

# 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

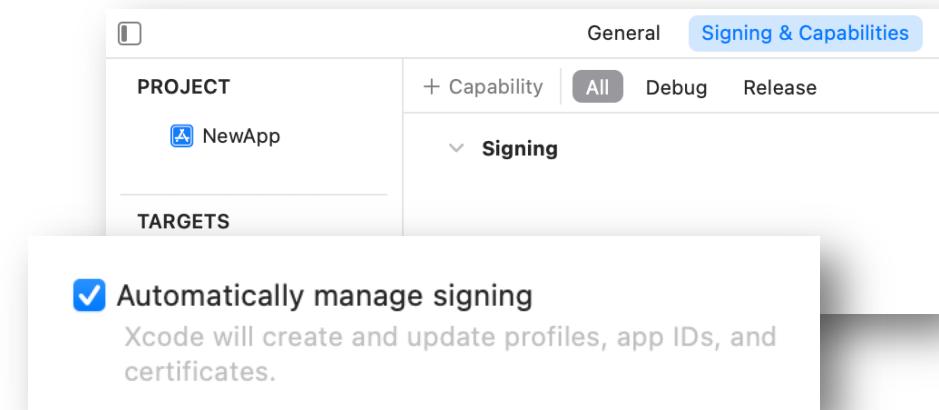
```
// UserStateManager.swift
class UserStateManager {
    var isLoggedIn = false
}

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

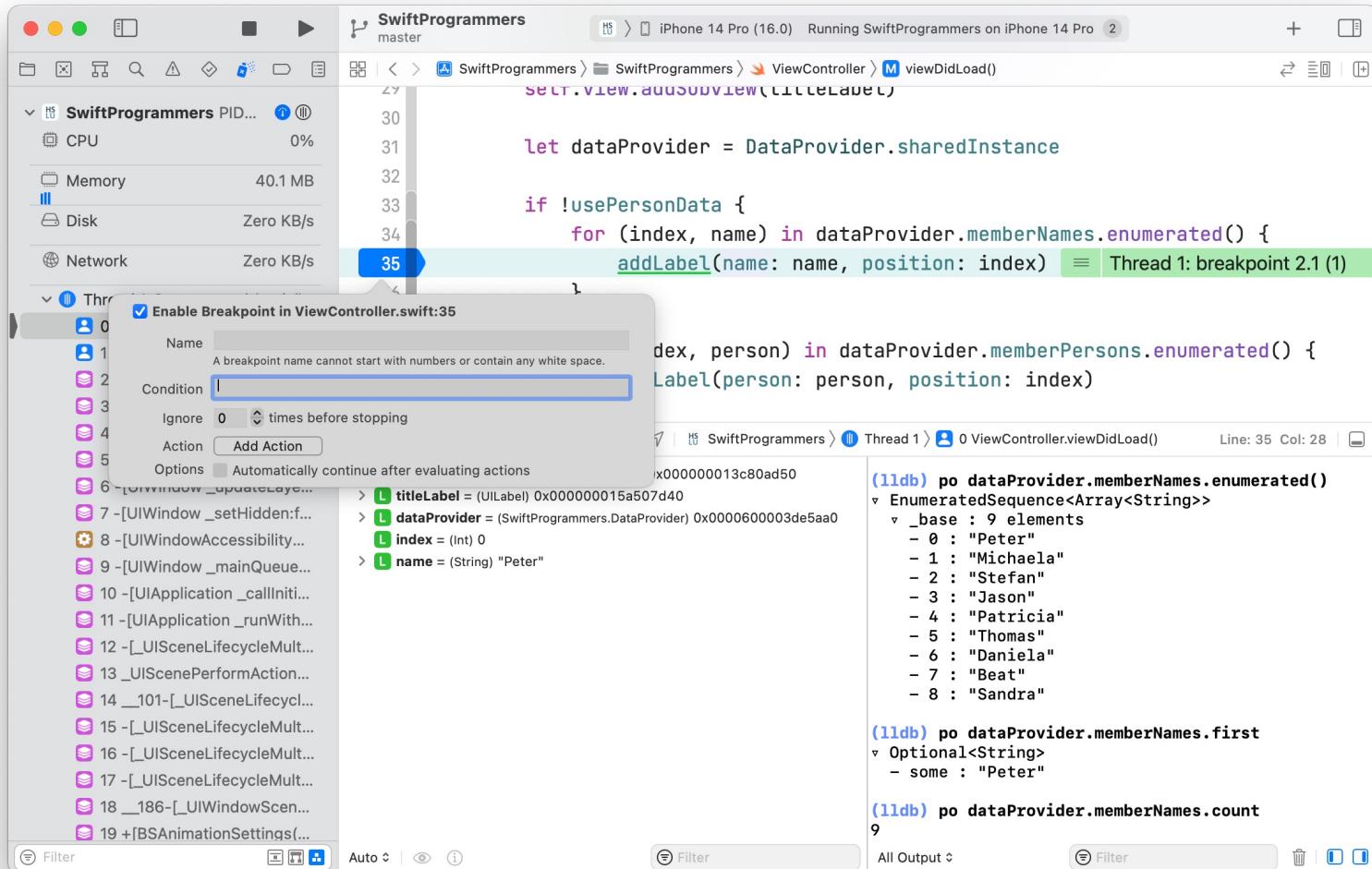
// UserStateManager+LoginDelegate.swift
extension MyImportantManager: LoginDelegate {
    func loginStateChanged(_ newState: Bool) {
        self.isLoggedIn = newState
    }
}
```

# 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





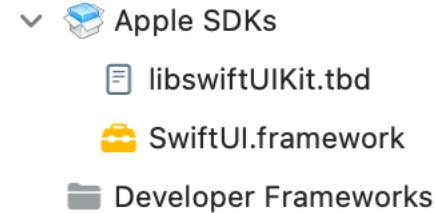
imgflip.com

# 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:

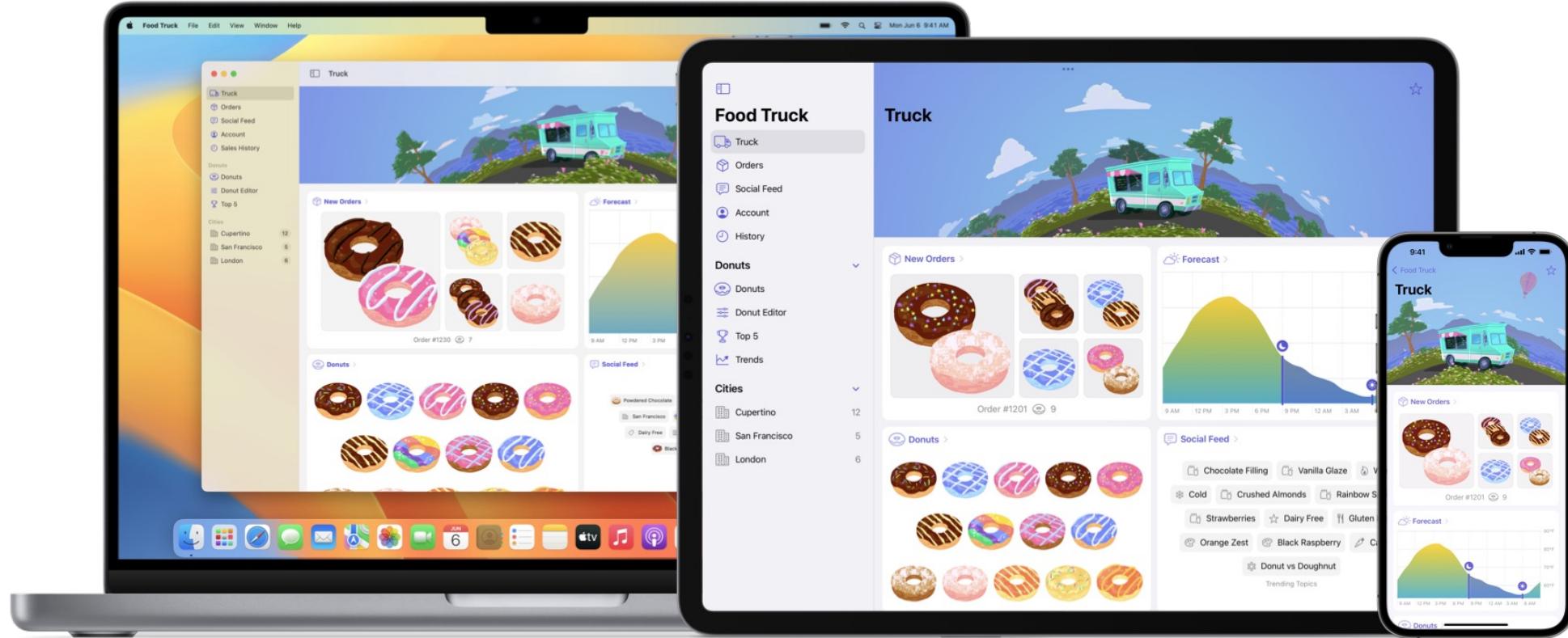


# Wieso SwiftUI lernen?

- 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

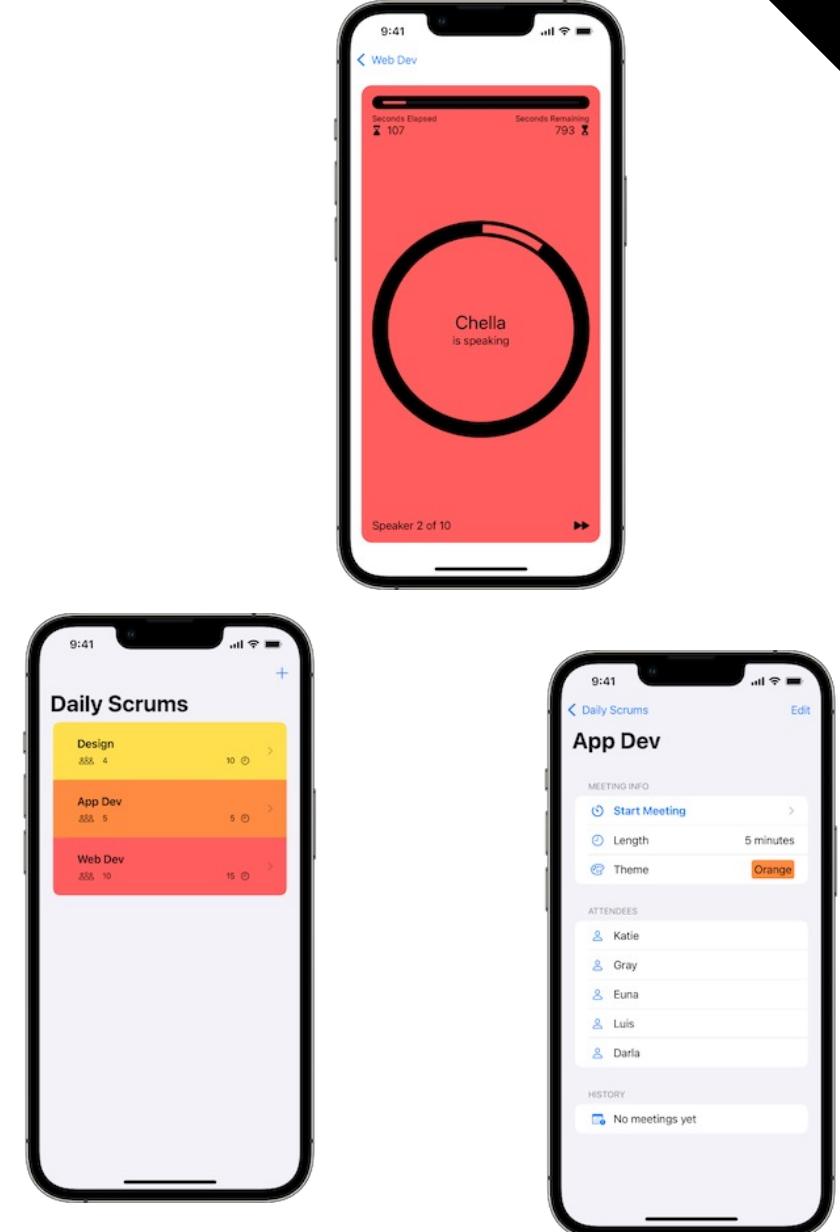


```
struct ContentView: View {  
    var body: some View {  
        Text("Hello, world!")  
            .padding()  
    }  
}
```



# 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

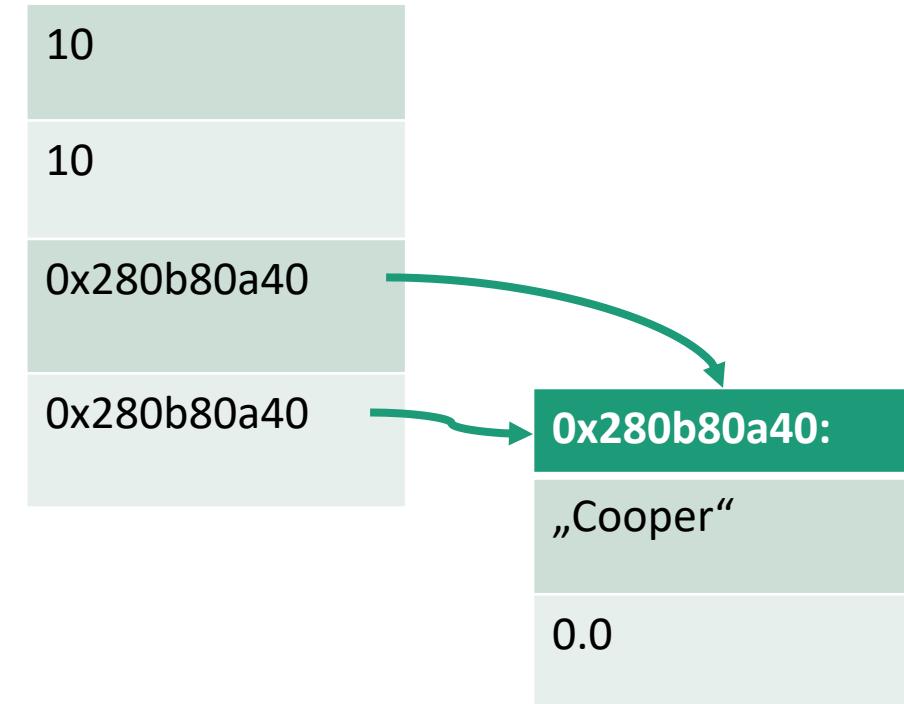
```
● ● ●  
struct Dog {  
    var name: String  
    var age: Double = 0.0  
    init(_ name: String) { self.name = name }  
}  
  
var foo = 42  
var bar = 10  
foo = bar  
var myDog = Dog("Cooper")  
var other = myDog
```

10
10
„Cooper“
0.0
„Cooper“
0.0

# struct vs. class (2/2)

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

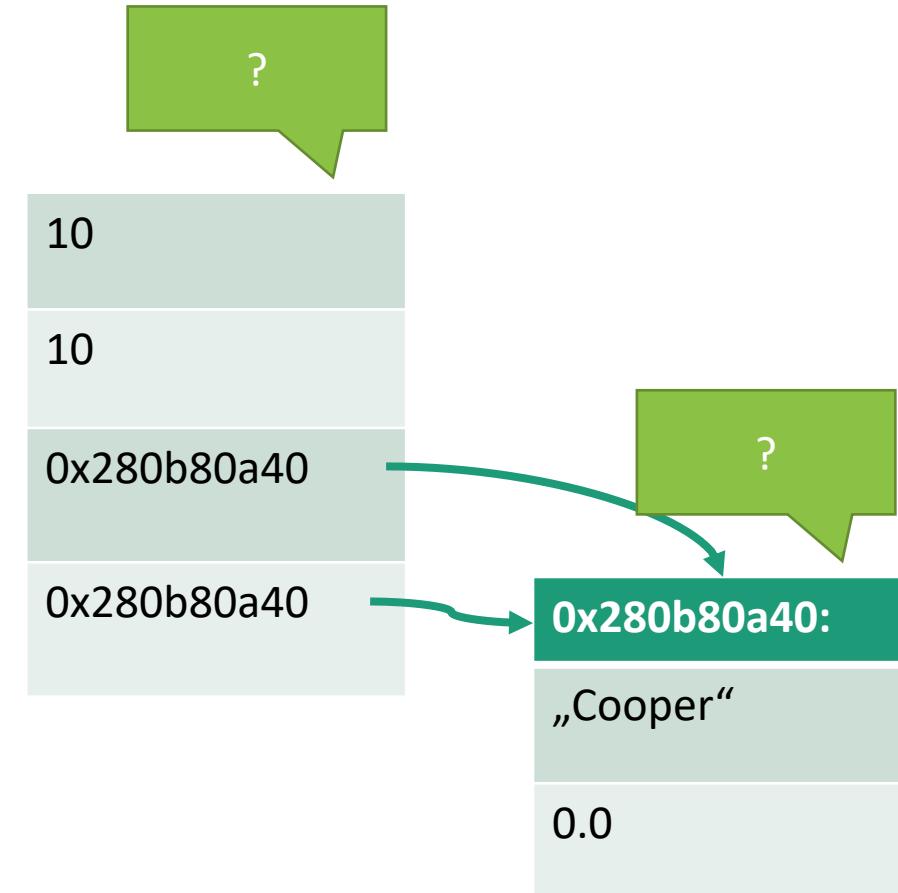
```
● ○ ●  
class Dog {  
    var name: String  
    var age: Double = 0.0  
    init(_ name: String) { self.name = name }  
  
    var foo = 42  
    var bar = 10  
    foo = bar  
    var myDog = Dog("Cooper")  
    var other = myDog
```



# struct vs. class (2/2)

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

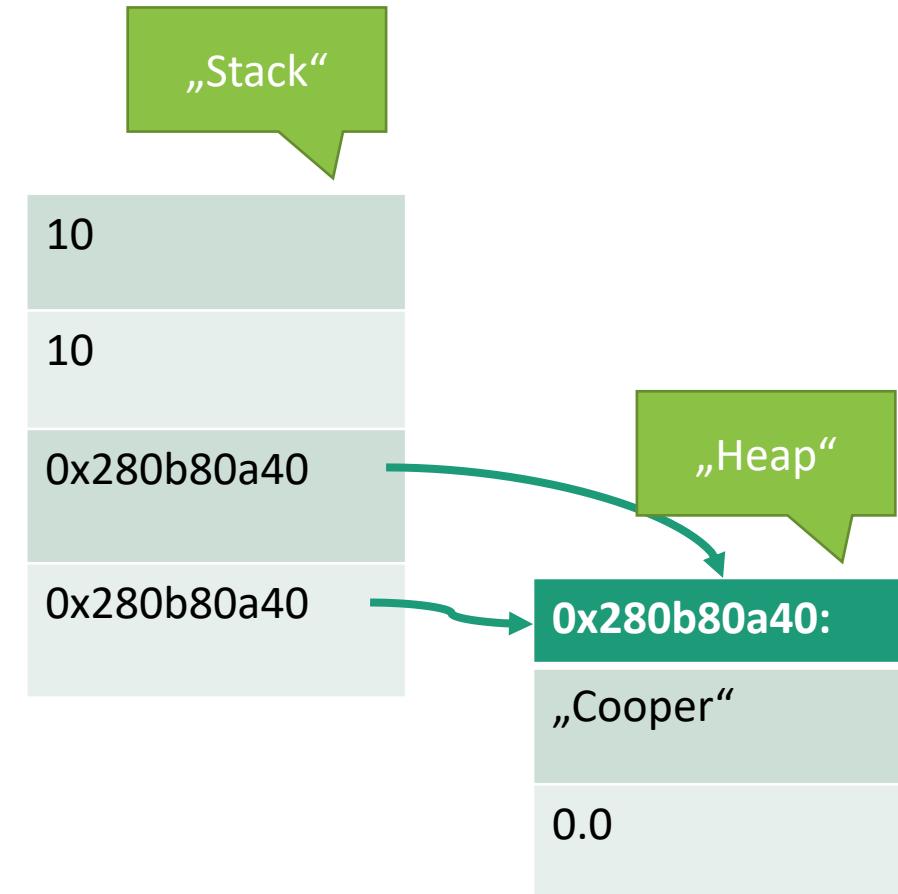
```
● ● ●  
class Dog {  
    var name: String  
    var age: Double = 0.0  
    init(_ name: String) { self.name = name }  
  
    var foo = 42  
    var bar = 10  
    foo = bar  
    var myDog = Dog("Cooper")  
    var other = myDog
```



# 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*?

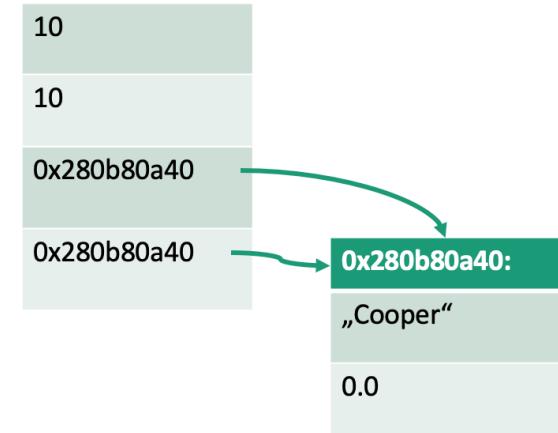
- Jede UIView speichert etwa 200 Properties und Abhängigkeiten (backgroundColor, frame und viele mehr), die nötig sind, um von der Grafikkarte effizient auf das Display projiziert zu werden.
- Das Erstellen neuer Objekte ist deshalb wesentlich teurer als das verändern einzelner Eigenschaften
- Erstellte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

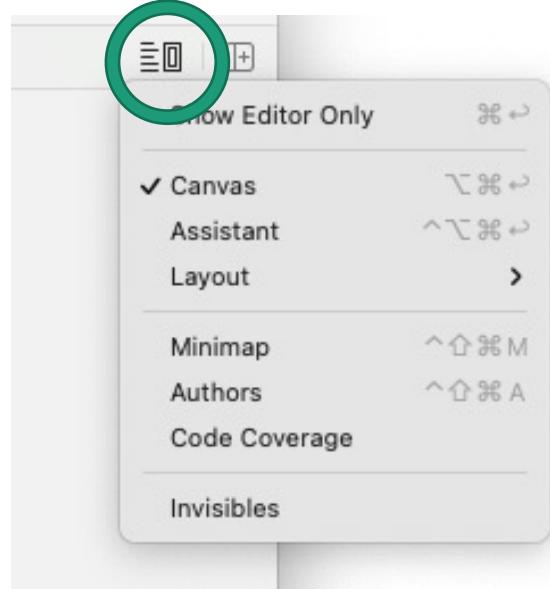


# Weitere Inhalte der Übung

- SF Symbols
- Asset Catalog
- Accessibility

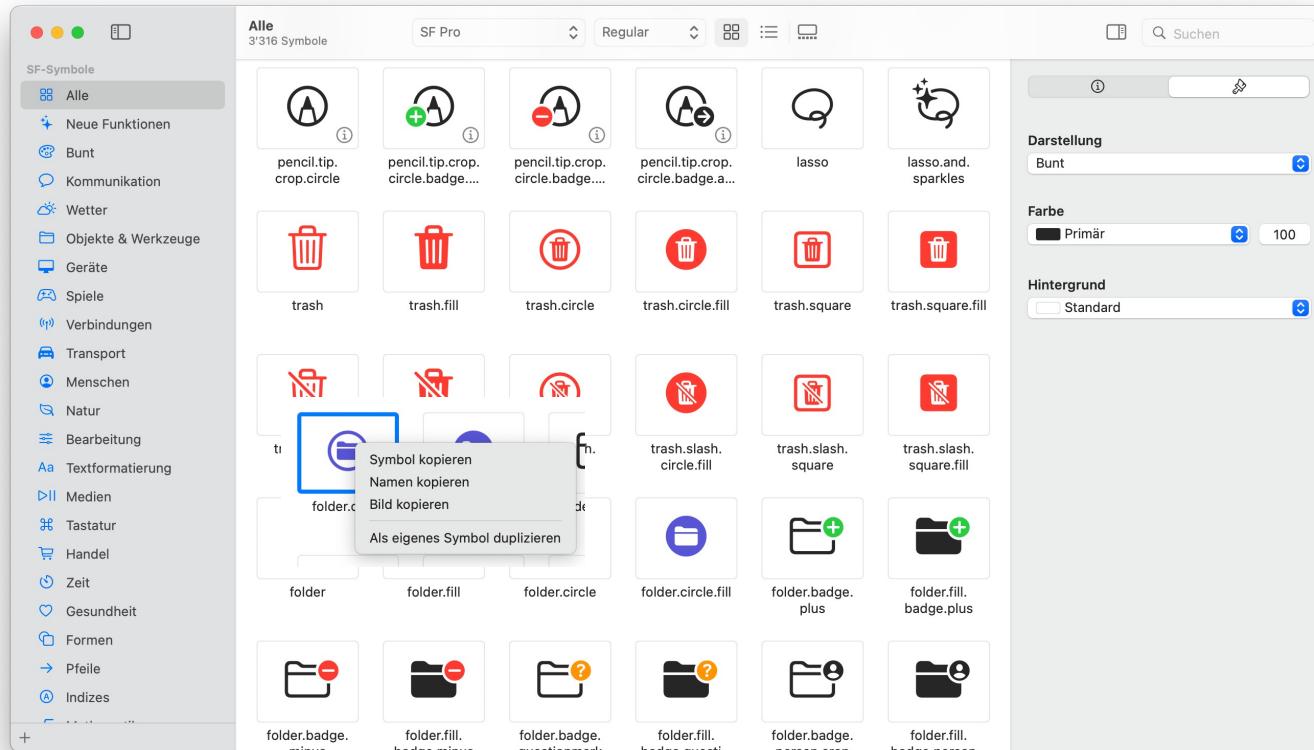
# Editor

- Live-Preview (Canvas)

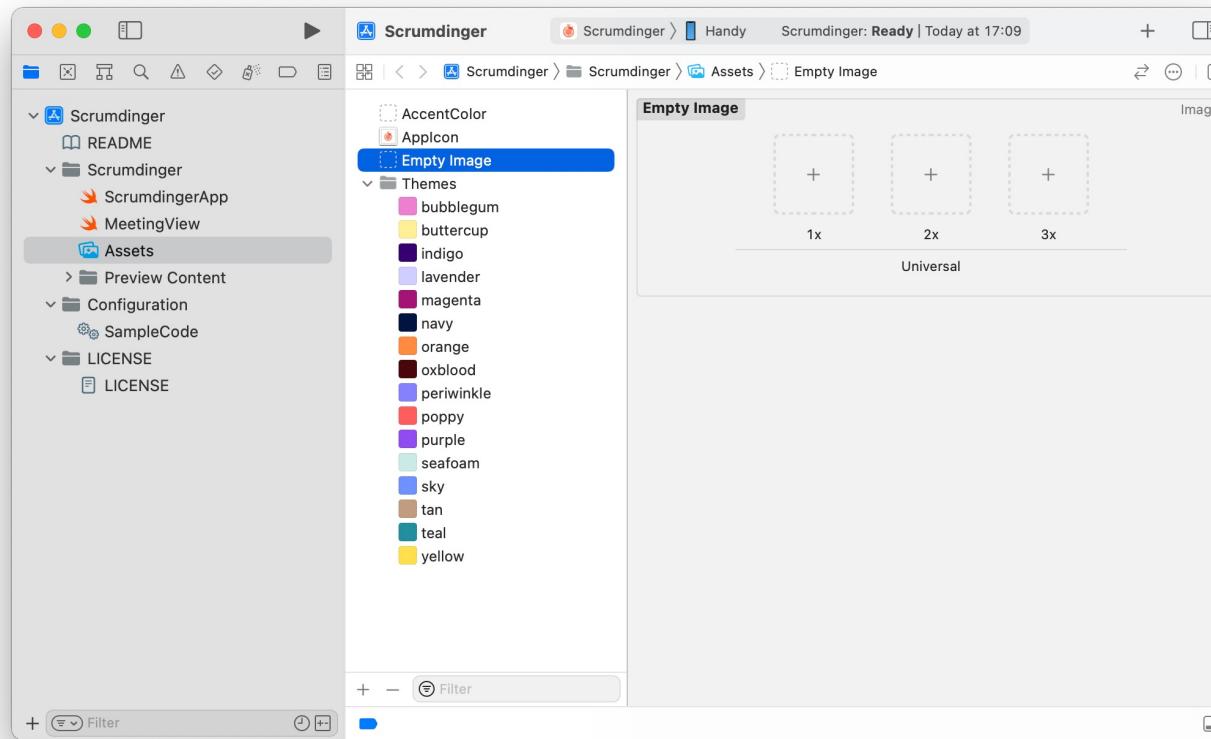
A screenshot of Xcode displaying the ContentView.swift file. The code defines a SwiftUI struct ContentView with a body containing a Text view that displays "Hello, world!". To the right of the editor, a large iPhone simulator window shows the actual application running on the device, displaying the text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

# SF Symbols

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



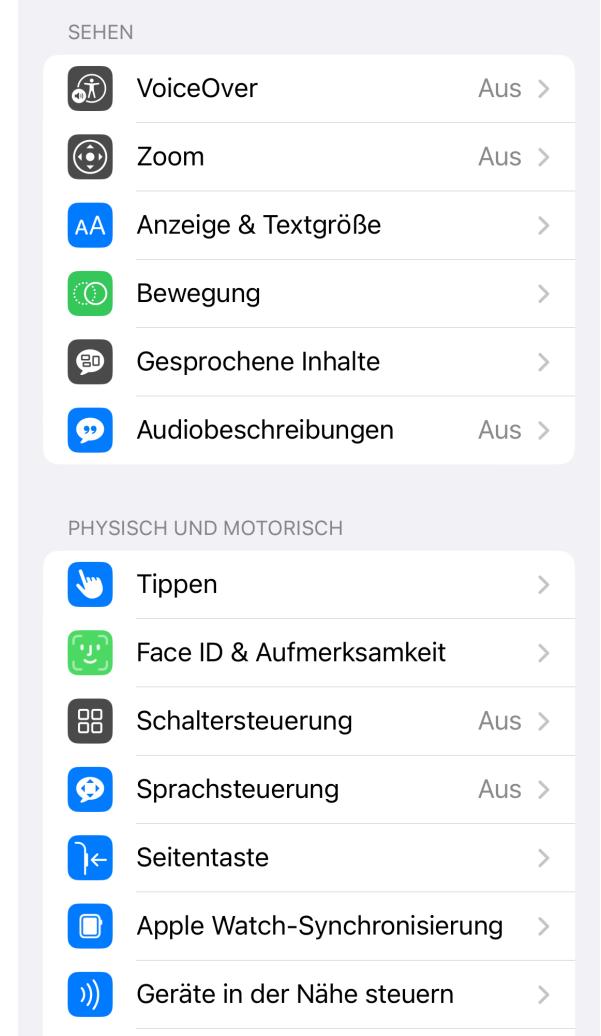
# Asset Catalog



- 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.

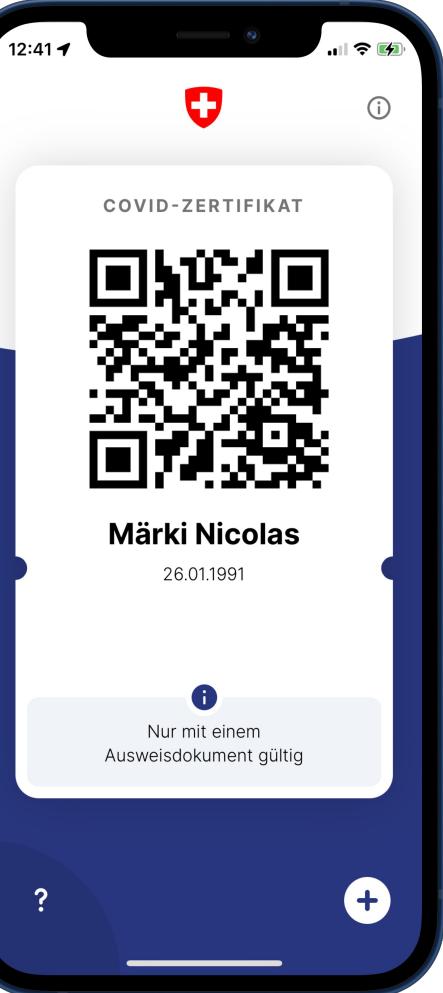
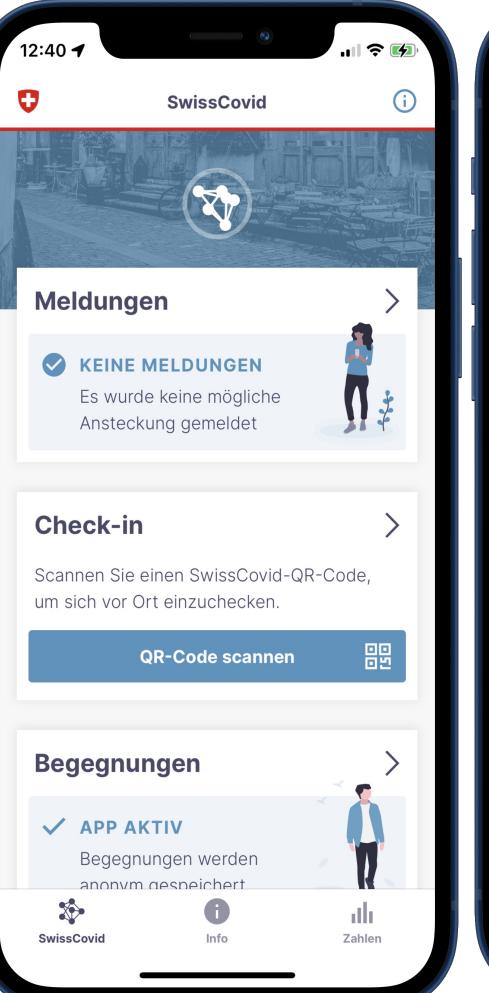
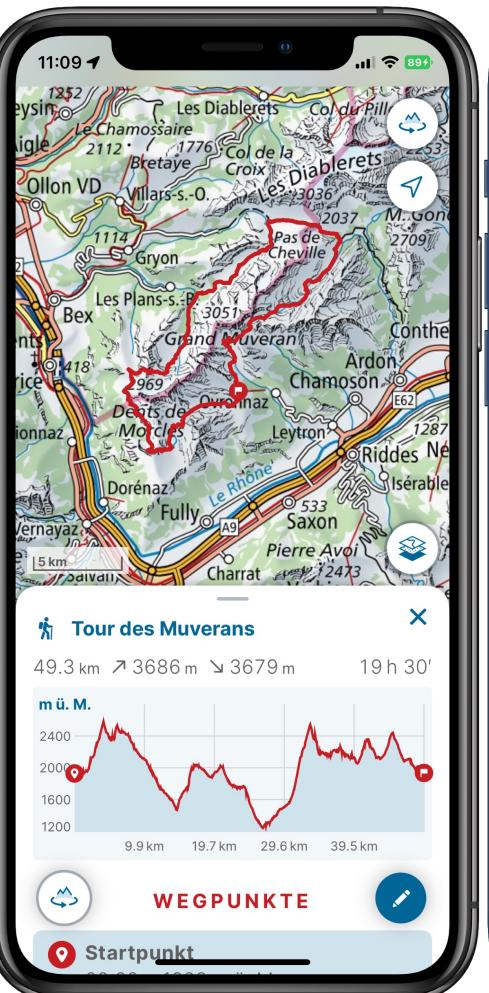
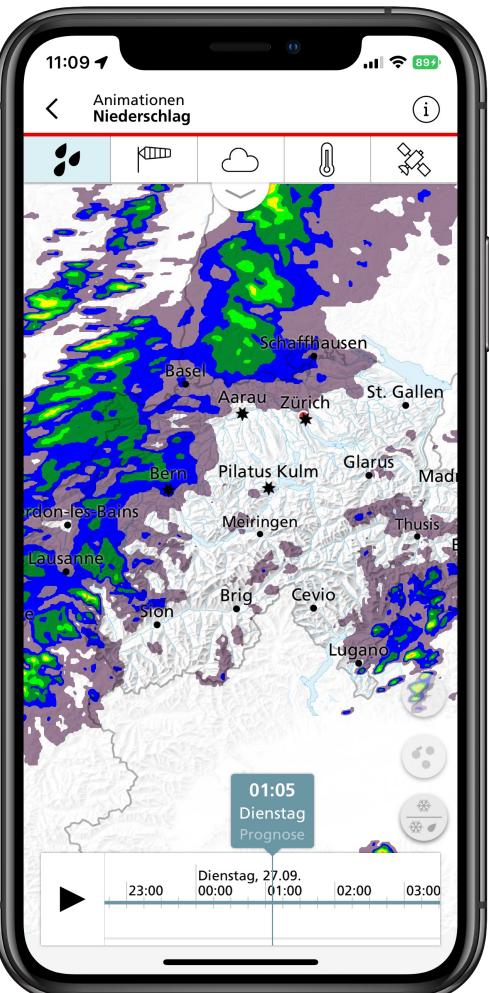
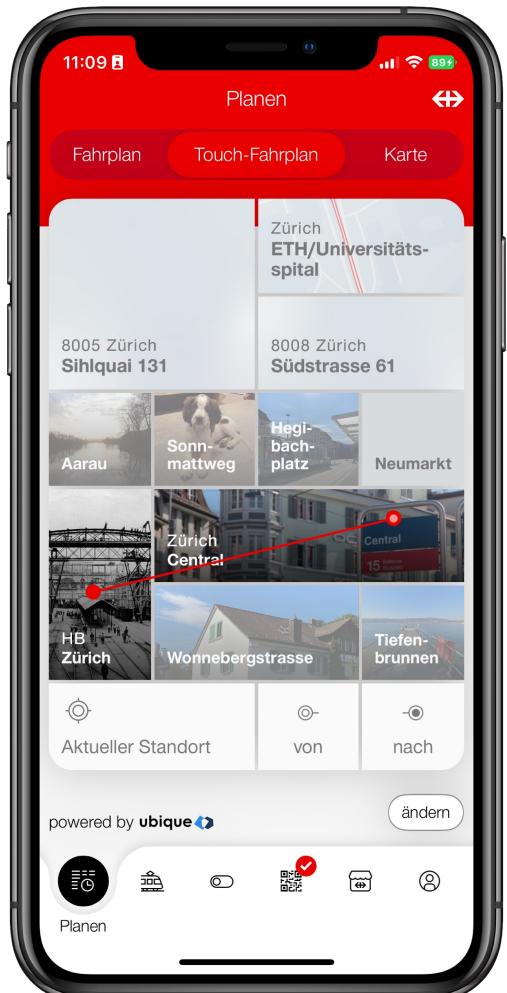
init<Data, ID, RowContent>(Data, rowContent: (Data.Element) -> RowContent)
computes its rows on demand from an underlying collection of
SelectionValue is Never and Content conforms to View.

>(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