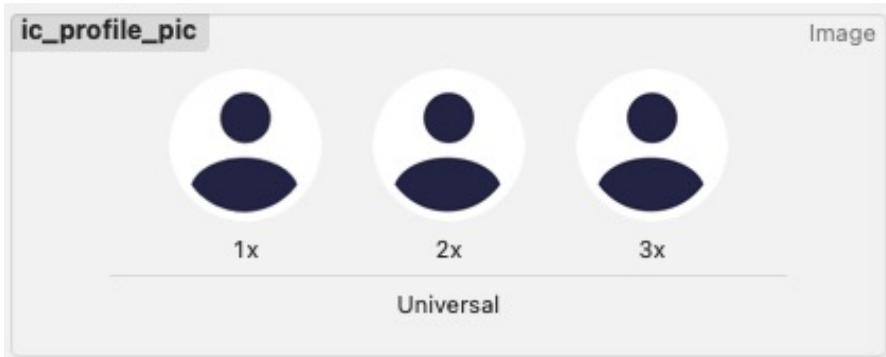
```
/// The native scale factor for the physical screen.
```

```
let nativeScale = UIScreen.main.nativeScale
```

- NativeScale kann mehrheitlich ignoriert werden, kann wichtig sein in Grafik-lastigen Anwendungen (z.B. Games)

# Exkurs: Bilder

- Bilder werden in 3 Auflösungen gespeichert
- App wählt richtige Auflösung für jeweiligen Screen-Scalefaktor
- Wenn richtig benennt: Drag and drop in Xcode
- Viele Design-Tools haben mittlerweile Export-Funktion mit richtigem Format

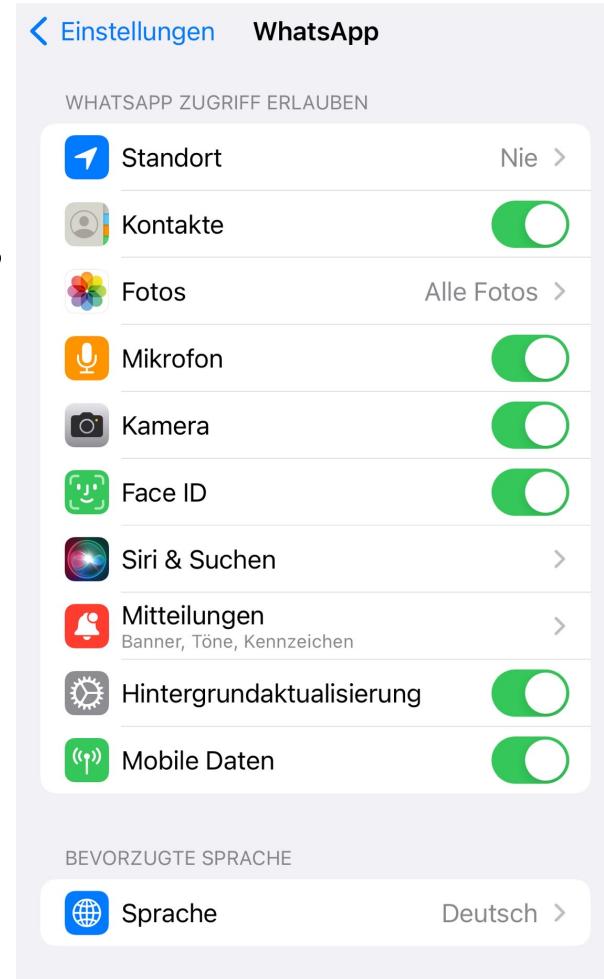


# Bildschirm-Auflösungen

- Grundsatz für UI-Programmierung: So wenig wie möglich mit hardcoded Point-/Pixel-Werten arbeiten → Relatives Layout
- Mit SwiftUI nochmals besser geworden: Man sagt, was man möchte (z.B [List](#)), System sorgt für sinnvolle Darstellung, sogar auf verschiedenen Devices (iPhone, iPad, Watch)
- Wenn fixe Größen nötig sind: Kleinstes / grösstes Gerät im Auge behalten (+ Displayzoom, Dynamic Type etc.)

# Geräte: Möglichkeiten / Einschränkungen

- Immer mehr neue Sensoren & Berechtigungen
- System “schützt” Nutzer vor Apps:
  - Sandbox → Apps haben nur sehr beschränkten Zugriff auf's Filesystem oder auf Daten anderer Apps
  - Gute Kontrolle darüber, welche Daten an welche App gegeben werden
  - Aber: Permission Handling aus Entwickler-Sicht kann mühsam werden. Viele Spezialfälle möglich (z.B. kein Background-Fetch, nur “ungefähre Location” freigegeben)



# Interaktions-Limitierungen

- Was es standardmässig auf iOS nicht gibt:
    - Maus
    - USB-Anschluss (für externe Keyboards oder andere Accessories)
  - Dafür gibt's:
    - Software-Keyboards (flexibler, z.B. optimiert für Inputfeld)
    - Multitouch → Gesten
    - Zugriff auf Sensoren
    - iPad: Apple Pencil
- Interessante, neue Mensch-Computer-Interaktionen!

# App-Design & Human Interface Guidelines

# App-Design



# App-Design

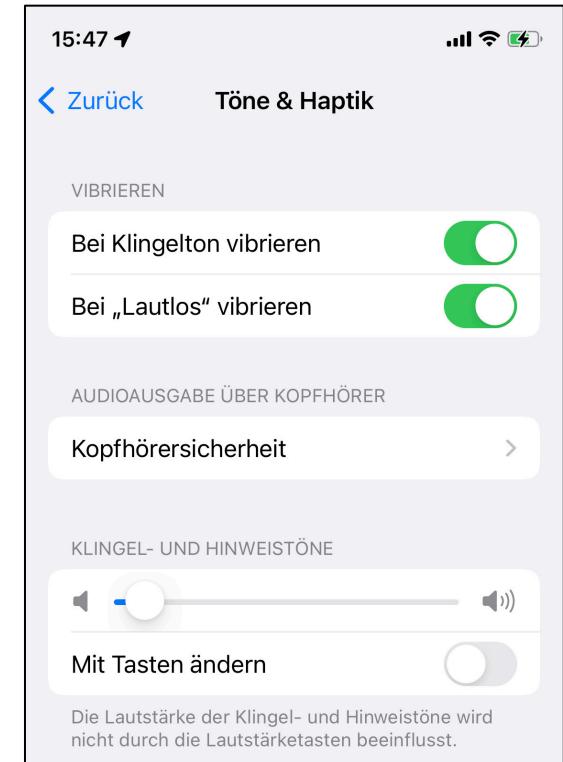
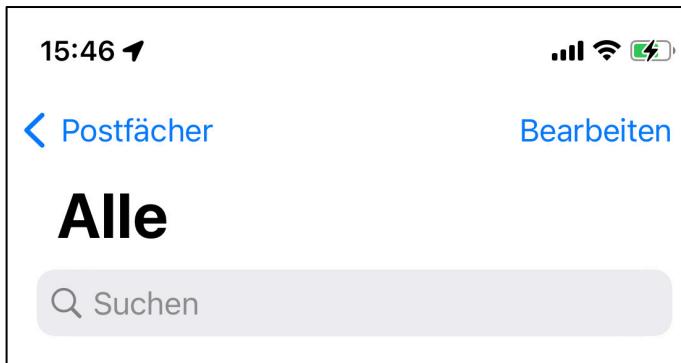
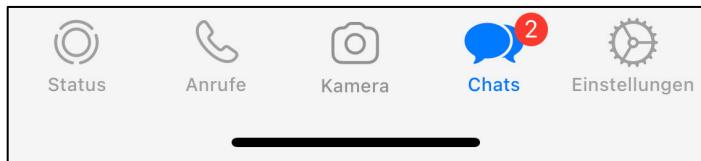
- Warum sehen viele iOS-Apps einheitlich und “schön” aus?

→ Apple macht Regeln!



# Standard-UI

- System-Frameworks (UIKit & SwiftUI) enthalten viele Standard-UI-Elemente:
  - Navigation Bars
  - Tab Bars
  - Buttons, Toggles, Sliders, ...



# Human Interface Guidelines

- Guide zur Erstellung von Apps für Apple-Plattformen
- Instruktionen & Hinweise, wie die Standard-UI-Elemente eingesetzt werden sollten
- Ausführliche Sammlung von Best Practices (“The Apple Way”)

**Im Grossen und Ganzen sinnvoll – aber nicht als einzige Wahrheit zu sehen!**

- Grosser Vorteil: Apps fühlen sich einheitlich und vertraut an, User müssen weniger “lernen”

# Human Interface Guidelines

Demo

<https://developer.apple.com/design/human-interface-guidelines/guidelines/overview/>

# Design, Usability & UX

# UX – Warum?



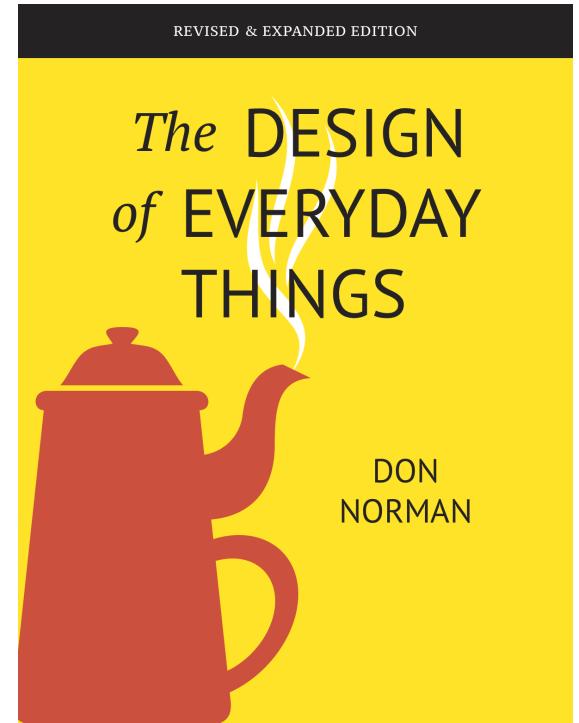
# UX – Warum?

# Definitionen

- Design: Spezifischer Zweck, Absicht, wie etwas zu gebrauchen ist / auszusehen hat.  
Kommunikation zwischen Objekt und User.
- Usability: “Gebrauchtstauglichkeit”, der Grad, die Qualität der Benutzbarkeit von etwas
  - Wie gut kann ein **User** in einem bestimmten **Kontext** ein bestimmtes **Ziel** erreichen
  - Viele Aspekte: nützlich, zuverlässig, sicher, übersichtlich, barrierefrei, ...
- User Experience: Wie einfach und angenehm etwas zu benutzen ist
  - Ziel: Glückliche User :-)

# Design-Prinzipien

- “The Design of Everyday Things”, Donald Norman (1988)
- Was macht gutes & schlechtes Design aus?
- Vorschlag verschiedener **Design-Prinzipien**



# Design-Prinzipien

- **Affordances**: Mögliche Handlungen, die ein Objekt erlaubt.  
Bsp.: Ein Stuhl erlaubt es, darauf zu sitzen (ein Tisch aber auch!).
- **Signifiers**: Helfen, die Affordances eines Objektes zu kommunizieren, v.a. dort wo sie nicht offensichtlich sind  
Bsp.: Labels, Schilder
- **Constraints**: Limitierungen, die die Benutzung eines Objektes einschränken
- **Mappings**: Beziehungen zwischen Objekt und Resultat. Können z.B. örtlich oder zeitlich sein.
- **Feedback**: Kommunikation der Resultate einer Handlung für den User.

# Affordance – Beispiele



# Affordance – Beispiele

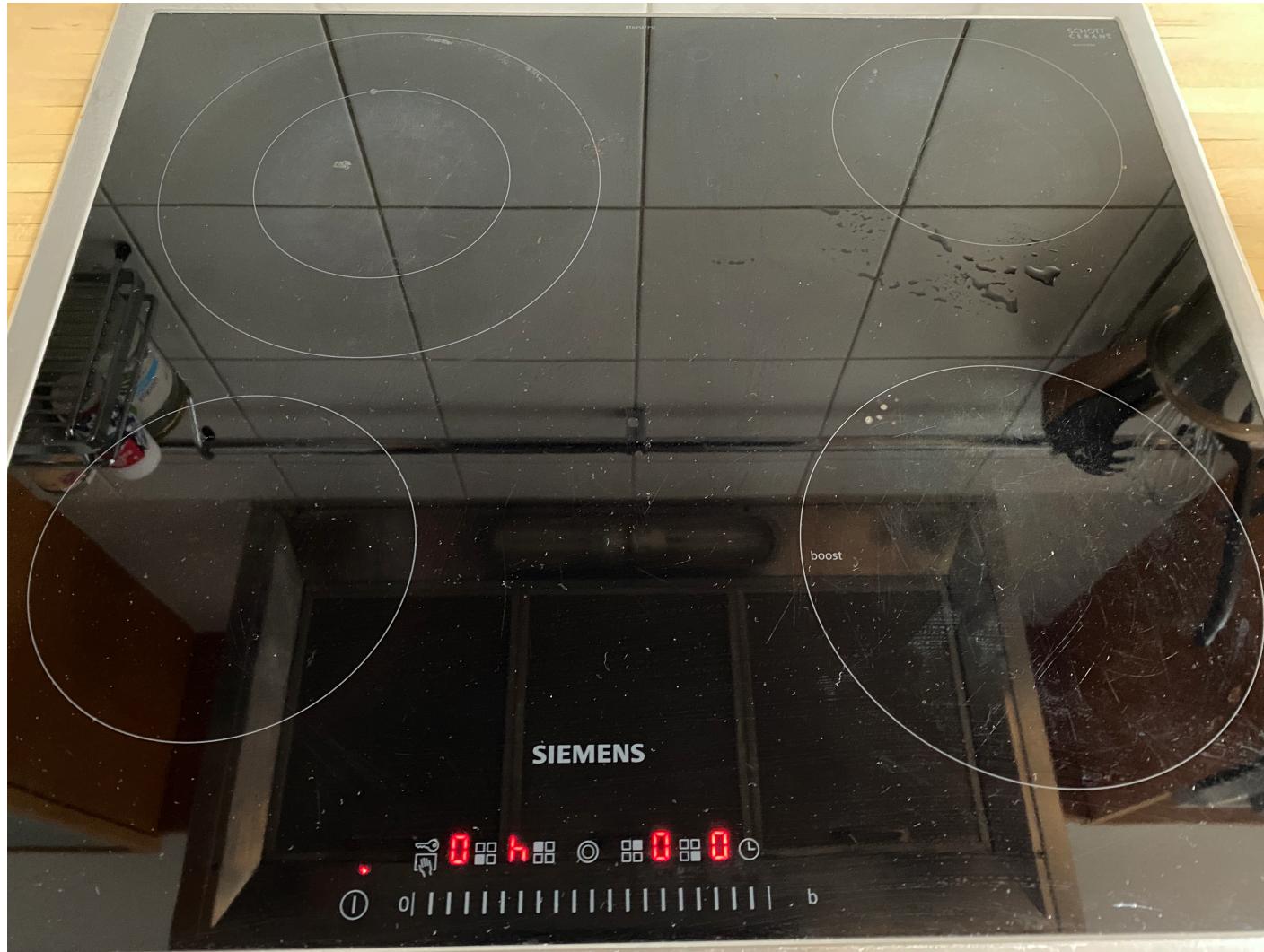
## ***Norman door (n.):***

- 1. A door where the design tells you to do the opposite of what you're actually supposed to do.**
- 2. A door that gives the wrong signal and needs a sign to correct it.**

# Mappings – Beispiele







# Mappings – Beispiele

- Besseres Mapping, weil Beziehung zur realen Welt sofort erkennbar ist
- Ausserdem auch Affordance sehr einfach erkennbar: Label und Button in einem → man weiss intuitiv, wo klicken



# UX-Design für Mobile Apps

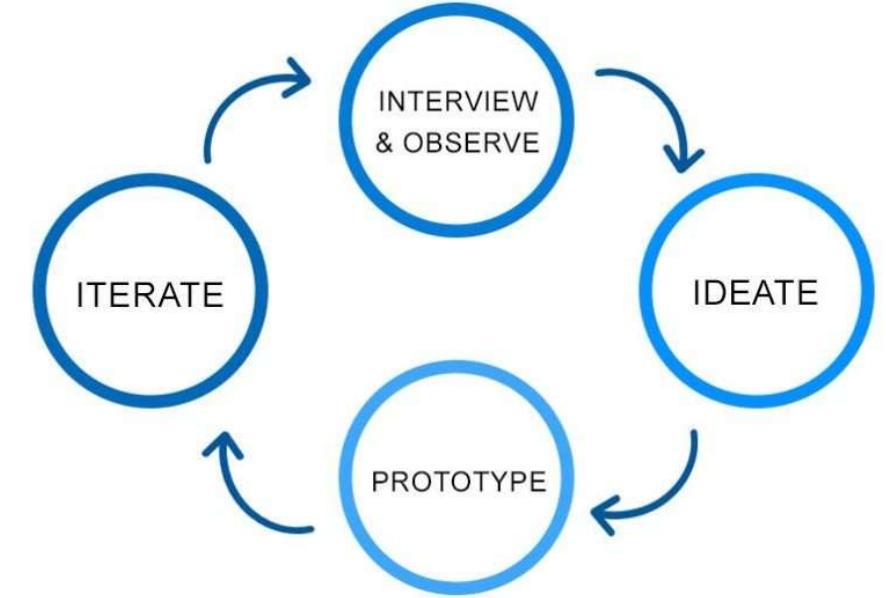
- Informations-Architektur: Wie ist der Inhalt strukturiert?
  - Navigations-Elemente: Tabbar, Navigationbar, Sheets, ...
  - Übersichten, Listen, Details, ...
- Design
  - Einfachheit
  - Einheitlichkeit (Design-System – Farben, Icons, Schriften, Spacing, ...)
- Interaktion
  - Feedbacks (z.B. Touch-States)
  - Loading & Error States
  - Sinnvolle Transitions / Animationen

# UX – Erfahrungen

- UX ist entscheidend für eine gute App
- UX ist auf allen Ebenen wichtig:
  - Konzeption
  - Design
  - Implementierung
  - ...
- Alle beteiligten Personen sollten UX-Fokus haben
- UX is hard!!

# Prototyping

- Ziel: Iterativ zum Ziel kommen, Ideen & Designs früh validieren
- Verschiedene Möglichkeiten:
  - Paper Prototyping
  - Software-Lösungen (Wireframes)
- Mit SwiftUI & Previews kann man ebenfalls sehr schnell iterieren!



# UX/Usability-Testing

- Personen von “ausserhalb” die App testen lassen → selber wird man oft blind für UX, wenn man zu tief im Projekt ist
- Keine vollständige App nötig, kann z.B. mit Papier-Prototypen oder Click-Through-Prototype gemacht werden
- Deckt UX-Probleme oft sehr schnell auf
- Ganz wichtig: Qualitatives statt quantitatives Feedback! (kleine Sample-Grösse, z.B. 2-4 Personen genügt völlig)

# Lokalisierung

# Internationalisierung

- **Internationalization – i18n:**

**Internationalization** is the process of making your app able to adapt to different languages, regions, and cultures. Because a single language can be used in multiple parts of the world, your app should adapt to the regional and cultural conventions of where a person resides. An internationalized app appears as if it is a native app in all the languages and regions it supports.

- **Localization – l10n**

**Localization** is the process of translating your app into multiple languages. But before you can localize your app, you internationalize it.

- **Achtung: Vor allem, aber nicht nur Sprache!**

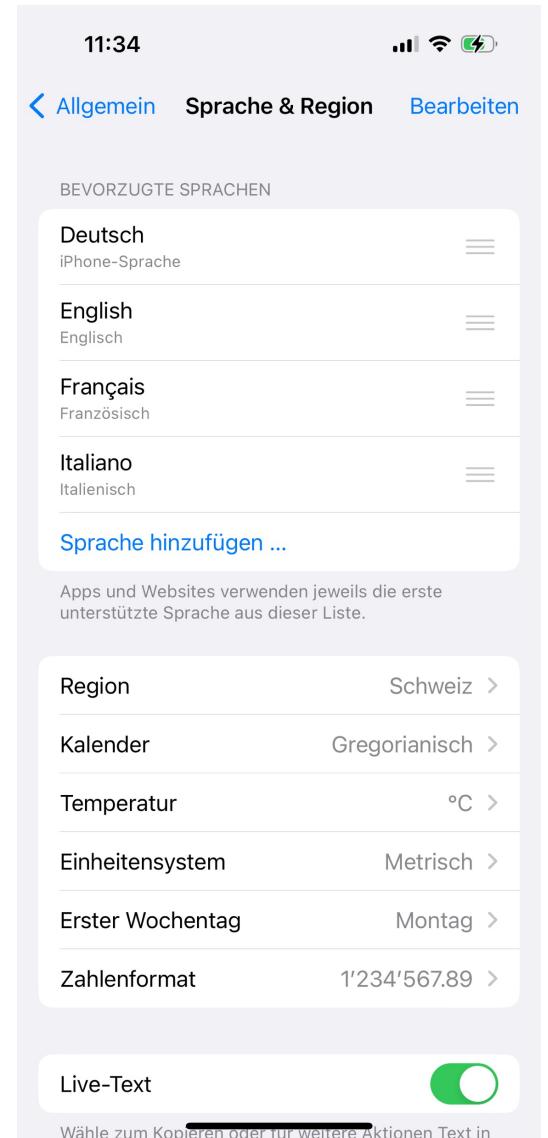
- Layout- und Text-Ausrichtung (right-to-left)
- Datums- / Zeitformate
- Zahlen
- Icons / Farben

<https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPInternational/Introduction/Introduction.html>

# Lokalisierung in iOS

- iOS gibt die Sprache vor
  - Einstellungen → Allgemein → Sprache & Region
- Apps verwenden die erste unterstützte Sprache aus dieser Liste
  - User kann pro App eine andere Sprache einstellen

Wie reagieren wir in der App auf diese Einstellungen?



# Lokalisierung in iOS

- **Locale**

Information about linguistic, cultural, and technological conventions for use in formatting data for presentation.

- Informationen über die aktuelle Konfiguration (Region, Sprache etc.)
  - Nützlich z.B. für Formatierung (Datum, Währung, Zahlen etc.)

```
func getButtonTitle() -> String {
 let languageCode = Locale.preferredLanguages[0]
 switch languageCode {
 case "en": return "Register"
 case "fr": return "Enregistrer"
 default: return "Registrieren"
 }
}
```

??

# Exkurs: ISO-Abkürzungen

- ISO 639-1: 2-Buchstaben-Kürzel für jede Sprache
  - Deutsch = de
  - Englisch = en
  - Französisch = fr
  - etc...
- ISO 3166-1: 2-Buchstaben-Kürzel für jede Region
  - Schweiz: CH
  - USA: US
  - Australien: AU
- Kann kombiniert werden, z.B. **en\_US** vs. **en\_AU**

# Lokalisierung in Xcode

- Xcode erlaubt es, Lokalisierung auf Datei-Ebene vorzunehmen
  - 1 Datei pro unterstützte Sprache/Region
- Hilfreich, um lokalisierte Inhalte von Implementation zu trennen  
→ Bessere Wartbarkeit (z.B. Arbeit mit Übersetzern)
- Funktioniert mit verschiedenen Dateitypen
  - Storyboards/Interface Builder Files
  - Texte
  - Bilder
  - Sounds
  - ...
- Im folgenden: Fokus auf Texte

# Lokalisierung in Xcode

PROJECT  
HSLUDemo

TARGETS  
HSLUDemo

Deployment Target  
iOS Deployment Target 16.1

Configurations

| Name      | Based on Configuration File |
|-----------|-----------------------------|
| > Debug   | No Configurations Set       |
| > Release | No Configurations Set       |

Use Release for command-line builds

Parallelize build for command-line builds (does not apply when using schemes)

Localizations

| Localization                       | Resources         |
|------------------------------------|-------------------|
| Base                               | 0 Files Localized |
| English — Development Localization | 0 Files Localized |

Use Base Internationalization

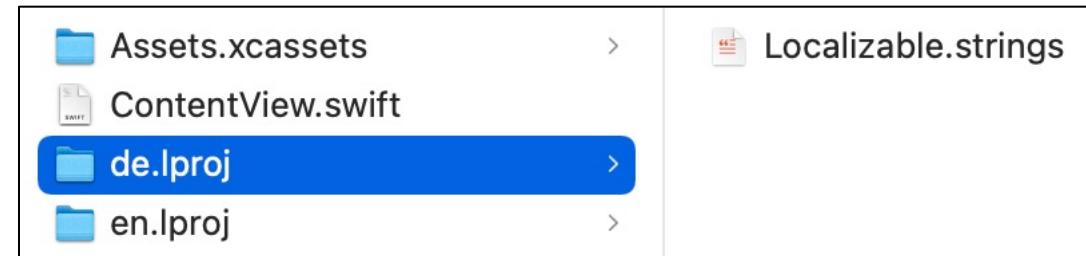
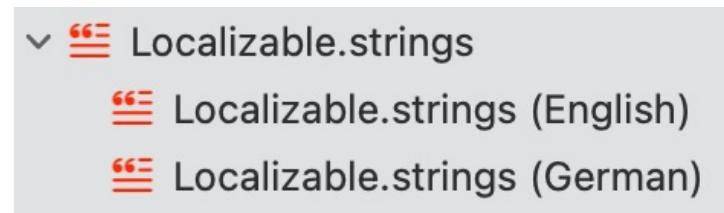
# Localizable.strings

- Dateiformat, um sprachabhängige Strings im Projekt zu verwalten
- 1 Datei pro Sprache
  - Localizable.strings
    - Localizable.strings (English)
    - Localizable.strings (German)
- Format: Key-value Pairs
  - Key = String, der im Code für Lookup verwendet wird
  - Value = übersetzter Text in jeweiliger Sprache
  - Semikolon nicht vergessen!

```
"welcome_message" = "Willkommen";
```

# Localizable.strings

- Auf dem Filesystem: 1 Folder pro lokalisierte Sprache



# NSLocalizedString

- Globale Funktion, um String-Lookup zu machen in .strings-Dateien
  - Per Default wird Localizable.strings verwendet (tableName leer lassen)
- UIKit:

```
let label = UILabel()
label.text = NSLocalizedString("welcome_message", comment: "")
```

```
func NSLocalizedString(
 _ key: String,
 tableName: String? = nil,
 bundle: Bundle = Bundle.main,
 value: String = "",
 comment: String
) -> String
```

# SwiftUI

- Ebenfalls möglich mit NSLocalizedString:

```
Text(NSLocalizedString("welcome_message", comment: ""))
```

- Noch besser / einfacher:

```
Text("welcome_message")
```

- Warum funktioniert das?

# SwiftUI

- Text hat verschiedene Initializers. Wenn ein String-literal übergeben wird, interpretiert der Compiler dies als LocalizedStringKey

## Creating a text view from a string

`init(LocalizedStringKey, tableName: String?, bundle: Bundle?, comment: StaticString?)`

Creates a text view that displays localized content identified by a key.

`init(LocalizedStringResource)`

Creates a text view that displays a localized string resource.

`init<S>(S)`

Creates a text view that displays a stored string without localization.

`init(verbatim: String)`

Creates a text view that displays a string literal without localization.

# SwiftUI – Text

```
struct ContentView: View {
 let text = "welcome_message"
 let text2: LocalizedStringKey = "welcome_message"

 var body: some View {
 VStack {
 Text("welcome_message") // ?
 Text(verbatim: "welcome_message") // ?
 Text(text) // ?
 Text(text2) // ?
 }
 }
}
```

# SwiftUI – Text

```
struct ContentView: View {
 let text = "welcome_message"
 let text2: LocalizedStringKey = "welcome_message"

 var body: some View {
 VStack {
 Text("welcome_message") // ?
 Text(verbatim: "welcome_message") // ?
 Text(text) // ?
 Text(text2) // ?
 }
 }
}
```



Welcome  
welcome\_message  
welcome\_message  
Welcome

# Lokalisierung – Demo

# Widgets

# iOS Widgets

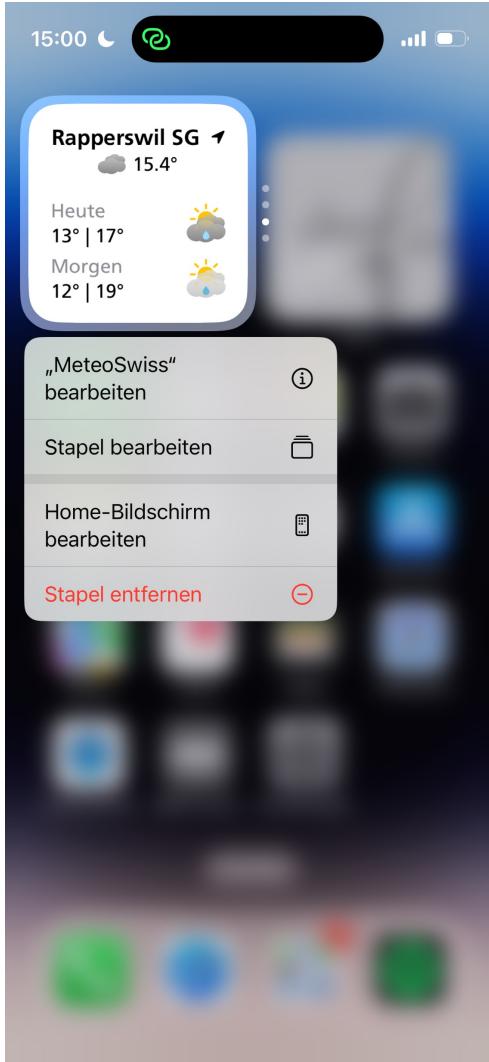
- Apple hat's (nicht) erfunden ;-)
- Seit iOS 14 (2020): Homescreen Widgets
  - Ebenfalls auf Today View (wer braucht den...?)
- Seit iOS 16 (2022): Lockscreen Widgets



# iOS Widgets – Merkmale

- Kleiner Ausschnitt einer App auf dem Homescreen
- Ist immer Teil einer App (können nicht ohne zugehörige App installiert werden)
- iOS erlaubt Widget-Stapel (Widgets wechseln manuell oder intelligent)
- Konfiguration einzelner Widgets möglich
- Verschiedene Größen: Small, Medium, Large, Extra-Large (iPad)
- Limitierte Interaktion möglich (keine Animationen etc.)
- → HIG zu Widgets: <https://developer.apple.com/design/human-interface-guidelines/components/system-experiences/widgets/>

# iOS Widgets – Konfiguration



# iOS Widgets – Dokumentation

- WidgetKit: Entsprechendes Framework  
<https://developer.apple.com/documentation/widgetkit/>
- Creating a Widget Extension  
<https://developer.apple.com/documentation/widgetkit/creating-a-widget-extension>

# iOS Widgets: Timeline

- Widgets haben sehr beschränkte Reload-Möglichkeiten
  - Abhängig davon, wie oft das Widget angeschaut/benutzt wird
  - Grundsätzlich: Keine Garantien, sondern Apple Magic...
  - Größenordnung für oft benutztes Widget: Reload alle 30 min sollte möglich sein

A widget's budget applies to a 24-hour period. WidgetKit tunes the 24-hour window to the user's daily usage pattern, which means the daily budget doesn't necessarily reset at exactly midnight. For a widget the user frequently views, a daily budget typically includes from 40 to 70 refreshes. This rate roughly translates to widget reloads every 15 to 60 minutes, but it's common for these intervals to vary due to the many factors involved.

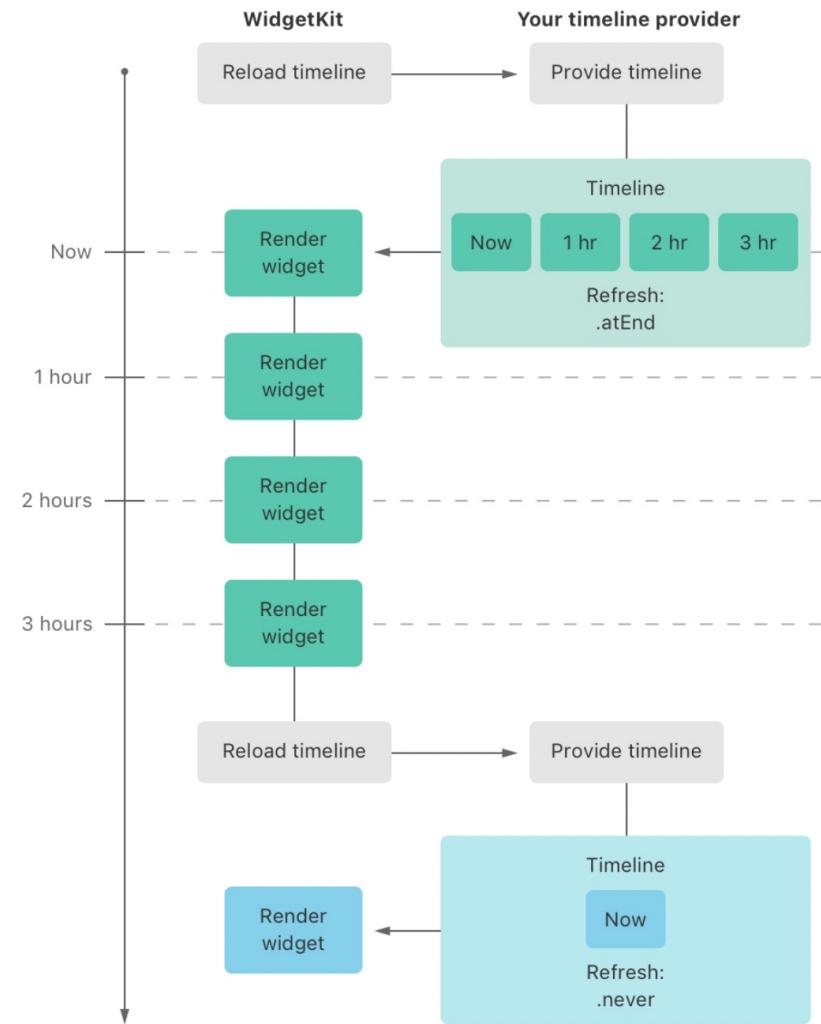
## Note

The system takes a few days to learn the user's behavior. During this learning period, your widget may receive more reloads than normal.

# iOS Widgets: Timeline

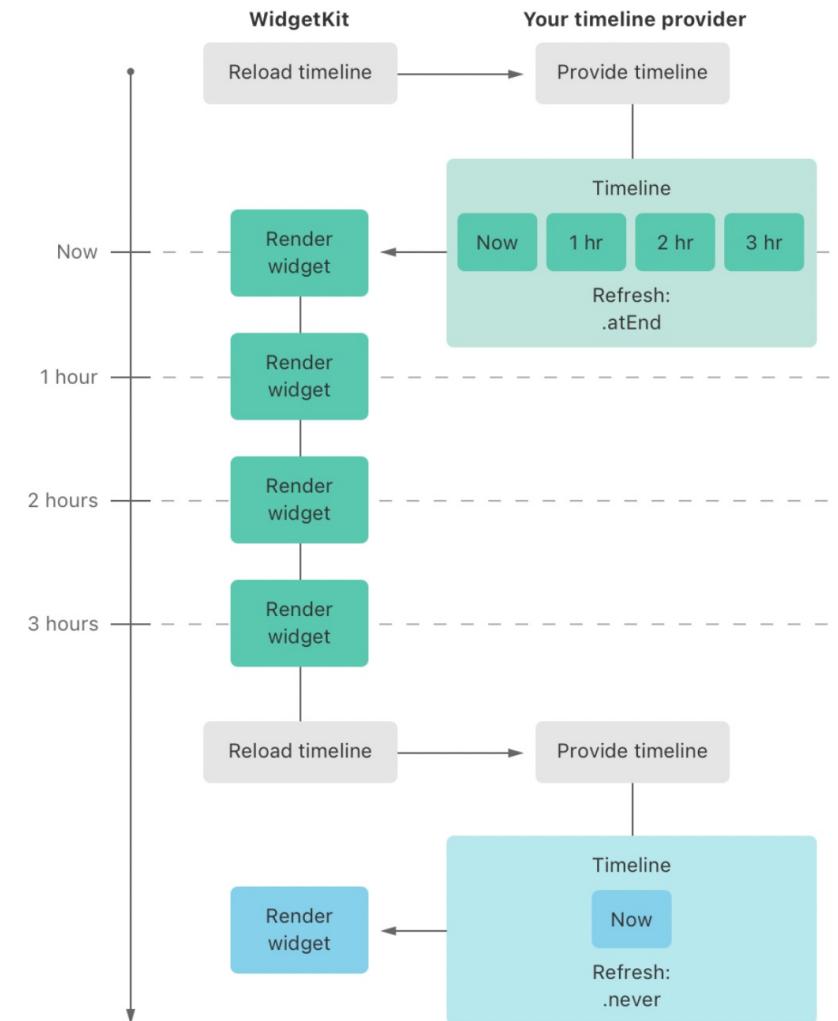
- Das System fragt nach einer Timeline
- Mittels **TimelineProvider** gibt man **TimelineEntries** an mit allen Infos, um das Widget zu rendern
  - Refresh-Policy: `.atEnd` / `.after(Date)` / `.never`
- Nach Ablauf fragt das System erneut
- Man kann Timeline-Reload auch manuell triggern (Achtung: keine Garantie!)

```
WidgetCenter.shared.reloadAllTimelines()
```



# iOS Widgets: Timeline Entries

- 2 Properties:
  - `date`: Zeitpunkt, wann das Widget gerendert werden soll
  - `relevance`: Momentane Relevanz des Inhalts für den User (optional)
- Relevanz wird benutzt, damit iOS lernt, wann Widget in Smart Stacks gezeigt werden soll
  - Bsp.: Wetter-Widget könnte Relevanz höher setzen, wenn schlechtes Wetter oder aktive Wetter-Warnungen angezeigt werden.



# Demo: Widget

# Exkurs: Sync von Daten mit App

- Widget ist eine “Extension”. Extensions laufen unabhängig von der App (eigener Prozess)
- Konsequenz: Daten-Austausch zwischen App und Widget ist nicht ganz trivial
- Einigermassen einfache Lösung: Widget & App in gemeinsamer App Group → können gemeinsame **UserDefaults** verwenden:

```
let userDefaults = UserDefaults(suiteName: "app.group.identifier")!
```

- App Group einrichten ist etwas mühsam... Für Gruppenprojekt aber sicher möglich!

# App-Installation & Provisioning

# App-Provisioning

- Grundsatz: Apple möchte möglichst viel Kontrolle darüber, welche Apps auf iOS-Geräten laufen
  - Dieser Ansatz hat offensichtlich Vor- und Nachteile
- Mit Xcode und Simulator kann man ohne Barriere entwickeln (sofern man einen Mac hat...), aber um App auf einem physischen Gerät laufen zu lassen, ist Developer-Account nötig (mittlerweile gratis)

# App-Provisioning

- **Certificates**: Zertifikat, das auf dem Mac in der Keychain gespeichert wird und benutzt wird, um die App zu signieren
- **Identifiers**: App IDs, die eine App eindeutig identifizieren. Damit sind gewisse Capabilities/App Services verbunden
- **Devices**: Geräte, auf denen man Apps installieren will, müssen ebenfalls registriert werden
- **Profiles**: Provisioning Profiles verknüpfen all diese Infos (Certificate, App ID, Device IDs) und werden mit der App auf dem Gerät installiert. Nur wenn alles korrekt ist, kann die App installiert und geöffnet werden.

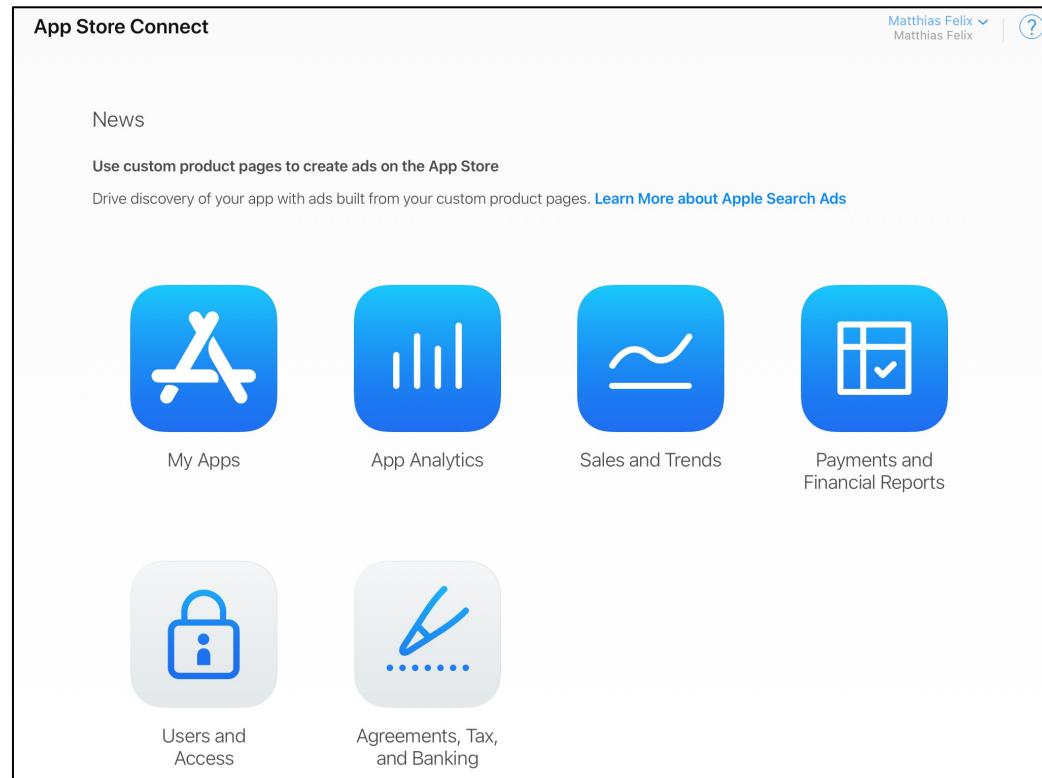
# App-Provisioning

- Prozess war früher sehr mühsam, heute macht Xcode im Idealfall (fast) alles selber
- Man muss für die meisten Schritte nicht mehr ins Developer Portal abspringen. Xcode kann u.a.
  - Zertifikate erstellen und in der Keychain speichern
  - App IDs erstellen
  - Geräte registrieren
  - Neue Capabilities hinzufügen
  - etc...

# App-Veröffentlichung

# App-Veröffentlichung

- Apps werden grundsätzlich über den App Store veröffentlicht
- Dafür nötig: Kostenpflichtiger Developer-Account (ca. 100 CHF/Jahr)
- Verwaltung dieses Prozesses passiert in AppStore Connect:  
<https://appstoreconnect.apple.com>



# AppStore Connect – Demo

# App-Veröffentlichung – Prozess

- App muss für Veröffentlichung mit Distribution Certificate signiert werden
- Release-Konfiguration im Xcode (Build wird nicht für Debugging, sondern für Release optimiert)
- Hochladen zu Apple (via Xcode möglich)
- Automatisiertes Processing
- Je nach Distribution Channel manuelles Review

# App-Veröffentlichung – Testing

- Apple stellt für Beta-Testing eine Plattform zur Verfügung: TestFlight
- Integriert in AppStore Connect
- Erlaubt es, Builds vor der Veröffentlichung an interne oder externe Tester zu verteilen
  - Interne Tester: Auf AppStore Connect registrierte Benutzer, z.B. Teammitglieder etc. (höchstens 100)
  - Externe Tester: Können via Emailadresse oder public link eingeladen werden (bis zu 10'000) → Achtung: App muss durch App Review!
- Installation auf Geräten über TestFlight-App



# App-Veröffentlichung – App Review

- Jede neue App bzw. jedes Update wird von Apple geprüft
- Grundsätzliche Funktionalität; grobe Verstöße aufdecken
- Human Interface Guidelines
- In App Purchases werden immer sehr genau geprüft → Schutz vor Betrug
- App Review Prozess: Sowohl automatisiert als auch manuell
  - Bsp. automatisch: Covid-Strings in einer App
  - Bsp. manuell: Demo ;-)
- Meistens Antwort innerhalb weniger Tage, manchmal sogar Stunden
  - Leider teilweise viel länger, gibt keine Garantie

# Persistenz, Core Data, Tests, Frameworks

Programmieren für iOS



# Inhalt

- Persistenz (Fokus auf Core Data)
- Tests
- Inter-App Kommunikation
- Frameworks
  - User Notifications
  - Swift Charts

# Persistenz

# Persistenz

- Persistenz = “Daten über Programmlaufzeit erhalten”
- Grundsätzlich in einer Datei oder DB
  - Evt. via Web, z.B. Backend, iCloud etc. (würde für Vorlesung zu weit gehen)
- iOS-Mechanismen für lokale Persistenz:
  - Dateisystem (schon gesehen)
  - UserDefaults (schon gesehen)
  - Keychain (nur kurz)
  - **SQLite & Core Data (neu)**

# Dateisystem

- Zugriff auf's Dateisystem wird aus Sicherheits-Überlegungen stark limitiert
- iOS hat keinen Finder / File Explorer
- "Every app is an island" → Jede App hat ihre Sandbox und keinen Lese- / Schreibzugriff auf andere Orte (mit kontrollierten Ausnahmen, z.B. dokumentbasierte Apps)
- Innerhalb der Sandbox kann ziemlich beliebig mit Dateien gearbeitet werden:
  - Lesen / Schreiben
  - Verzeichnisse, Dateien erstellen, löschen etc.



# Dateisystem

- Bevorzugter Weg, um mit dem Filesystem zu interagieren: Klasse **FileManager** (mit Singleton-Instanz **FileManager.default**)
- URL für Documents Directory:

```
func getDocumentsDirectory() -> URL {
 let paths = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
 let documentsDirectory = paths[0]
 return documentsDirectory
}
```

- FileManager hat Methoden für viele gängige Operationen:
  - Verzeichnis erstellen / löschen
  - Prüfen, ob Datei existiert
  - Datei erstellen / schreiben / auslesen

# Dateisystem

- Für generische Objekte: **Codable**-Conformance → mit **JSONEncoder** & **JSONDecoder** encoden / decoden, speichern als JSON-Objekte
- Schreiben:

```
struct Object: Codable {
 let value: String
}
let o = Object(value: "Hello")
let data: Data = try! JSONEncoder().encode(o)
let success = FileManager.default.createFile(atPath: path, contents: data)
```

- Lesen:

```
let data = FileManager.default.contents(atPath: path)!
let o = try! JSONDecoder().decode(Object.self, from: data)
```

# UserDefaults

- System-Mechanismus zum Persistieren kleiner Datenmengen.  
Geeignet u.a. für:
  - Benutzereinstellungen
  - App-Zustand
  - App-Metainformationen (z.B. `isFirstAppStart` o.ä.)
- Achtung: Wird unverschlüsselt gespeichert, darum nie für sensitive Daten verwenden!
- Sehr einfach zu benutzende API, dafür Datentypen beschränkt
  - Wobei `Data` möglich → somit mittels `Codable` beliebige Datentypen

# UserDefaults – API

- Schreiben & lesen:

```
UserDefaults.standard.set(42, forKey: "intValue")
let intValue = UserDefaults.standard.integer(forKey: "intValue")

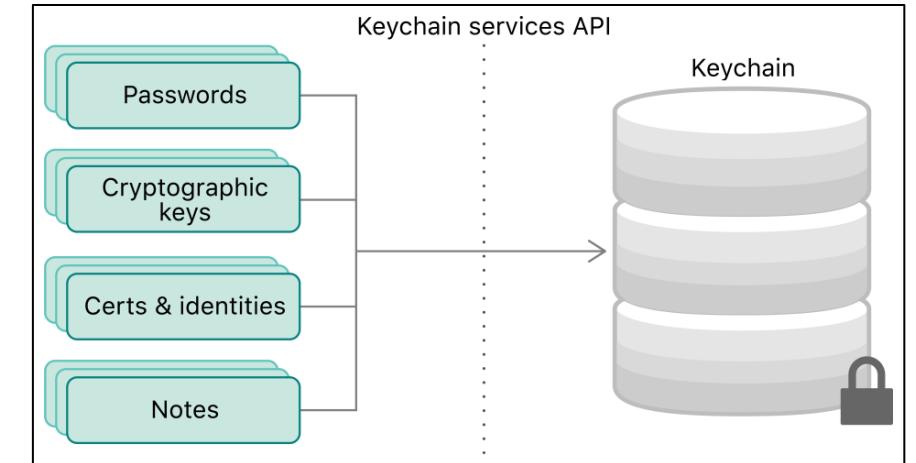
UserDefaults.standard.set(true, forKey: "boolValue")
let boolValue = UserDefaults.standard.bool(forKey: "boolValue")
```

- Für SwiftUI: @AppStore Property Wrapper

```
@AppStorage("editCount") var editCount: Int = 0
```

# Keychain

- Keychain: Kleine Mengen an Userdaten in verschlüsselter DB speichern
- V.a., aber nicht nur für Passwörter
- DB wird zwar (verschlüsselt) auf Filesystem gespeichert, Schlüssel dafür in der Secure Enclave
  - *Secure Enclave in a Nutshell: Dediziertes Hardware-Subsystem, das getrennt vom Main-Prozessor lebt. Dort gespeicherte Keys verlassen die Secure Enclave nie.*

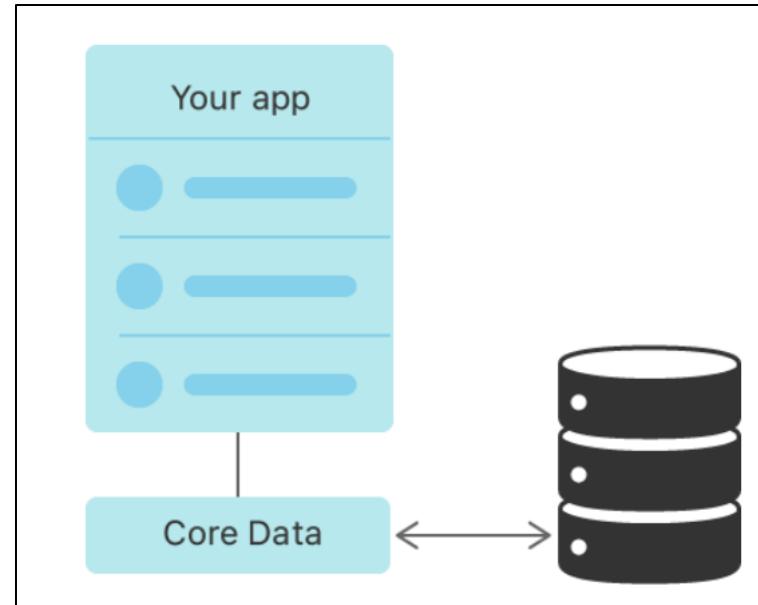


# SQLite

- SQLite: Schlanke SQL-Engine
  - <https://www.sqlite.org/>
  - Open Source
  - Läuft auf jedem Smartphone und fast allen Computern
- Auf iOS ebenfalls verfügbar
  - Gebraucht in diversen Standard-iOS-Apps
  - Wird von Core Data intern ebenfalls verwendet
- Direkt-Verwendung ist theoretisch möglich (C-Library), aber nicht üblich/empfohlen
  - Bevorzugt über Swift-Wrapper-Frameworks verwenden (z.B. Core Data, aber auch andere → GitHub), z.B. wegen Type Safety

# Core Data

- Offizielles Apple-Framework, um beliebige (auch grosse) Datenmengen zu persistieren, cachen und synchronisieren (via iCloud)
- Zusätzliche Abstraktionsebene über klassische Datenbank
  - Intern wird DB benutzt, aber als Framework-User interagiert man nicht direkt damit

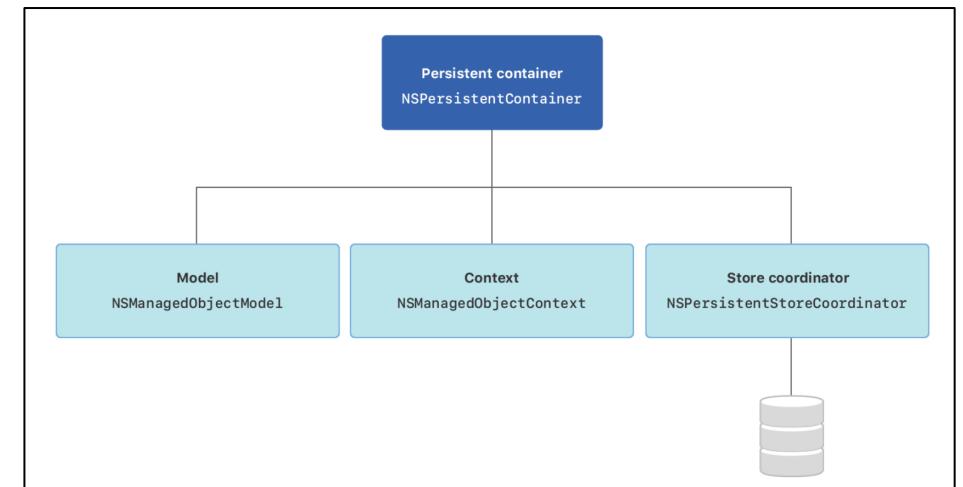


# Core Data

- Unterstützung für diverse Tasks:
  - Undo/Redo-Operationen
  - Batch Changes / Rollbacks
  - Synchronisation mit UI-Layer (Integrationen für diverse Views in UIKit oder SwiftUI), z.B.
    - `NSFetchedResultsController` (UIKit)
    - `@FetchRequest` (SwiftUI)
  - Versionierung & Migration
    - Bei Änderung von Datenmodell: Einfache Migrationen werden wenn möglich automatisch gemacht
    - Komplexere Migrationen: Manuelle Unterstützung notwendig (Mapping-Model)

# Core Data – Aufbau

- **NSManagedObjectModel**: Repräsentation der .xcdatamodeld-Datenmodelle
- **NSManagedObjectContext**: Koordiniert Changes in den Models (primäre Interaktionen passieren auf dieser Klasse: insert, update, save, rollback, ...)
- **NSPersistentStoreCoordinator**: Verantwortlich für Speichern und Lesen von Objekten aus einem Store
- **NSPersistentContainer**: Wrapper, der sich um all diese Objekte kümmert



# Core Data – Features (1/2)

Core Data drastically decreases the amount of code you write to support the model layer. This is primarily due to the following built-in features that you do not have to implement, test, or optimize:

- **Change tracking** and built-in management of **undo** and **redo** beyond basic text editing.
- Maintenance of **change propagation**, including maintaining the **consistency of relationships** among objects.
- **Lazy loading** of objects, partially materialized futures (faulting), and copy-on-write data sharing to reduce overhead.
- **Automatic validation** of property values. Managed objects extend the standard key-value coding validation methods to ensure that individual values lie within acceptable ranges, so that combinations of values make sense.

# Core Data – Features (2/2)

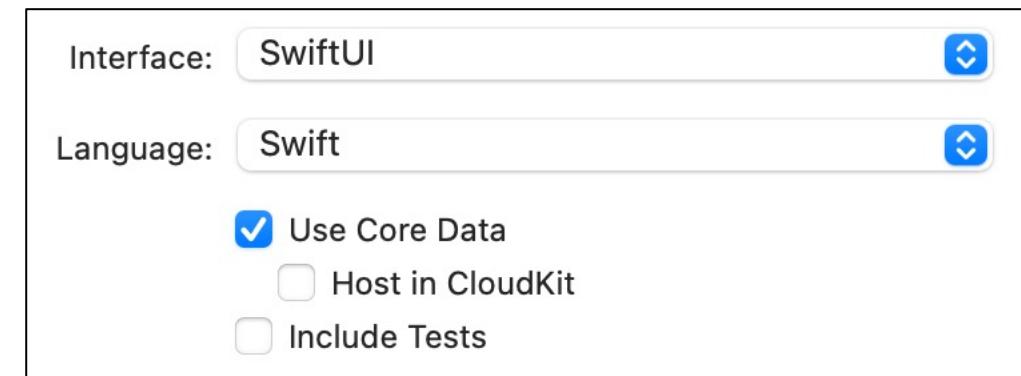
- **Schema migration** tools that simplify schema changes and allow you to perform efficient in-place schema migration.
- Optional integration with the application's controller layer to support user **interface synchronization**.
- **Grouping, filtering, and organizing** of data in memory and in the user interface.
- Automatic support for **storing objects in external data repositories**.
- Sophisticated **query compilation**. Instead of writing SQL, you can create complex queries by associating an **NSPredicate** object with a fetch request.
- **Version tracking and optimistic locking** to support automatic multiwriter conflict resolution.

# Was Core Data *nicht* ist

- Core Data ist keine relationale Datenbank oder ein *Relational Database Management System (RDBMS)*
  - *Core Data provides an infrastructure for change management and for saving objects to and retrieving them from storage. It can use SQLite as one of its persistent store types. It is not, though, in and of itself a database. ...*
- “Core Data is not a silver bullet”
  - Core Data does not remove the need to write code. Although it is possible to create a sophisticated application solely using the Xcode data modeling tool and Interface Builder, for more real-world applications you will still have to write code.

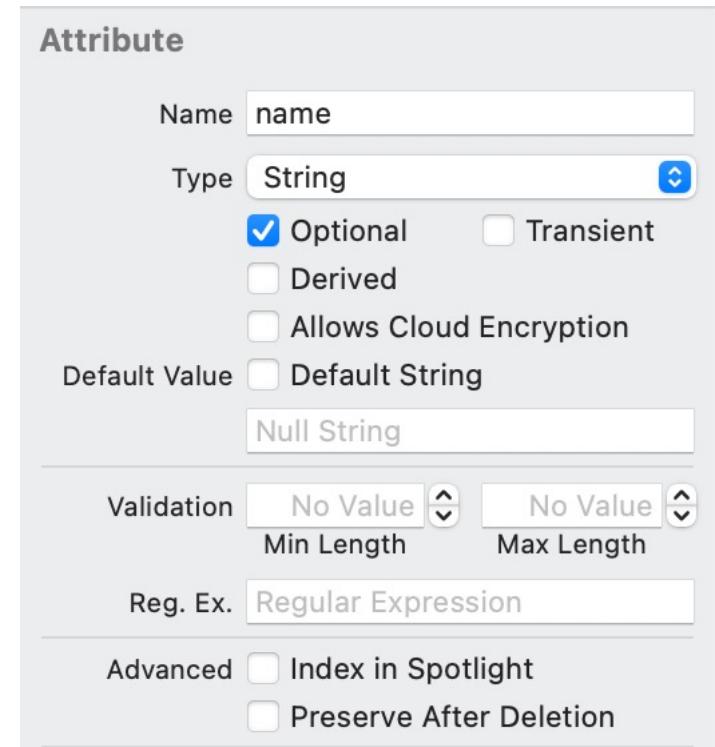
# Core Data: Xcode Template

- iOS App als Template wählen, “Use Core Data” Häkchen setzen
- Generiert ziemlich viel (Boilerplate-)Code und Objekte
  - PersistentController
  - Datenmodell-Datei (.xcdatamodeld)
  - @FetchRequest PropertyWrapper, um Model mit (SwiftUI-)View zu connecten
    - Optional mit Predicate und SortDescriptor



# Datenmodell

- Datei mit Endung .xcdatamodeld
- Visueller Editor für Models
- Primär wichtig: Entities mit...
  - Attributen (Name & Typ, plus zusätzliche Attribute)
  - Relationships (Beziehungen zwischen Entities)
- Swift-Klassen für diese Entities werden automatisch generiert



| Relationships |             |         |
|---------------|-------------|---------|
| Relationship  | Destination | Inverse |
| M teams       | Team        | members |

# Core Data – Take-aways

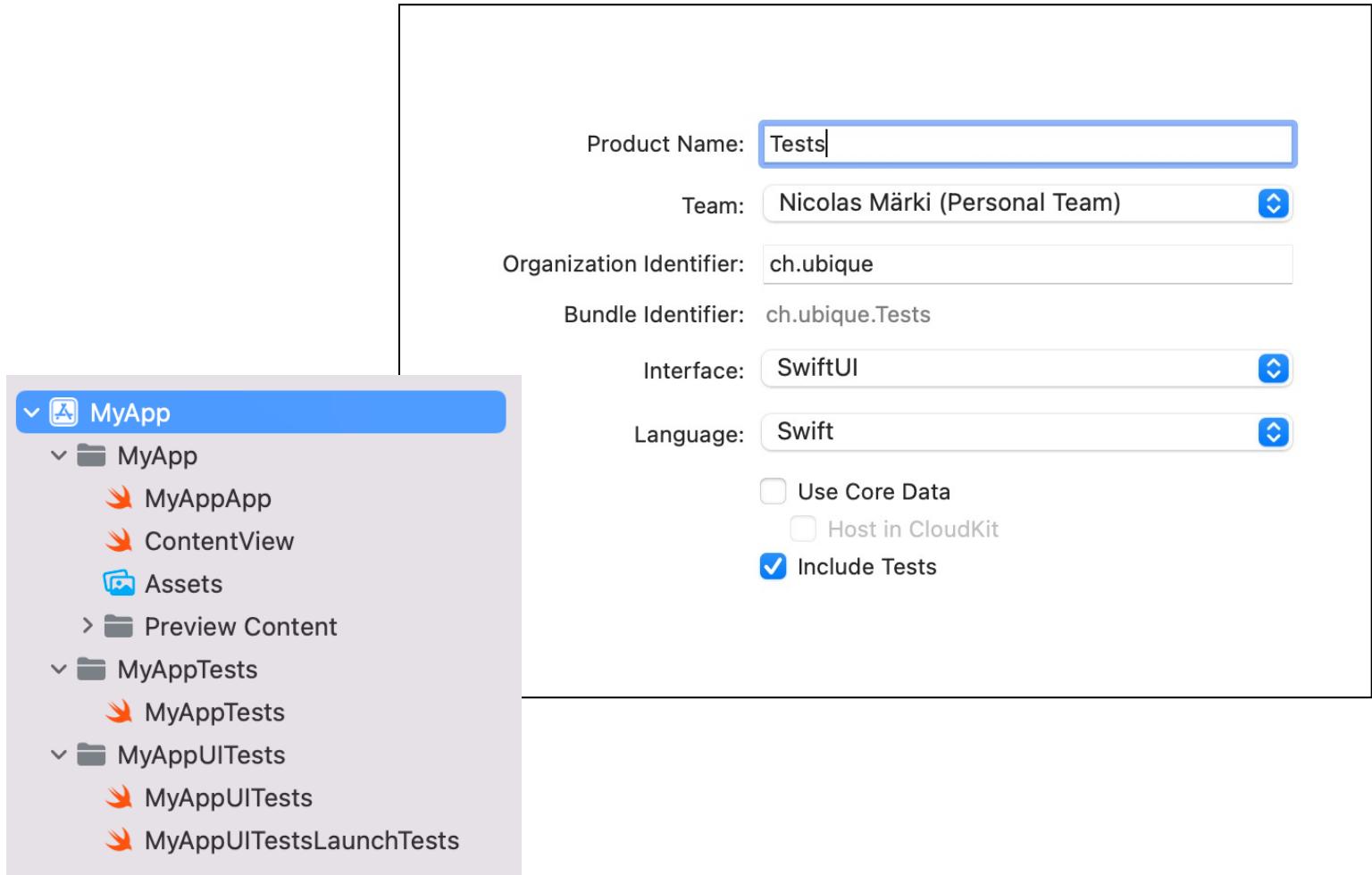
- Core Data ist definitiv keine Entry-Level Technologie. Sehr mächtiges Tool, das aber Zeit braucht, um gelernt zu werden
  - That being said: Mit Projekt-Template und einem nicht allzu komplexen Use-Case durchaus gute Alternative (weil auch andere DB-ähnliche Alternativen nicht unbedingt trivial...)
- Grosser Funktionsumfang, kann interessant sein für diverse Use-Cases
- Core Data ist verhältnismässig “altes” Framework
  - → Gibt relativ viel Support & Dokumentation
  - → Sehr gut getestet

# Core Data – Demo

# Testing

# Testing

- 2 Test-Typen:
  - Unit Tests
  - UI Tests



# Unit Tests mit XCTest

- Xcode bietet Unit Testing an: Prüfen korrektes Funktionieren einzelner Code-Einheiten (“Units”)
  - Projekt-Templates können automatisch Tests inkl. Target erstellen
- <https://developer.apple.com/documentation/xctest>

# Unit Tests

- Klassische Unit-Tests: Test von einzelnen Methoden, Datenstrukturen, deren Zusammenspiel, ...
- Werden losgelöst von der App ausgeführt
  - → d.h., es gibt z.B. kein AppDelegate
  - → Evt. Mock-Objekte nötig
- Tests laufen lassen: Product → Test (oder direkt via Play-Button im Editor)

```
8 import XCTest
9
10 final class HSLUCoreDataUITests: XCTestCase {
11
12 override func setUpWithError() throws {
13 // Put setup code here. This method is
14
15 // In UI tests it is usually best to st
16 continueAfterFailure = false
17 }
```

# Test-Methoden

- Methoden-Name beginnt mit `test`
- Rückgabetyp: `void`
- `setup` & `tearDown`:
  - Diese Methoden werden vor resp. nach jeder Test-Methode ausgeführt

# Assertions

- Diverse Assertions, um Werte zu prüfen
  - XCTAssertTrue
  - XCTAssertEqual
  - XCTAssertNotNil

```
24
25 func testFunction() {
26 XCTAssertEqual(multiply(x: 3, y: 5), 14) ✗ testFunction(): XCTAssertEqual failed: ("15") is not equal to ("14")
27 }
28
```

# UI Tests

- Basierend auf Accessibility API
- Automatisierung von UI Interaktionen
  - Elemente finden
  - Mit der UI interagieren (z.B. Button klicken)
  - Eigenschaften und Status von Elementen validieren
- UI Recording: Quick Start für UI Tests

# Unit vs. UI Tests

- “Interne” vs. “externe” Sicht
- Unit Tests:
  - Zugriff auf Methoden, Funktionen, Variablen und Status der App
- UI Tests:
  - Simulieren von UI-Interaktionen aus User-Sicht in einem separaten externen Prozess

# Unit Tests – Demo

# Inter-App Kommunikation

# Inter-App Kommunikation

- Bereits behandelt: Jede App ist eine Insel (“Sandbox”-Konzept)
  - → Wie funktioniert Austausch mit anderen Apps?
- Verschiedene Möglichkeiten, z.B.:
  - Andere App starten → URL Schemes
  - Document Interaction
    - Z.B. PDF-Dateien
  - Universal Links

# URL Schemes

- System-Methode, um beliebige URLs zu öffnen
- System prüft, ob URL von bestimmter App geöffnet werden kann/soll

```
let url = URL(string: "mailto:ios@hslu.ch")!
UIApplication.shared.open(url)
```

# URL Schemes

- URLs für System-Apps:
  - http(s)
  - mailto
  - tel
  - sms
  - youtube
  - itunes
  - maps
- Gibt's auch für Apps von Drittanbietern
  - Z.B. SBB, Facebook, ...
  - Inoffizielle Listen im Internet

# URL Schemes

- Probleme / Nachteile:

## Warning

URL schemes offer a potential attack vector into your app, so make sure to validate all URL parameters and discard any malformed URLs. In addition, limit the available actions to those that don't risk the user's data. For example, don't allow other apps to directly delete content or access sensitive information about the user. When testing your URL-handling code, make sure your test cases include improperly formatted URLs.

## Note

If multiple apps register the same scheme, the app the system targets is undefined. There's no mechanism to change the app or to change the order apps appear in a Share sheet.

# URL Schemes – Demo

# Lösung: Universal Links

- Öffnen von “normalen” http(s) Links mit der eigenen App
- Meta-File auf dem Server notwendig
  - → Sicherheitsmechanismus, um sicherzustellen, dass nur “eigene” URLs geöffnet werden können
- Bevorzugt gegenüber URL Schemes
- <https://developer.apple.com/documentation/xcode/allowing-apps-and-websites-to-link-to-your-content>

# User Notifications

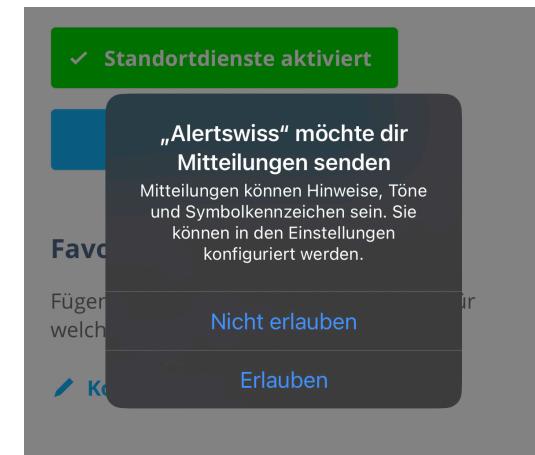
# User Notifications

- Framework für Benachrichtigungen
- Wird verwendet für zwei Arten von Benachrichtigungen:
  - Push-Benachrichtigungen von einem Server
  - Lokale, direkt von der App generierte Benachrichtigungen



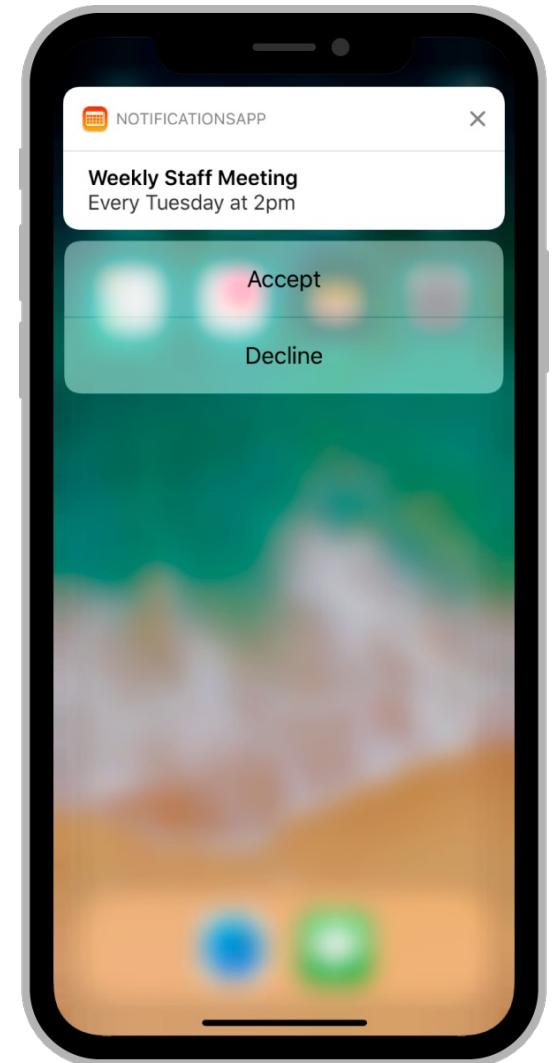
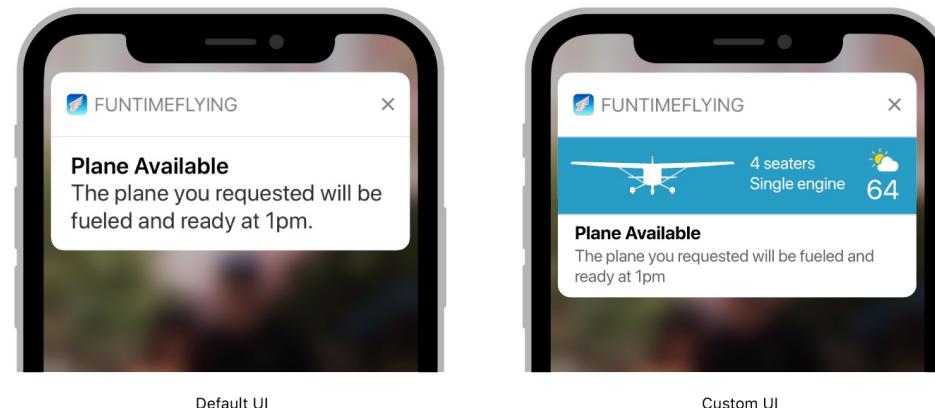
# User Notifications – Grundlagen

- Benachrichtigungen können wichtiger Kanal für eine App sein, um Informationen zu kommunizieren
- Informieren den User, auch wenn die App gerade nicht verwendet wird
- Brauchen explizite Berechtigung → viele User sind eher zurückhaltend (wollen nicht mit Benachrichtigungen “zugespammed” werden)
- HIG zu Notifications: <https://developer.apple.com/design/human-interface-guidelines/components/system-experiences/notifications/>



# User Notifications – zusätzliche Möglichkeiten

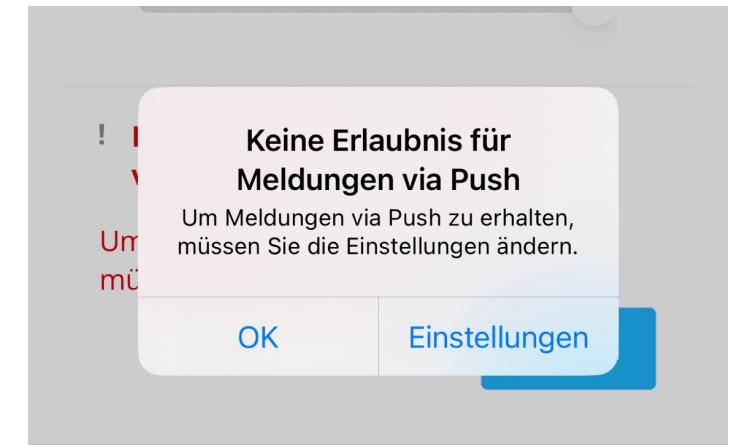
- Man kann Actions definieren → User kann reagieren, ohne App öffnen zu müssen
- Man kann (mittels Extension) Custom UI implementieren (`UNNotificationContentExtension`)
- Ebenfalls mittels Extension: Möglichkeit, vom Server erhaltene Push Notification zu ändern (`UNNotificationServiceExtension`)



# User Notifications – Permission

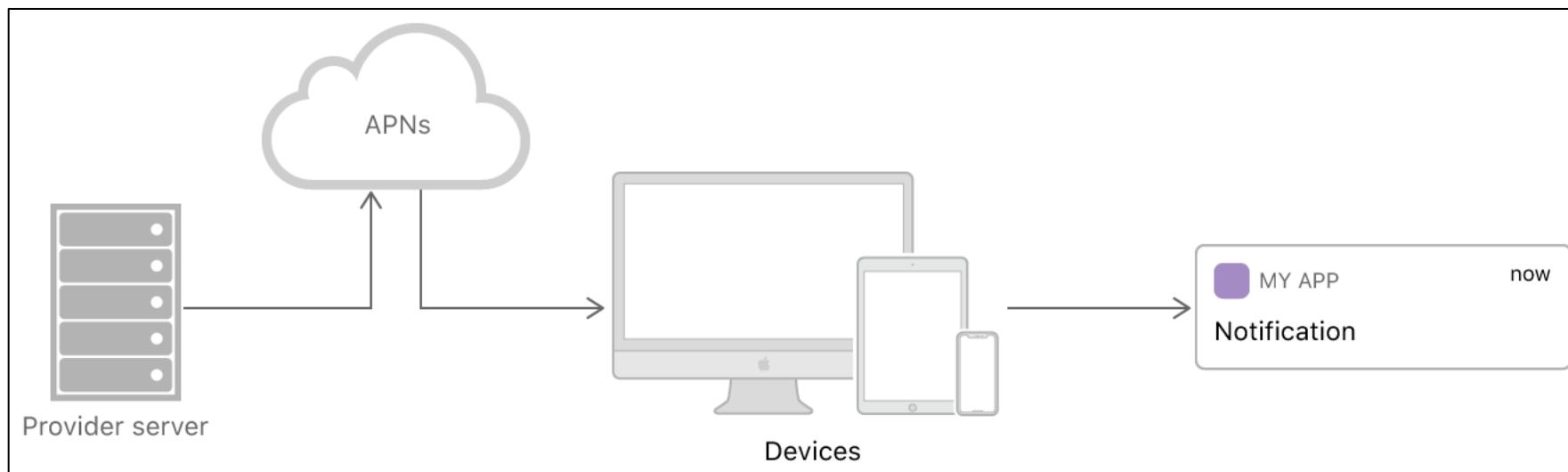
- Grundsatz: Eine App darf 1x nach Erlaubnis fragen, danach nicht mehr
- User kann “Entscheidung” jederzeit in den Einstellungen anpassen
- Häufiger Flow: Fehlermeldung, falls Permission abgelehnt → inkl. Absprung in die Einstellungen

```
let center = UNUserNotificationCenter.current()
center.requestAuthorization(options: [.alert, .sound, .badge]) { granted, error in
 if let error = error {
 // Handle the error here.
 }
 // Enable or disable features based on the authorization.
}
```



# User Notifications – Remote

- Architektur:
  - App Server: Verantwortlich für Business-Logik (wann muss Push an welche User versendet werden?)
  - APNs – Apple Push Notification service: Verantwortlich für's Ausstellen von Push Tokens sowie Versenden von Push-Nachrichten an Devices

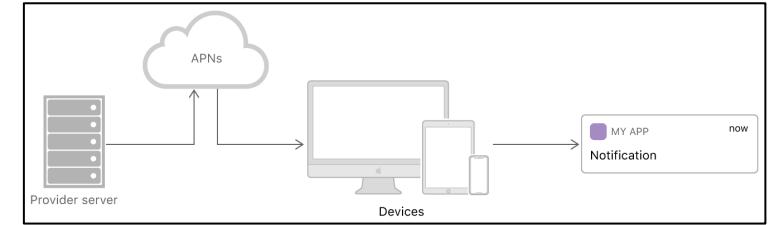


# User Notifications – Remote

- Flow:
  - App erlangt Permission für Benachrichtigungen vom User
  - App fragt nach Token via System-Call zu APNs

```
UIApplication.shared.registerForRemoteNotifications()
```
  - App erhält Token und schickt es an App-Server

```
func application(_ application: UIApplication, didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
 // Send token to app server...
}
```
  - App-Server merkt sich Token (z.B. in DB)
    - mit User verknüpft, z.B. via Account, Unique ID, ...
  - Push-Benachrichtigung senden: App-Server schickt Inhalt und Token zu APNs  
→ Device erhält Push “direkt von Apple”



# User Notifications – Local

- Aus Usersicht kein Unterschied erkennbar → Unterschied zu remote Notifications: Zeitpunkt & Inhalt werden direkt von der App gemanaged
- Notifications werden geschedulet, entweder für sofortige Delivery oder optional mit Trigger
- 3 Arten von Triggers:
  - [UNCalendarNotificationTrigger](#)
  - [UNTimeIntervalNotificationTrigger](#)
  - [UNLocationNotificationTrigger](#) (Achtung: braucht wiederum Location-Permission...)
- Geschedulete Notifications können auch canceled werden

# User Notifications – Demo

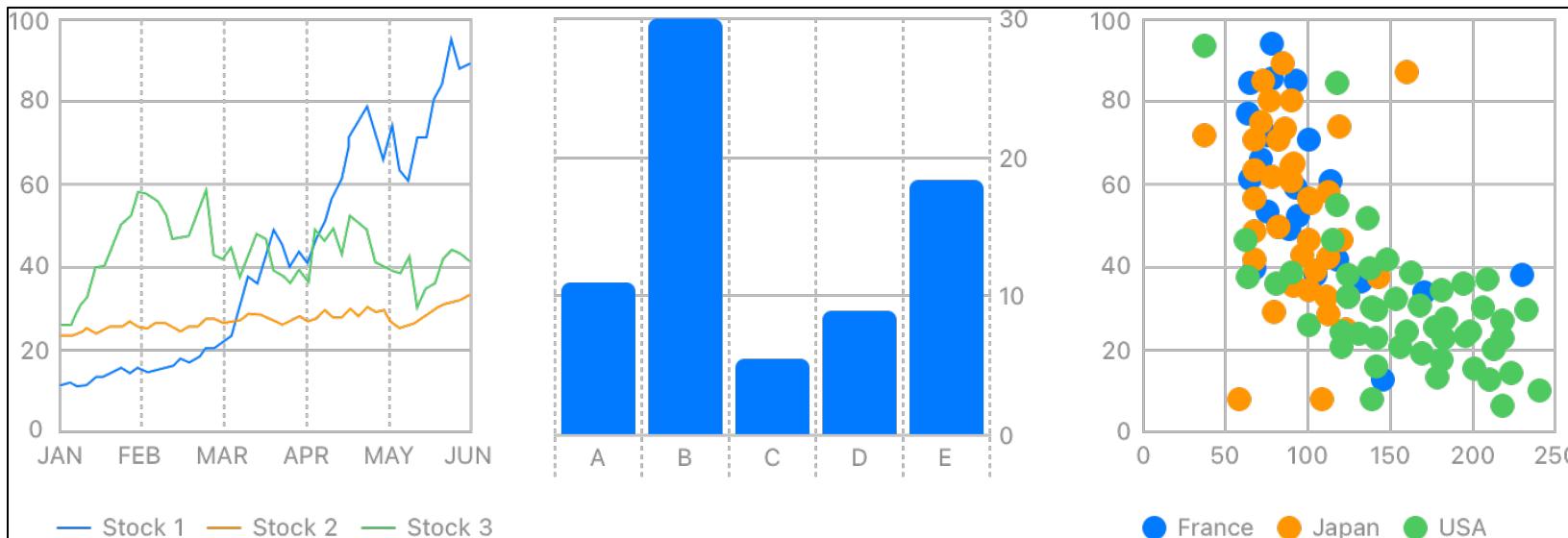
# Exkurs: Silent Push Notifications

- Server kann Notifications auch “silent” schicken → App bekommt Push-Nachricht, ohne dass User etwas mitbekommt
- Beispiel UZH: Entscheiden, ob User sich tatsächlich in für Alarm relevanter Region befindet → Locations müssen nicht an Server gesendet werden
  - Alertswiss hat ähnliche Logik für Meldungen

# Swift Charts

# Swift Charts

- Neues SwiftUI-Framework (WWDC 2022, iOS 16)
- Sehr intuitiv, SwiftUI-ige deklarative Syntax
- Interessant für einfache, aber auch komplexe Daten-Visualisierungen



# Swift Charts – Einführung

- Wichtig bei Daten-Visualisierung:
  - Welche Daten habe ich?
  - Was möchte ich kommunizieren?
  - Was ist die geeignete Darstellung dafür?
- Wäre wohl eigene Vorlesung wert...

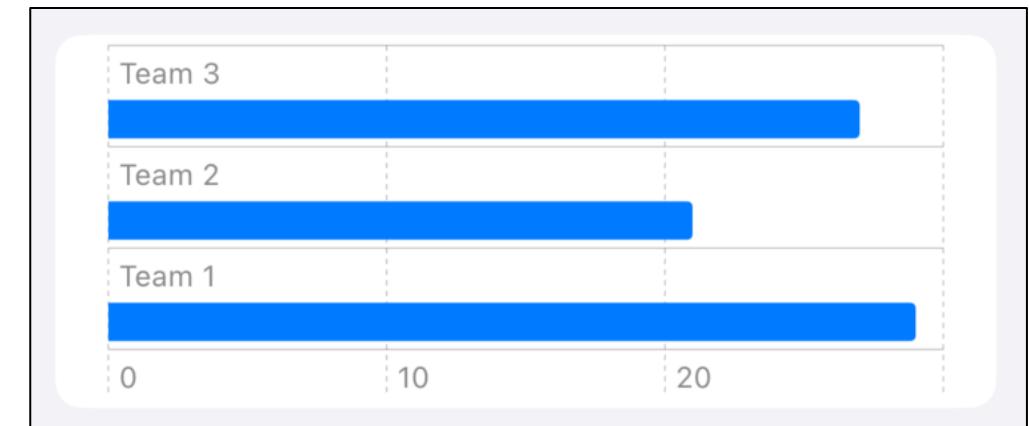
# Swift Charts – Syntax

- Chart als Wrapper mit verschiedenen “Marks”
  - BarMark, LineMark, PointMark, RectangleMark, AreaMark, ...
- Meistens mit ForEach, kann aber auch explizit mit einzelnen Marks erstellt werden
  - Oder Syntax-Vereinfachung:

```
Chart {
 ForEach(teams, id: \.id) { team in
 BarMark(x: .value("Points", team.points),
 y: .value("Name", team.name))
 }
}
```

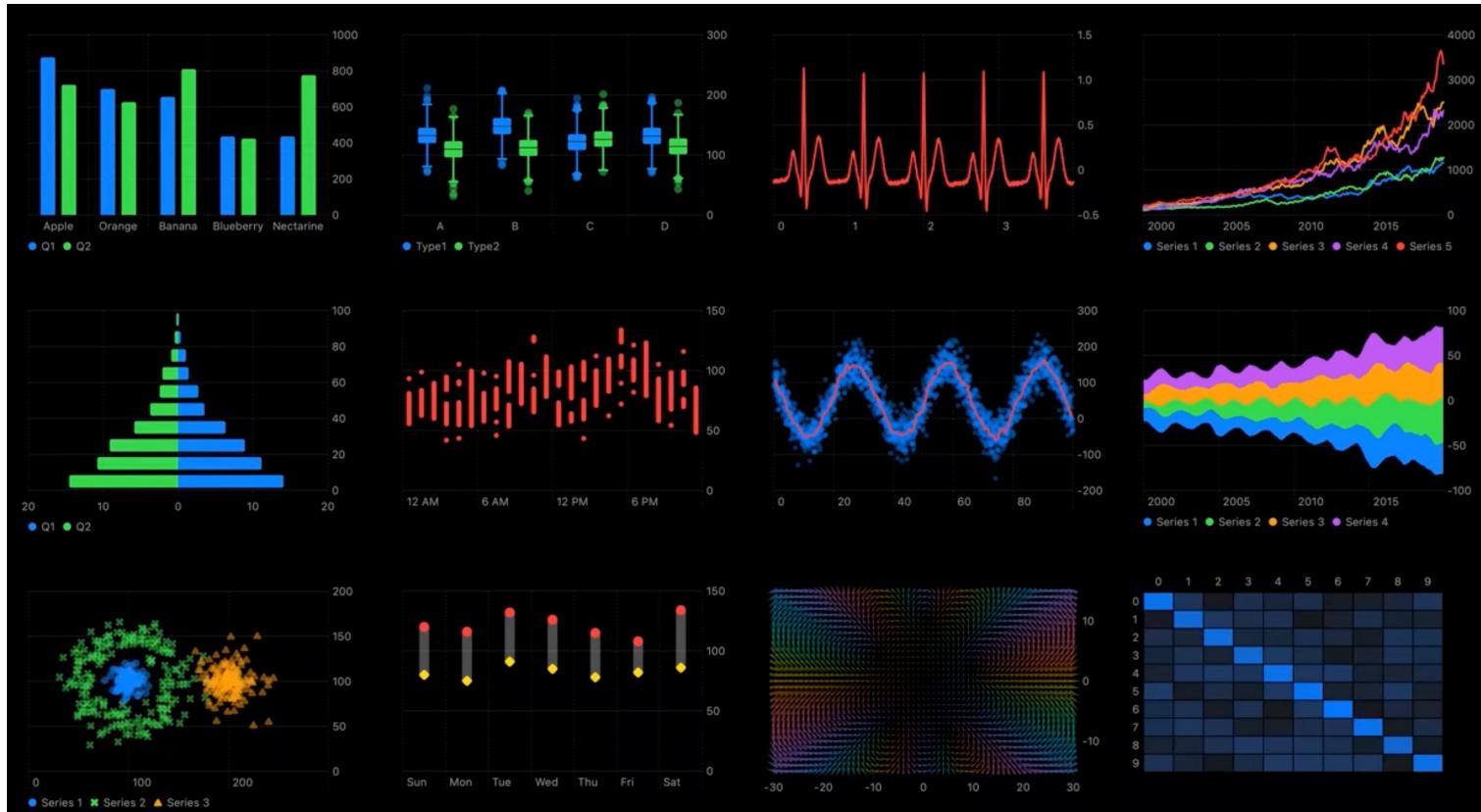


```
Chart(teams) { team in
 BarMark(x: .value("Points", team.points),
 y: .value("Name", team.name))
}
```



# Swift Charts – Syntax

- Diverse Modifiers, um Style und Funktionalität anzupassen
  - Achsen, Legende, Farben, Styles, interaktive Elemente, ...



# Swift Charts – Ressourcen

- <https://developer.apple.com/documentation/charts>
- Sample-Projekt mit vielen Beispiel-Charts:  
[https://developer.apple.com/documentation/charts/visualizing your app s data](https://developer.apple.com/documentation/charts/visualizing_your_app_s_data)
- WWDC-Videos:
  - <https://developer.apple.com/videos/play/wwdc2022/10136/>
  - <https://developer.apple.com/videos/play/wwdc2022/10137/>

# Swift Charts – Demo

# Frameworks: Round-Up

- Es gibt eine Vielzahl spannender Frameworks / Technologien im iOS-Universum
- <https://developer.apple.com/documentation/technologies>
- Alle kennen: praktisch unmöglich
  - Viele sind sehr spezifisch / advanced, haben klaren Use Case
- Liste kann evt. als Inspirationsquelle für Gruppenprojekt dienen
  - Wenn möglich: Sample-Projekt downloaden und rumspielen

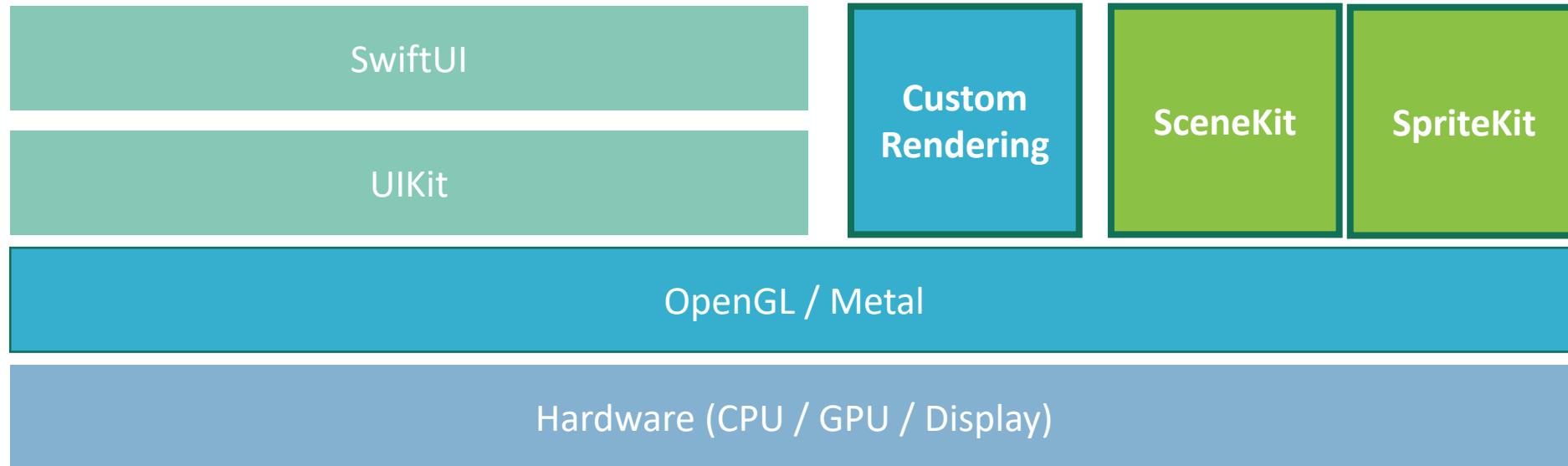
# Graphics

## Programmieren für iOS



# iOS Graphics

- Bisher: Bildschirm mit UI-Elementen (SwiftUI / UIKit) füllen
- Heute: Wie kommen die Pixel wirklich auf den Screen (stark vereinfacht)



# Low-Level Graphics

- Heute: Wie befülle ich einen Bereich im Speicher («Buffer») mit Zahlen zwischen 0 und 1, damit es schöne Farben gibt?
- Annahme: Hardware stellt die Werte dann irgendwie dar.



|     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0.8 | 0.3 | 0.8 | 1.0 | 0.9 | 0.7 | 1.0 | 1.0 |
| 0.8 | 0.3 | 0.8 | 1.0 | 0.9 | 0.7 | 1.0 | 1.0 |
| 0.8 | 0.3 | 0.8 | 1.0 | 0.9 | 0.7 | 1.0 | 1.0 |
| 0.0 | 0.9 | 0.9 | 1.0 | 0.9 | 0.7 | 1.0 | 1.0 |
| 0.0 | 0.9 | 0.9 | 1.0 | 0.9 | 0.7 | 1.0 | 1.0 |
| 0.0 | 0.9 | 0.9 | 1.0 | 0.9 | 0.7 | 1.0 | 1.0 |
| 0.0 | 0.9 | 0.9 | 1.0 | 0.9 | 0.9 | 0.3 | 1.0 |
| 0.0 | 0.9 | 0.9 | 1.0 | 0.9 | 0.9 | 0.3 | 1.0 |

# CoreGraphics

- Buffer (Arrays) können mit normalem Swift-Code gefüllt werden.
- Das Framework **CoreGraphics** bietet viele nützliche Vereinfachungen.
- Aber: Beides ist für grosse Bilder oder hohe Framerates **zu langsam**.
- Lösung: GPU!

```
let buffer = ... // initialize memory

for i in 0 ..< UIScreen.main.size.width {
 for j in 0 ..< UIScreen.main.size.height {
 buffer.setColor(i,j, computeColor(i, j))
 }
}

let img = buffer.toImage()
```

```
let renderer = UIGraphicsImageRenderer(size: UIScreen.main.size)

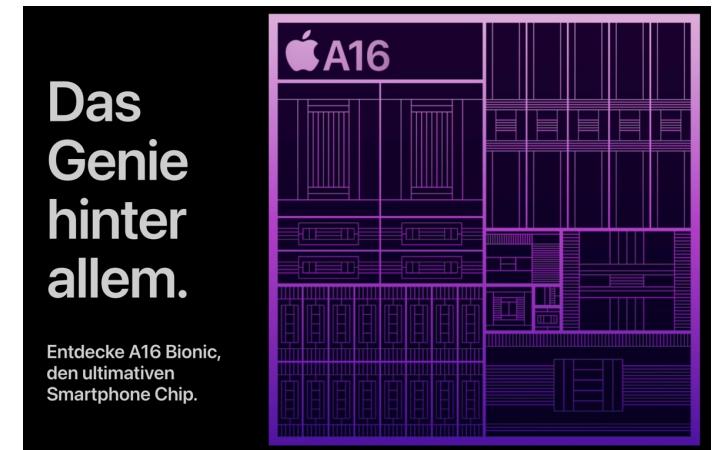
let img = renderer.image { ctx in
 ctx.cgContext.move(to: CGPoint(x: 20.0, y: 20.0))
 ctx.cgContext.addLine(to: CGPoint(x: 260.0, y: 230.0))
 ctx.cgContext.addLine(to: CGPoint(x: 100.0, y: 200.0))
 ctx.cgContext.addLine(to: CGPoint(x: 20.0, y: 20.0))

 ctx.cgContext.setLineWidth(10)
 ctx.cgContext.setStrokeColor(UIColor.black.cgColor)

 ctx.cgContext.strokePath()
}
```

# CPU vs. GPU

- Obwohl alles auf einem Chip zusammengeführt wird, haben **CPU** (Central Processing Unit) und **GPU** (graphics processing unit) deutlich andere Fähigkeiten und Aufgaben.
- Die Architektur von GPUs ist darauf ausgelegt, viele einfache Operationen schnell und parallel auszuführen.
- A15 GPU: 1224 GFLOPS
  - 1 GFLOPS = 1 Milliarde Fließkomma-Operationen pro Sekunde
- A15 CPU: 2 x 3 GHz (2 High-Performance Cores)
  - + Low-Performance Cores

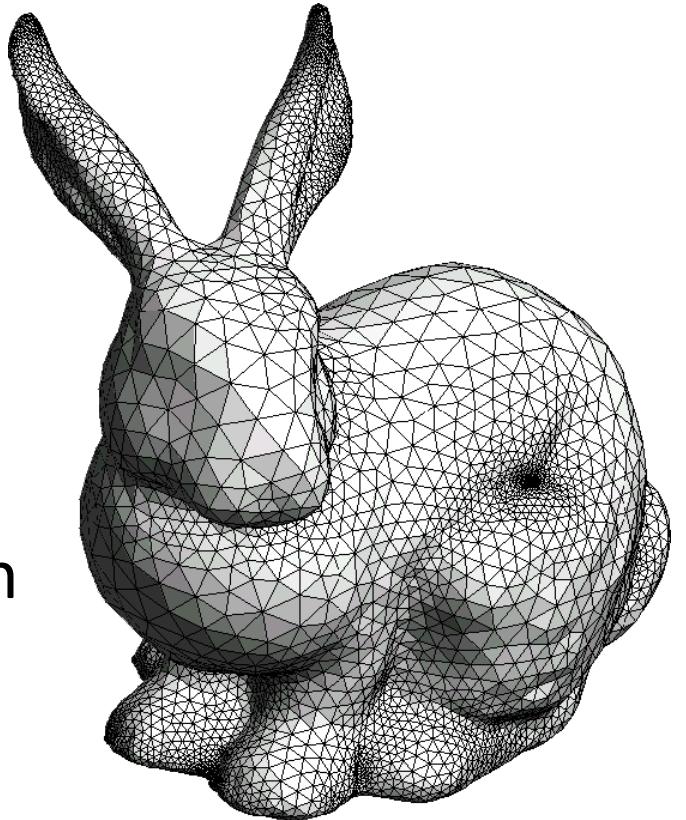
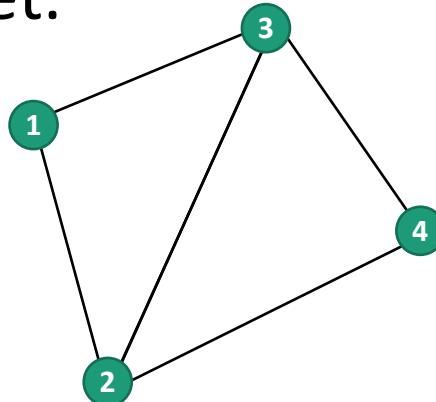


# Graphics Pipeline – Was macht eine GPU?

- Die GPU führt in wesentlichen diese drei Schritte aus:
  1. Geometry Processing
  2. Rasterizer
  3. Fragment Processing

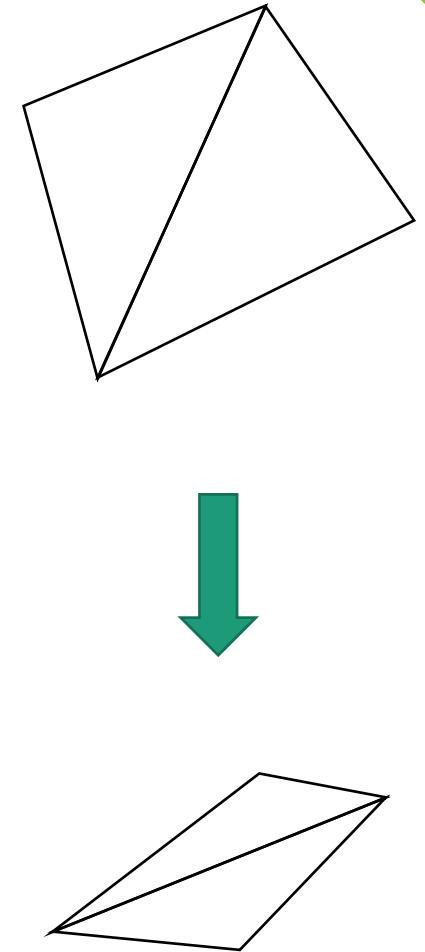
## Datengrundlage:

- Objekte auf der GPU werden fast ausschliesslich durch Dreiecke repräsentiert. Deren Ecken werden als **Vertices** bezeichnet.
- **Mesh:** 3D-Modell aus Dreiecken oder Polygonen.



# 1. Geometry Processing

- Vorbereitung: CPU kopiert Koordinaten und weitere Daten der Vertices (2D oder 3D) auf GPU in einen Buffer. Koordinaten sind in lokalem Koordinatensystem des Objektes definiert.
- Processing für jeden Vertex:
  - Koordinaten werden mit Matrix-Multiplikationen zu Positionen auf Bildschirm umgerechnet.
    - Lokal -> Welt -> Kamera -> Projektion
  - Clipping: Dreiecke, die ausserhalb vom Bildschirm landen, werden ignoriert.
  - Zusatzinformationen pro Vertex können berechnet werden.



# Matrix-Multiplikationen

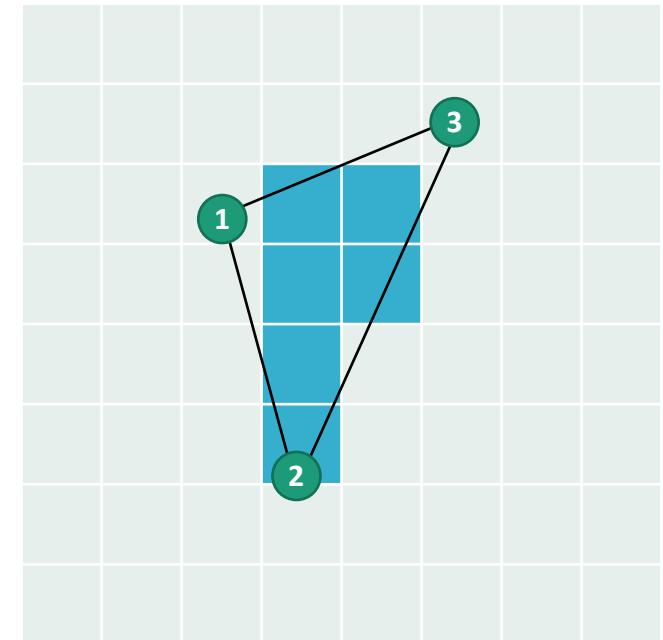
- Mini-Refresher: Verschiebung, Rotation, Skalierung etc. lassen sich sehr gut als Matrizen repräsentieren.

$$\begin{array}{c} \text{Verschiebung} \\[1ex] \left( \begin{array}{cccc} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right) * \left( \begin{array}{c} 1 \\ 2 \\ 3 \\ 1 \end{array} \right) = \left( \begin{array}{c} 6 \\ 2 \\ 3 \\ 1 \end{array} \right) \\[1ex] \text{Position} \end{array}$$

- I.d.R. eine Matrix für relative Position von Objekt, eine für Kamera-Position, eine für Projektion von 3D auf Screen
- Endresultat = Position auf Screen
- Funktioniert auch in 2D (bsp. ScrollView)

## 2. Rasterizer

- Dreieck wird in Pixels (**Fragments**) aufgeteilt.
- Zentrum des Pixels muss innerhalb vom Dreieck liegen. Jedes Pixel (auch auf Kante) wird nur einem Dreieck zugeordnet.
- Interpolation: Zusatzinformationen der Vertices werden für jedes Fragment interpoliert.



## 3. Fragment Processing

- Pro Fragment wird eine beliebige Farbe berechnet. Häufig werden dafür konstante Farben, Sampling und Lighting kombiniert.
- Texture Sampling: Farbe eines Pixel von einem Bild (**Textur**) auslesen.
- Lighting / Shading: Original-Farbe basierend auf Einfallwinkel und Intensität des Lichtes modifizieren.

# Graphics Programming

- Erste GPUs: Fixes Geometry Processing und wenige Operationen für Fragment Processing.
- Heute: Geometry und Fragment Processing der GPU kann programmiert werden. Diese Programme werden **Vertex Shader** und **Fragment Shader** genannt.
- Die GPU wird mit einem Framework gesteuert und programmiert, dafür stehen je nach Plattform unterschiedliche Optionen zur Auswahl:
  - **OpenGL (iOS, Android, Windows, macOS, Web)**
  - **Metal (iOS, macOS)**
  - DirectX (Windows, Xbox)
  - Vulkan (Windows, Linux, macOS, Android)
  - Gnm, Gnmx (Playstation)

# OpenGL / Metal



- 1992 veröffentlicht
- Meistverbreitete Lösung
- Seit iOS 12 deprecated
- GLSL (OpenGL Shading Language)
  - Basiert auf C
  - Runtime-compiled Strings
- Globale Funktionen und State Machines



- 2014 von Apple veröffentlicht
- Für Apple-Hardware optimiert
- Im Allgemeinen performanter
- MSL (Metal Shading Language)
  - Basiert auf C++
  - Precompiled
- Objekt-orientierte API



Demo

# SpriteKit / SceneKit

- Fazit: OpenGL / Metal == Viel Boilerplate-Code
- Die beiden Frameworks **SpriteKit** und **SceneKit** basieren auf Metal und ermöglichen High-Performance Rendering für Spiele, ohne dass man sich mit Metal herumschlagen muss.



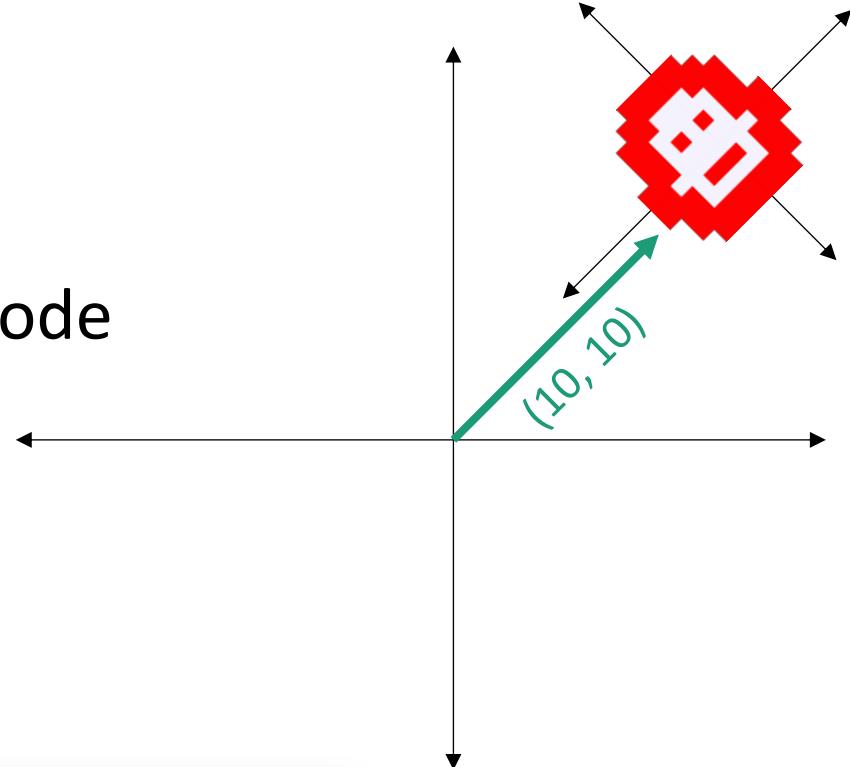
**SpriteKit** (2D)



**SceneKit** (3D)

# SpriteKit

- Die Anwendung platziert **Sprites** (Grafikobjekte) und andere Objekte als Node in einer Szene
- Framework erstellt und aktualisiert alle Buffers, Dreiecke, Texturen, Shaders, etc.



```
let image = SKSpriteNode(imageNamed: "sprite.png")
image.position = CGPointMake(10, 10)
image.zRotation = .pi / 4.0

// Add the image to the scene.
scene.addChild(image)
```

# Actions & Physics

- Nodes können mit **Actions** vielseitig animiert und modifiziert werden
- Falls den Nodes ein **Physics Body** zugewiesen wird, simuliert SpriteKit Kräfte, Kollisionen, etc.

```
spinnyNode.run(SKAction.repeatForever(
 SKAction.rotate(byAngle: .pi, duration: 1)
)

spinnyNode.run(SKAction.scale(by: 0.1, duration: 2))

spinnyNode.run(SKAction.sequence([
 SKAction.wait(forDuration: 0.5),
 SKAction.fadeOut(withDuration: 0.5),
 SKAction.removeFromParent()
])))
```

```
let body = SKPhysicsBody(rectangleOf: CGSize(width: w, height: w * 2))
body.mass = 100
body.friction = 0.3
spinnyNode.physicsBody = body

body.applyImpulse(CGVector(dx: 10, dy: 0))
body.applyTorque(0.3)
```

# SceneKit

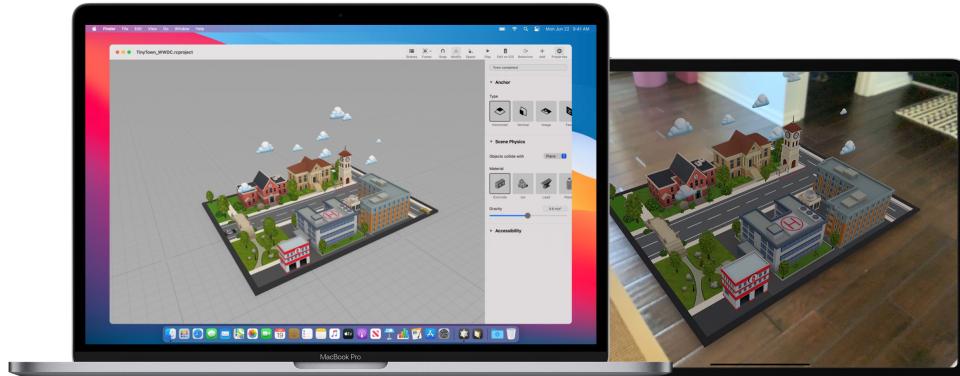
- Die gleichen Grundkonzepte wie SpriteKit: Nodes, Actions, Physics, ...
- Viele weitere APIs für 3D-Meshes, Licht, Rendering, etc.



Demos

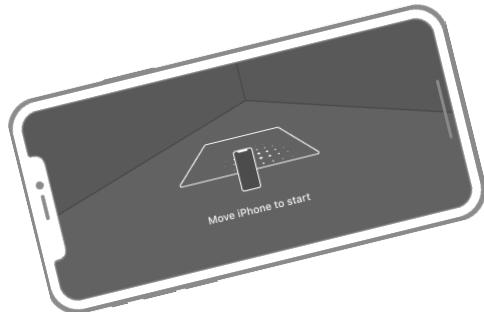
# Extended Reality

- Unter dem Begriff **Extended Reality (XR)**, werden Virtual und Augmented Reality zusammengefasst
- **Virtual Reality (VR)**: Simulierte Welt durch 3D-Rendering in Echtzeit, in der Regel mit 3D-Brillen
- **Augmented Reality (AR)**: Kombination von echten und virtuellen Inhalten, häufig optisch (modifiziertes Kamerabild oder teil-transparenter Bildschirm, aber auch akustisch, haptisch, etc.)



# ARKit

- SpriteKit, SceneKit und die meisten Metal-Anwendungen verwenden das Konzept einer **Kamera**, deren Position und Rotation die Projektion von globalen Koordinaten auf den Screen definieren.
- Das Framework **ARKit** berechnet aus den Kameras, Beschleunigungs- und Rotations-Sensoren eine virtuelle Kamera, mit der die Illusion von Objekten im Raum gerendert werden kann.

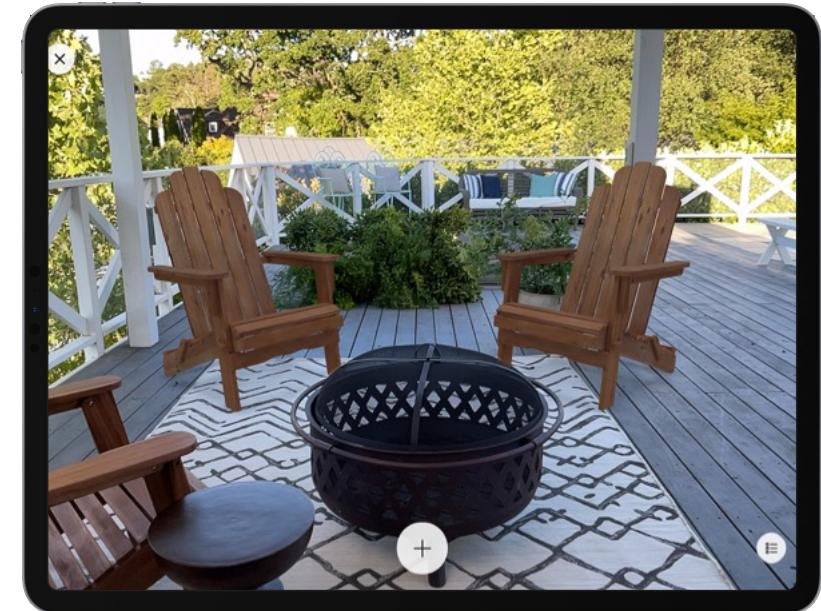


$$\begin{pmatrix} 0.3 & 0.5 & 0.1 & 1.4 \\ 3.4 & 0.9 & 4.4 & 2.3 \\ 7.5 & 5.2 & 0.6 & 0.2 \\ 0.1 & 0.6 & 1.4 & 1.0 \end{pmatrix}$$

- Rendering mit einem beliebigen Framework: Metal, SpriteKit, SceneKit, oder **RealityKit**.

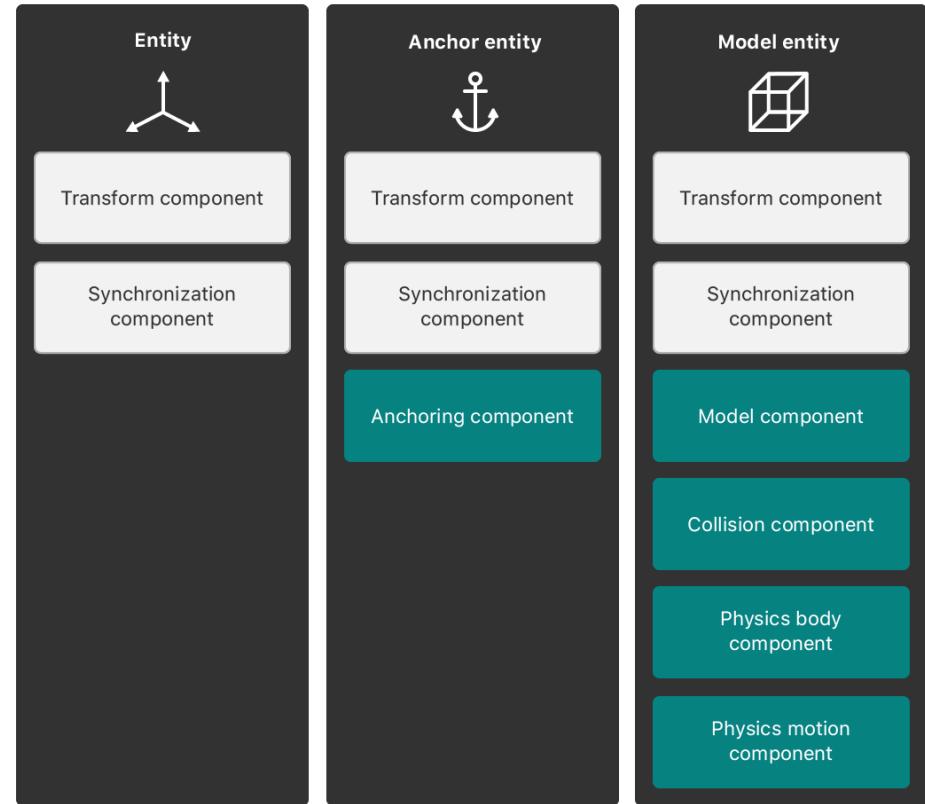
# RealityKit

- Neues, speziell für AR entwickeltes 3D-Graphics-Framework.
- Grundfunktionen wie SceneKit: 3D-Meshes, Animationen, Physik, ...
- Rendering-Effekte für mehr Realismus
  - Körnung
  - HDR
  - Grund-Schatten
  - Motion-Blur
  - Tiefenschärfe
  - Okklusion von Personen
  - Beleuchtung



# ECS

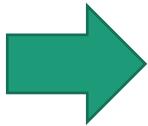
- RealityKit modelliert Objekte als Entities nach dem Software-Architektur Pattern **Entity-Component-System (ECS)**.
- Statt komplexer Vererbung wird die einfache Klasse **Entity** mit verschiedenen **Components** erweitert.
- Ein **System** agiert auf allen Entities, die bestimmte Components enthalten.
  - Model → Rendering
  - Physics Body → Schwerkraft



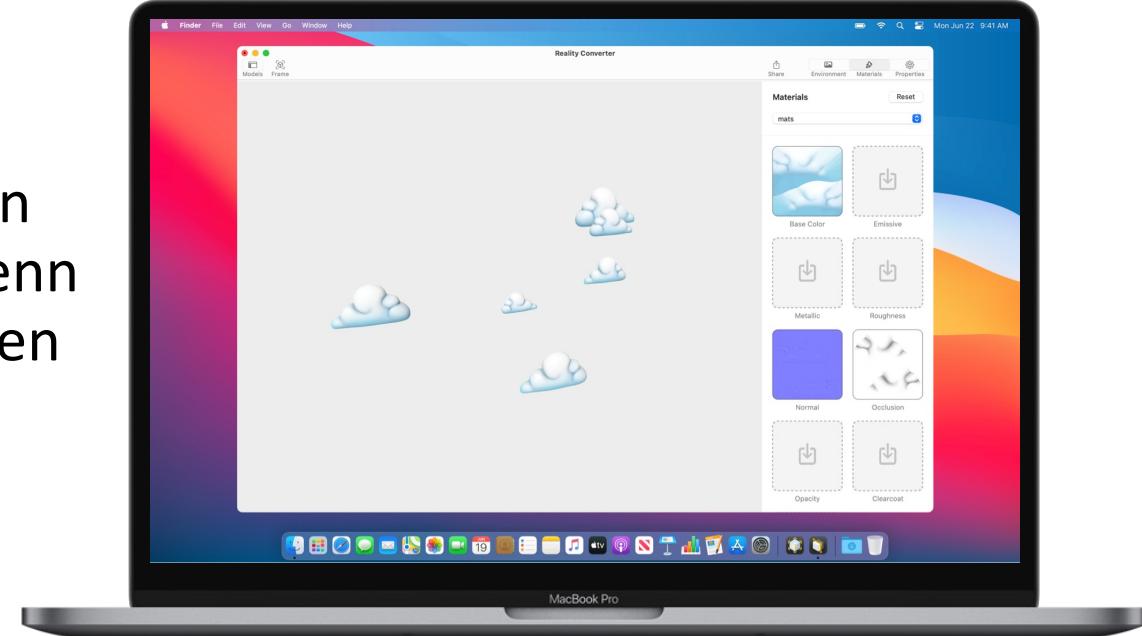
```
let entity = ModelEntity(mesh: .generateSphere(radius: 0.25))
entity.components[FlyingComponent.self] = FlyingComponent()
entity.components[SwarmComponent.self] = SwarmComponent()
```

# AR Creation Tools

- Komplete AR-Welt kann in Code erstellt werden, zusätzliche Tools vereinfachen den Prozess
  - Reality Converter: 3D-Objekte für AR optimieren
  - Object Capture: macOS-API, aus Bildern ein 3D-Model erstellt werden kann, wenn genügend Winkel aufgenommen wurden
  - Reality Composer: AR Szene einfach «zusammenklicken».

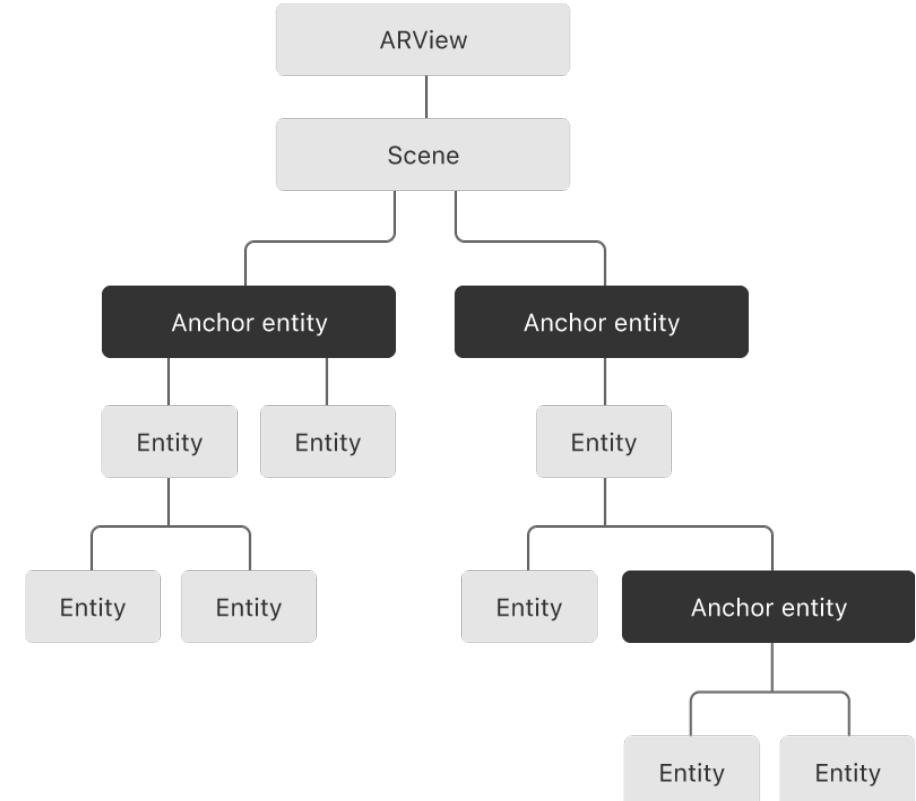


Demo



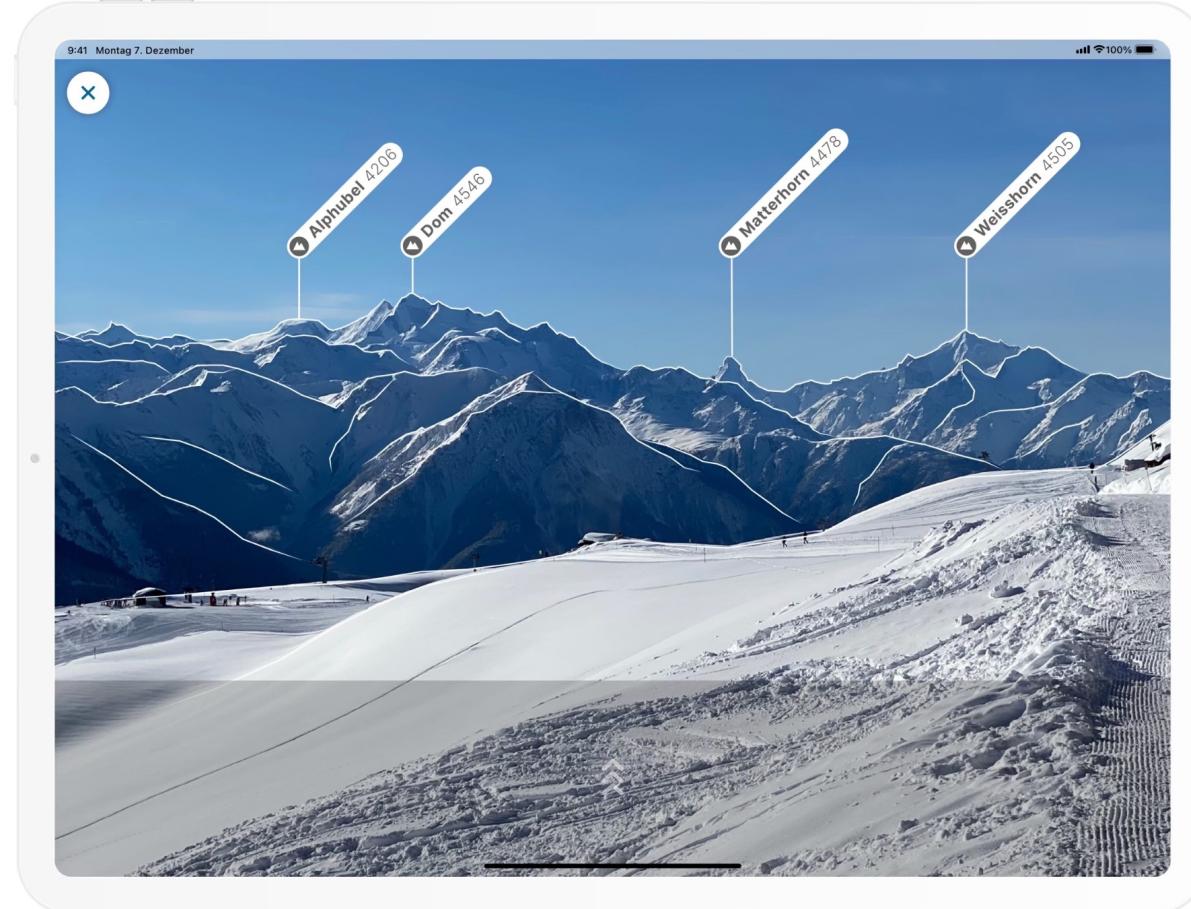
# Anchors

- Virtuelle Objekte werden mit **Anchors** in der Szene platziert.
- Verschiedene Subklassen
  - Plane: Horizontale/vertikale Ebene
  - Object: Referenz 3D-Modell
  - Image: Referenz 2D-Bild
  - Mesh: Rekonstruktion eines Objektes
  - Face: Gesicht in Front-Kamera
  - Body: Körper/Skeleton
  - Participant: Andere User
  - GeoAnchor: Koordinaten



# GeoAnchor

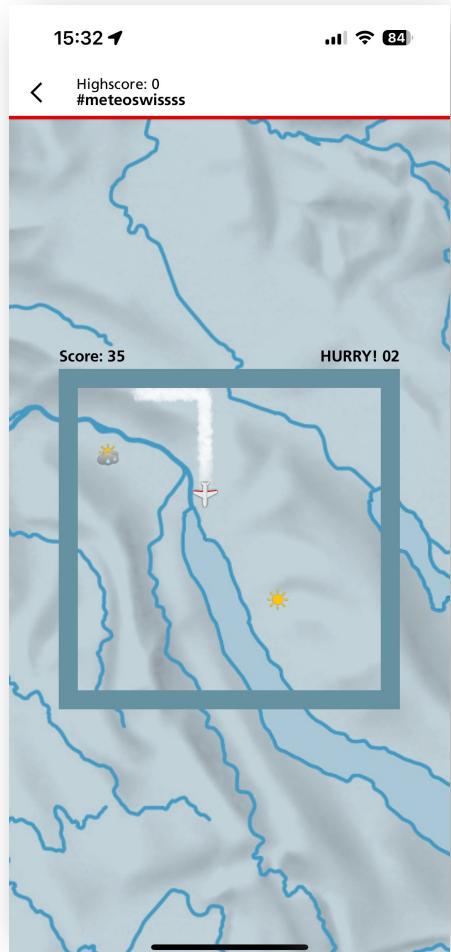
- Achtung: Nur in einigen Städten (primär USA) verfügbar!
- iOS kombiniert GPS und Kompass mit «Street View» - Aufnahmen für genaue Position.
- Beispiel swisstopo App: GPS und Kompass manuell zu Transformations-Matrix umrechnen (ungenauer).



# «Zusammenfassung»



SwiftUI



OpenGL



SpriteKit



SceneKit