

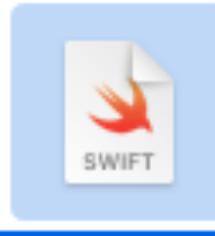
SwiftUI 2

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!

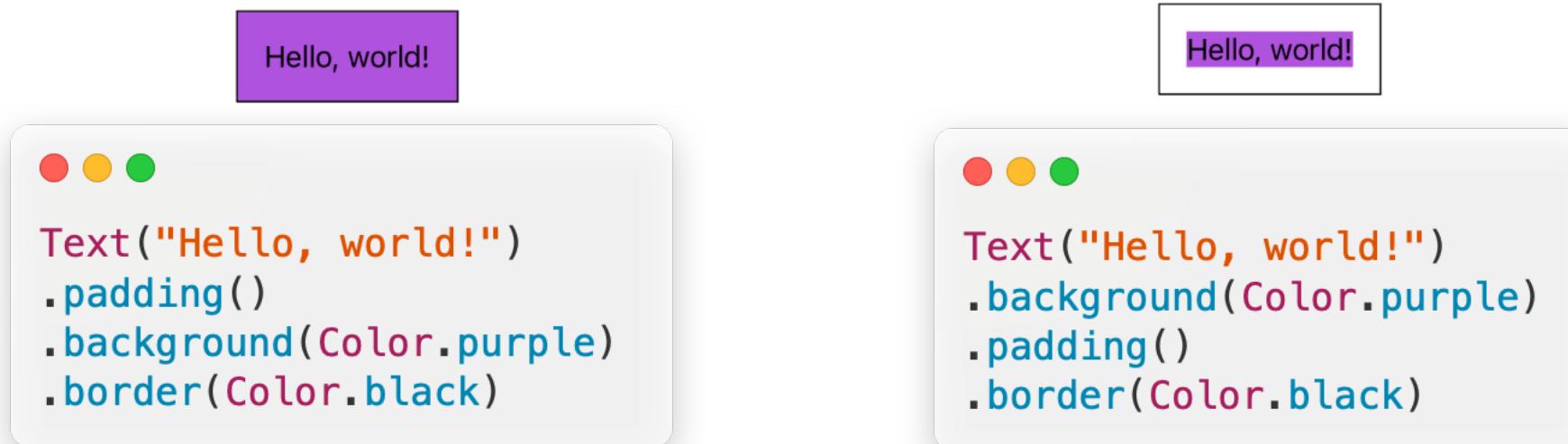
View Modifiers

- 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**

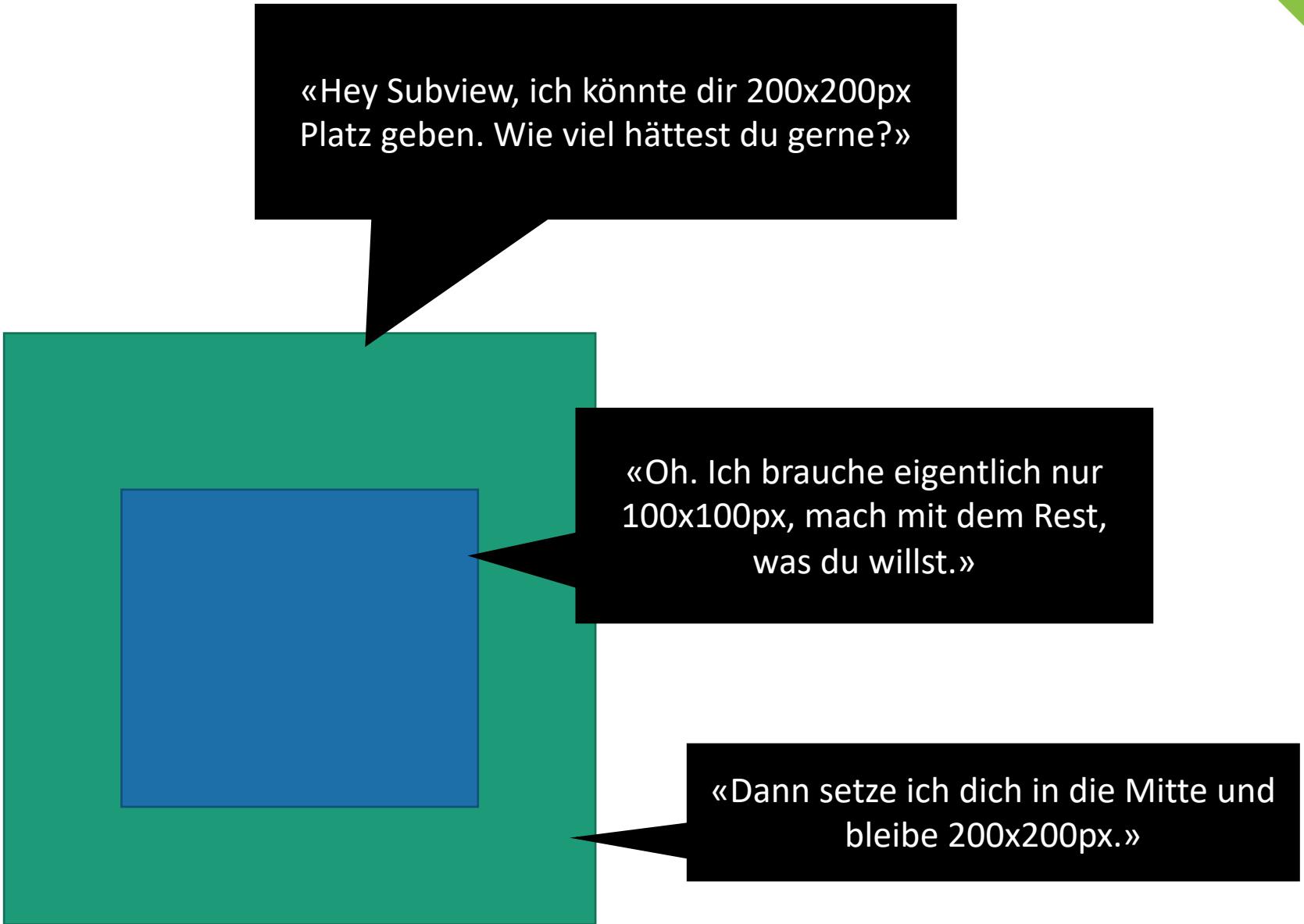
Hallo 

Layout

Programmieren für iOS



Layout-Dialog

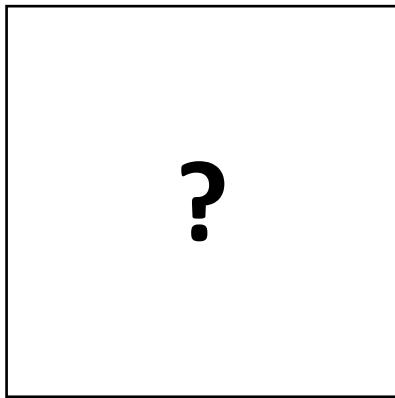


Layout Mechanismus

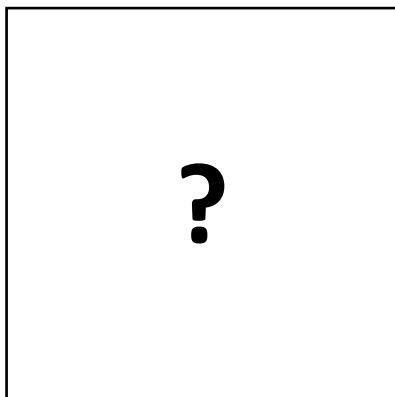
- Alle Layouts in SwiftUI funktionieren mit den gleichen drei Regeln:
 1. Eine View bietet einer Subview eine bestimmte Grösse an
 2. Die Subview kann diese Grösse annehmen oder **sich für eine andere Grösse entscheiden**
 3. Die Container-View positioniert die Subview
- Diese Schritte werden rekursiv auf alleSubviews 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

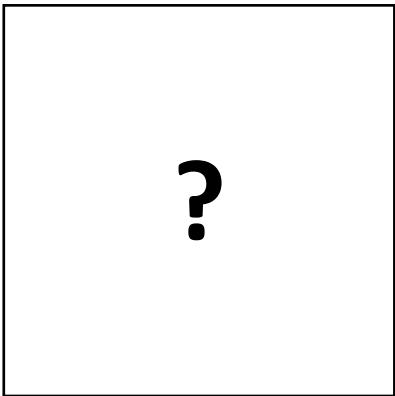


```
VStack(spacing: 0) {  
    Color.blue  
    Color.red  
}.frame(width: 200, height: 200)
```



```
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
```

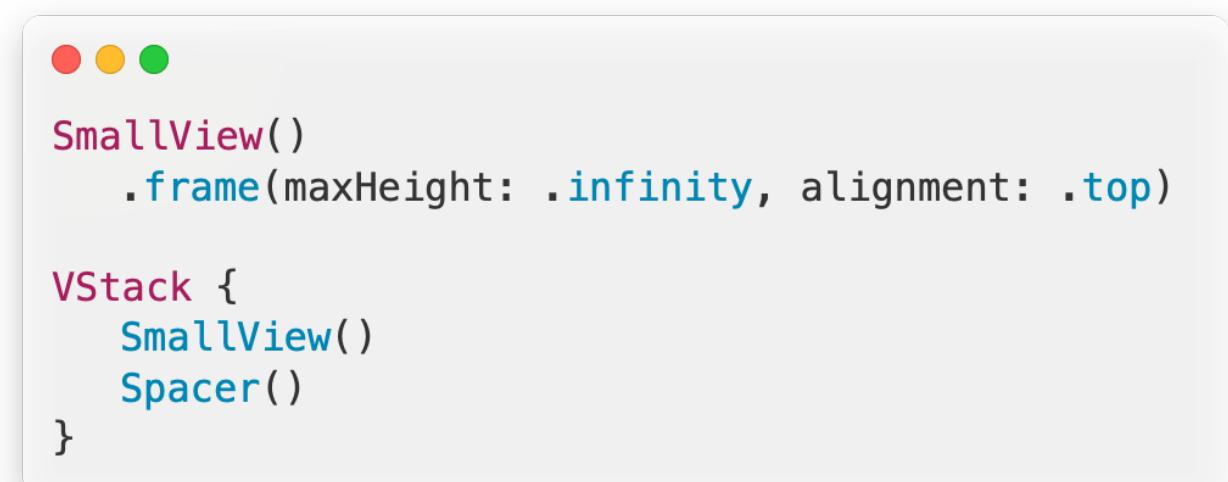
- Die ursprüngliche View **muss die neue Grösse nicht übernehmen**
- Falls die View die Grösse vom neuen Frame ignoriert, wird sie mit dem angegebenen *Alignment* positioniert



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)
```



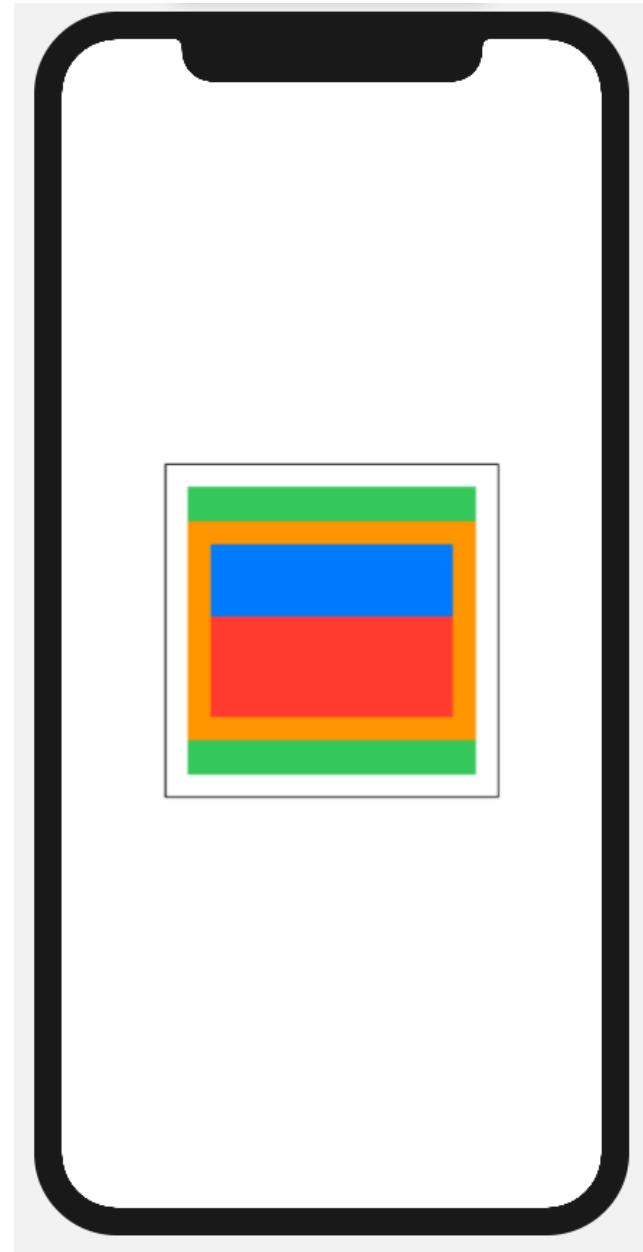
```
VStack(spacing: 0) {  
    Color.blue.frame(height: 350)  
    Color.red.frame(height: 100)  
}.frame(width: 200, height: 200).border(Color.black)
```



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

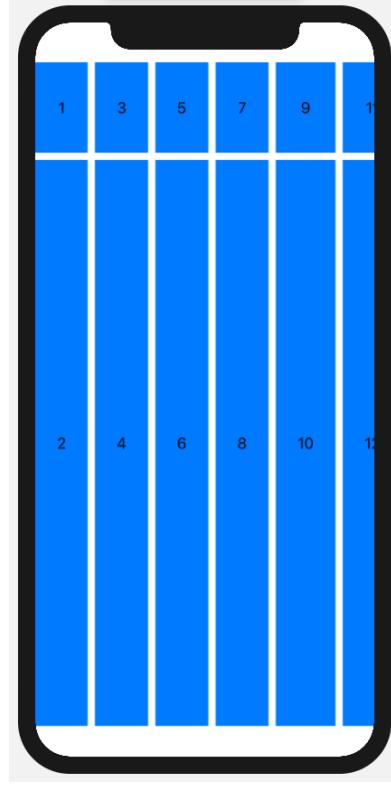
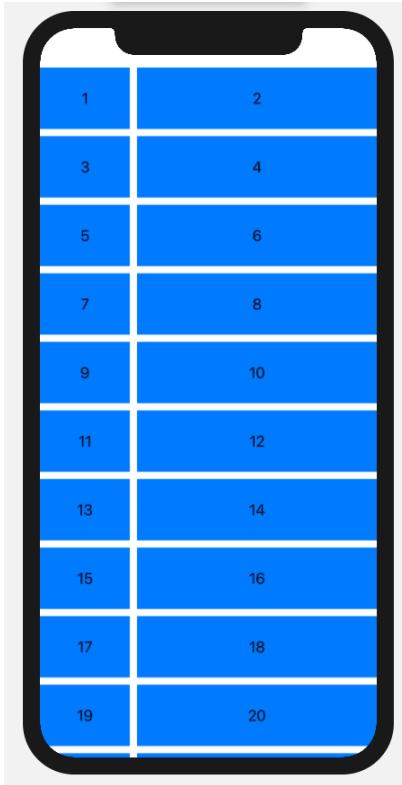
- Mit *LazyVGrid* und *LazyHGrid* können Raster-Layouts beschrieben
- «Lazy»: Es wird nur soviel Inhalt wie nötig geladen
 - *ForEach* wird nur teilweise ausgeführt
 - Andere Views wie *List* laden zwar immer den ganzen Inhalt, sind aber trotzdem sehr effizient
- Layout abhängig von den *GridItems*,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

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

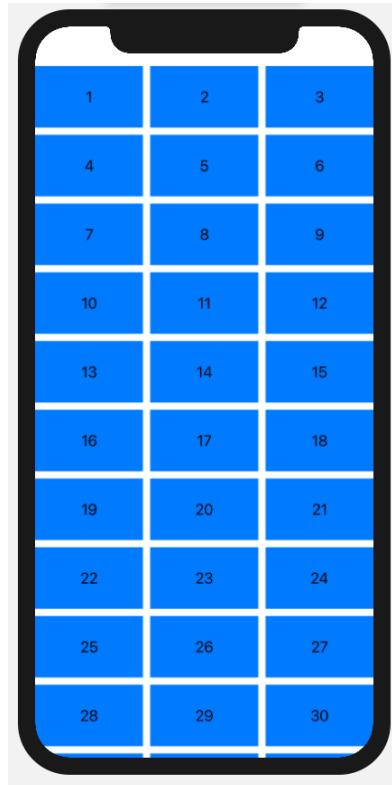
static let items: [GridItem] = [⋮]

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)
                }
            }
        }
    }
}
```

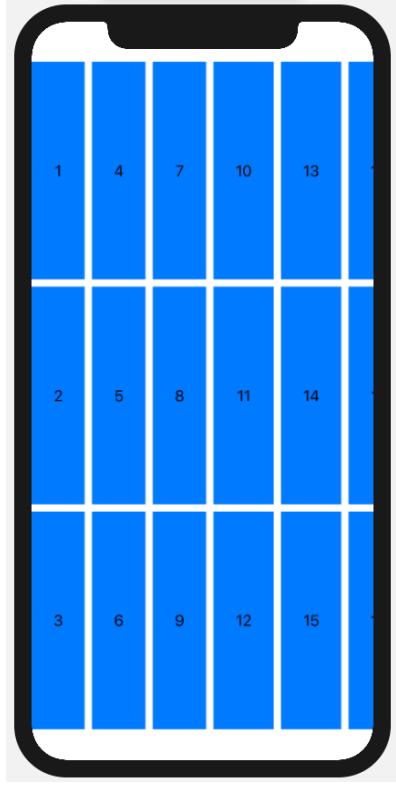
Grids, Teil 2



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



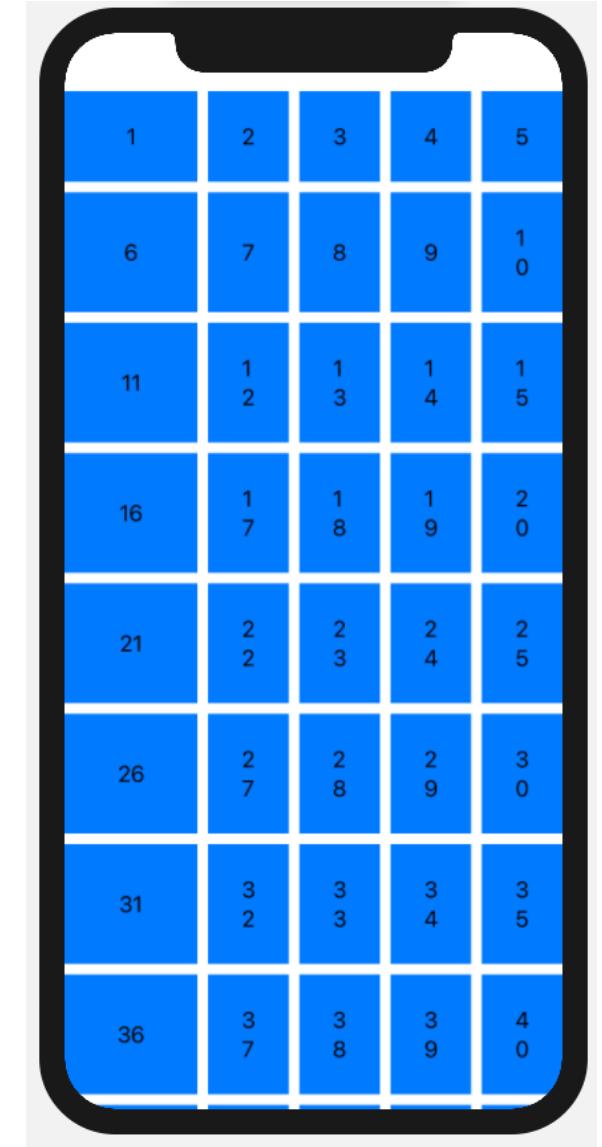
```
static let items: [GridItem] = [  
    GridItem(.flexible()),  
    GridItem(.flexible()),  
    GridItem(.flexible())  
]
```



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



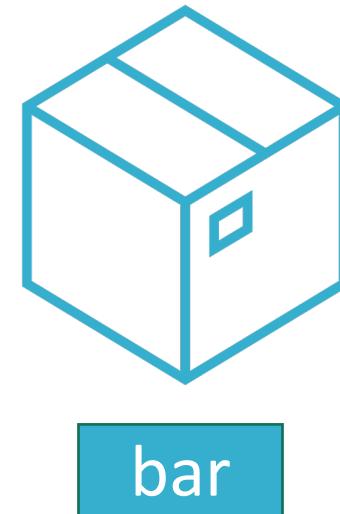
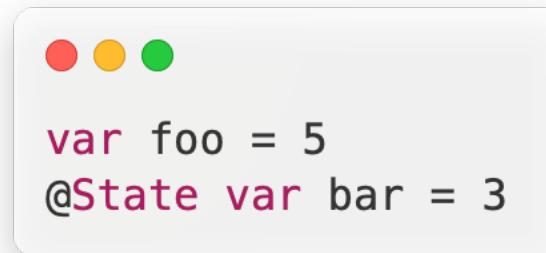
```
struct ContentView: View {  
    @State let text = "foo"  
    var body: some View {  
        Text(text)  
        .onAppear {  
            text = "bar"  
        }  
    }  
}
```

Property wrapper
can only be applied
to a 'var'

State

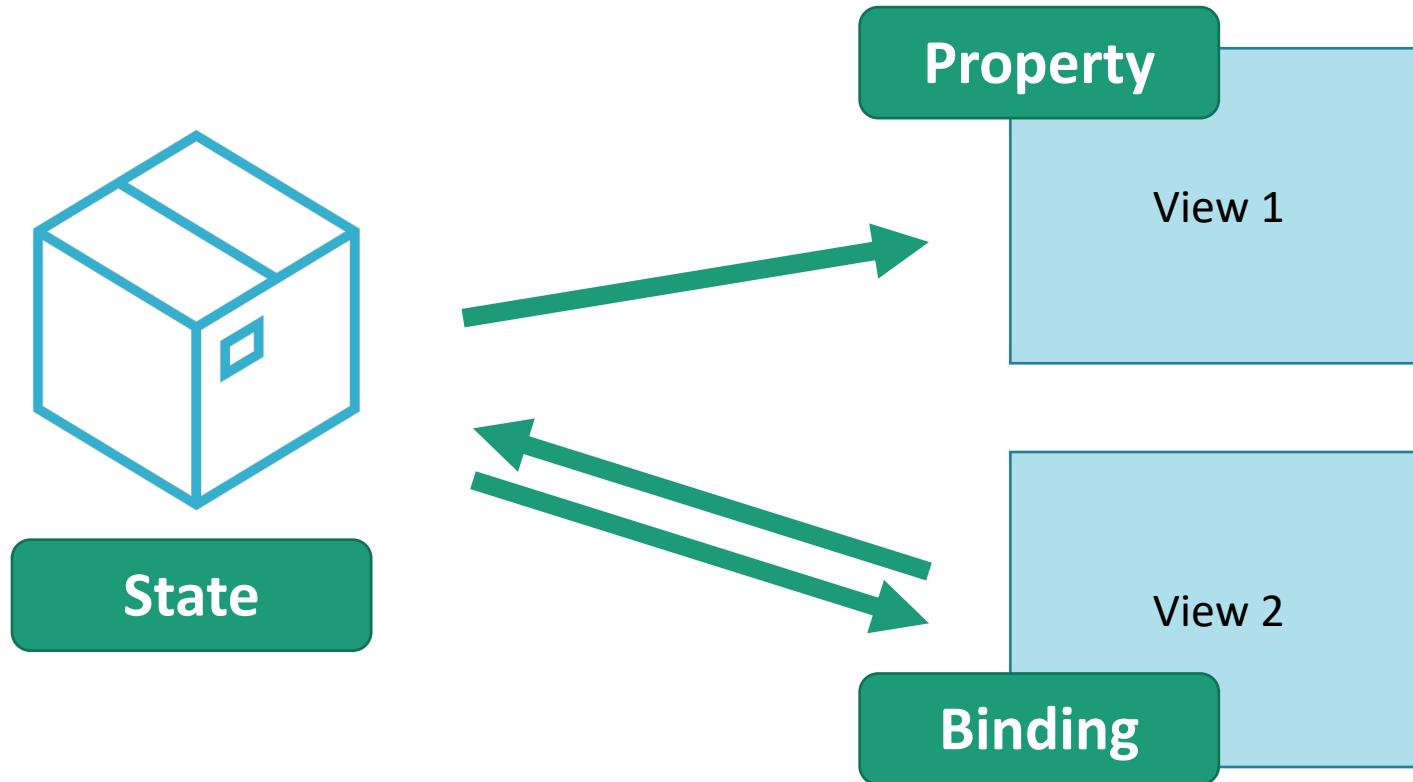
- Alle Variablen, die den Zustand einer View beschreiben, werden mit **@State** annotiert
- Wenn der State ändert, wird die View automatisch aktualisiert → eine wichtige Fehlerquelle fällt weg
- Auch wenn State-Properties normal verwendet werden können, kann es hilfreich sein, sich diese als Box mit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

5
foo



Binding

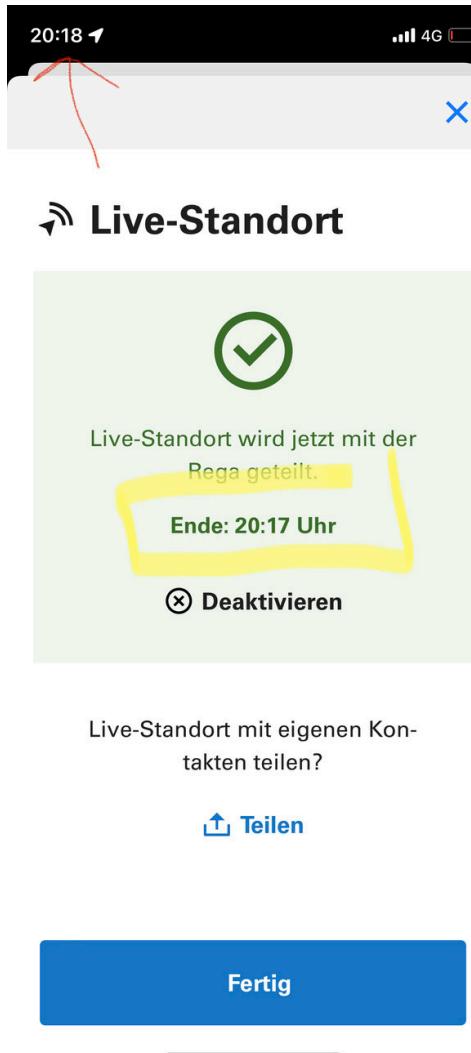
- Eine View kann den aktuellen State einer anderen View als Property übergeben oder mit einem **Binding** den Zugriff auf die Box teilen.



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

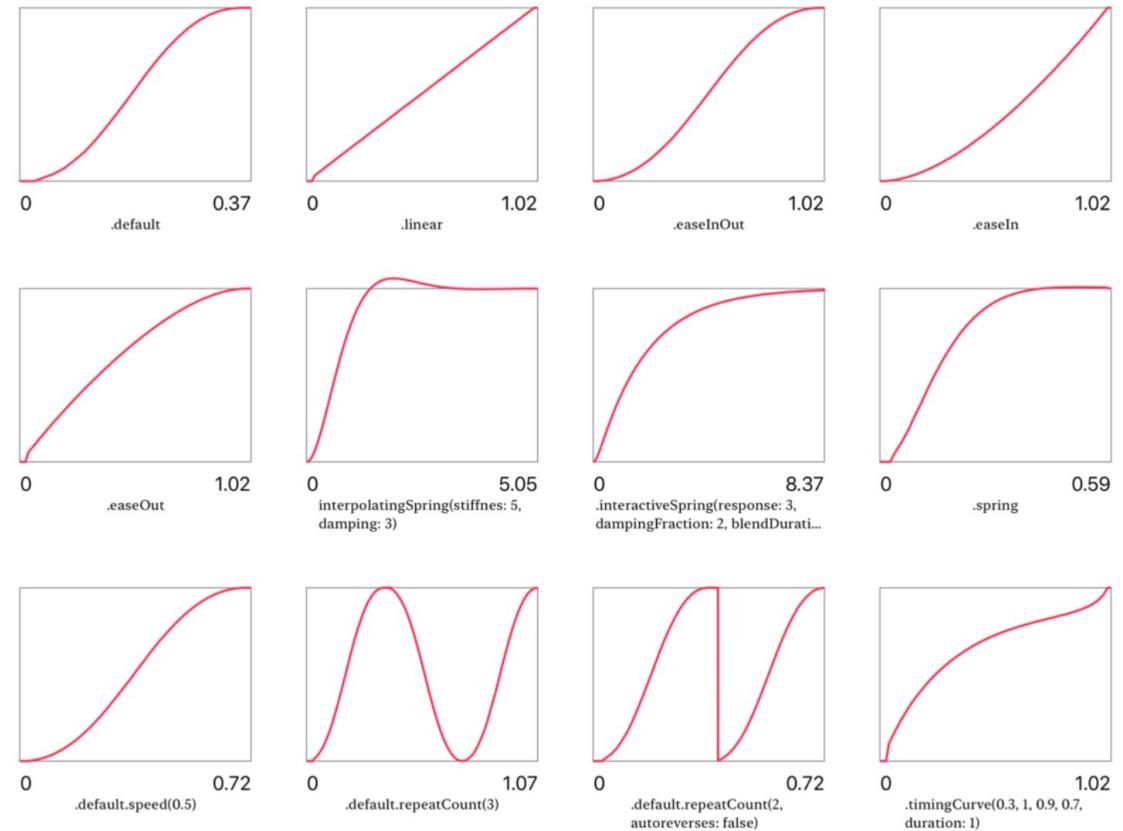
```
@State var scale = 1.0
var body: some View {
    Button("Tap me") {
        scale += 1
    }.scaleEffect(scale)
    .animation(.default, value: scale)
}
```

```
@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

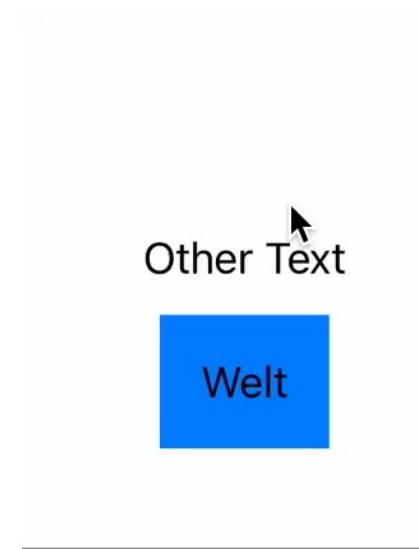
- Es gibt einen Modifier `.id()`, damit kann die Identity einer View explizit neu gesetzt werden
- Zwei unterschiedliche Views mit gleichem ID-Modifier sind aber immer noch zwei unterschiedliche Views
 - Beispiel recht: ConditionalContent-View mit zwei IDView

```
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