Fachhochschule
Nordwestschweiz

iOS Modul
Property Observers, @State, Animation

Property Observers

- Möglichkeit Variablen zu beobachten und bei einer Veränderung des Wertes zu reagieren

```
var isFaceUp: Bool {
        willSet {
                if newValue {
                        startUsingBonusTime()
                } else {
                        stopUsingBonusTime()
                }
        }
}
```

  - newValue enthält den Wert, der gesetzt wird.
  - Zur Benachrichtigung, nachdem ein Wert gesetzt wurde, wird didSet (oldValue) benutzt

@State

- Alle Views sind readonly
  - var wird nur verwendet, wenn der Wert während der Initialisierung gesetzt wird oder wenn er sich um eine computed Property (read-only) handelt.
  - Hierdurch wird das Statehandling vereinfacht und die Effizienz optimiert. Andernfalls würde die Gefahr bestehen, dass State und UI auseinander laufen.
  - Die meiste Zeit zeichnen Views den State des ViewModels. Gelegentlich ist es jedoch sinnvoll, wenn auch Views einen eigenen temporären State haben. Permanenter State wird immer im Model gehalten.
    - Ein Edit-Mode sammelt Informationen, die dann nur zusammen weiter verarbeitet werden
    - Temporäre Einblendung eines anderen Views (z.B. Alert)
    - Definieren einer Animation

@State

- Alle Variablen, die einen State halten, werden mit @State ausgezeichnet.

```
@State private var somethingTemporary: SomeType
```

- Immer wenn so ausgezeichnete Variablen verändert werden, wird das UI neu gezeichnet.
  - Analog zu @ObservedObject vom ViewModel.
  - Verändert sich das Model, was ein neues Zeichnen des Views zur Folge hat, werden die Werte der @State-Variablen gehalten
- Allgemein: Wenn immer Variablen im View nötig sind und auch mutiert werden, müssen diese mit @State ausgezeichnet werden

Animation

- Eine Animation hat immer einen zeitlichen Bezug und visualisiert Dinge, die sich vor kurzem geändert haben.
  - Hierdurch wird die User Experience verbessert, da sich UIs nicht schlagartig ändern.
  - Es wird die Aufmerksamkeit des Nutzers auf sich ändernde Dinge gelenkt.
- Diese Dinge lassen sich animieren, sofern die zum View gehörenden Container bereits angezeigt werden:
  - Das Erscheinen und Verschwinden von Views
  - Ändern von Werten von animierbaren ViewModifiern
  - Ändern von Werten bei der Erzeugung von Shapes

Animation

- Animationen lassen sich auf zwei Arten starten:
  - Implizit in dem man den ViewModifier .animation(Animation) verwendet.
  - Explizit, indem der ändernde Code innerhalb von .withAnimation(Animation){…} eingefügt wird.

Animation

- Implizite Animation
  - „Automatische Animation“: Alle ViewModifier-Argumente werden immer animiert.
  - Beispiel:

```
Text(“😛”)
        .opacity(scary ? 1 : 0)
        .rotationEffect(Angle.degrees(upsideDown ? 180 : 0))
        .animation(Animation.easeInOut)
```

    - Immer, wenn die Variablen scary oder upsideDown geändert werden, wird die Opacity und die Rotation animiert.
    - Ohne .animation() würde die View schlagartig geändert.

Animation

- Der Animation wird ein Animation-Struct übergeben. Dieses enthält Parameter zur Steuerung.
  - Dauer der Animation
  - Verzögerung zum Start
  - Anzahl der Wiederholungen bzw. unendlich
  - Animationskurve:
    - .*linear*: Geschindigkeit der Animation ist konstant
    - .*easeInOut*: Animation startet langsam, beschleunigt und wird wieder langsamer
    - .*spring*: Rückprall am Ende der Animation

Animation

- Implizite vs. Explizite Animationen
  - Implizite Animationen funktionieren nur auf einzelnen Views und nicht auf Containern.
    - D.h. in Views, die unabhängig von anderen Views sind
    - Daher lassen sich mit impliziten Animationen keine Animationen über verschiedene Views hinweg synchronisieren
  - Explizite Animationen sind vor allem für Animationen sinnvoll, die sich über mehrere Views erstrecken
    - Hierbei reagieren sie häufig auf eine Aktion des Benutzers oder eine Modelländerung
  - Explizite Animationen überschreiben Implizite Animationen nicht.

Animation

- Transitionen beschreiben, wie das Erscheinen und Verschwinden von Views im Container abläuft.
- Transitionen sind ein Paar von ViewModifiern. Hier wird beschrieben, wie der View vor und nach der Animation gestaltet sein soll.
- Asymmetrische Transitionen bestehen aus zwei Paaren von ViewModifiern. Jeweils für das Erscheinen und Verschwinden.
  - Z.B. eine View wird beim Erscheinen eingeblendet und fliegt beim Verschwinden über den Bildschirm.
- Transitionen beschreiben nur, wie die ViewModifier angepasst werden. Durch sie wird die Animation aber nicht ausgeführt.
- Transitionen funktionieren ausschliesslich mit Impliziten Animationen.

Animation

```
ZStack {
        if isFaceUp {
            RoundedRectangle() // default-transition = .opacity
            Text("Emoji").transition(.scale)
        }
        else {
            RoundedRectangle(cornerRadius: 10).transition(.identity)
        }
}
```

- isFaceUp == true: Rückseite verschwindet sofort, Text wächst aus dem Nichts, Vorderes RR blendet ein
- is FaceUp == false: Rückseite erschein sofort, Text schrumpft ins Nichts, Vorderes RR blendet aus

Animation

- Transitionen sind im Struct AnyTransition definiert
  - .opacity: Ein und Ausblenden von Views
  - .scale: Nutzt den .frame modifier um den View zu vergrössern / verkleinern
  - .offset(CGSize): Der View wird verschoben
  - .modifier(active: identity:): Eigene ViewModifier können so verwendet werden
- Transitionen lassen sich durch die .animation-Property an eigene Anforderungen anpassen

```
.transition(.opacity.animation(.linear(duration: 20)))
```

Animation

- Transitionen funktionieren nur mit Views, deren Container bereits angezeigt werden.
  - Soll ein View eingeblendet werden muss dazu eingegriffen werden, nachdem der Container sichtbar wurde.
  - Hierzu gibt es die .onAppear{}-Funktion. Sie wird beim Erscheinen eines Views aufgerufen. Für das Verschwinden gibt es die .onDisappear{}-Funktion.
    - Diese Funktionen werden auf dem Container aufgerufen, der die zu animierende View enthält.

Animation

- Shape- und ViewModifier-Animationen
  - Der Benutzer definiert in Animationen den Ausgangs- und Endpunkt der Animation.
  - Das Animationssystem teilt die Animation in einzelne Schritte über die Dauer der Animation auf und verändert die ViewModifier dabei entsprechend. Die Anpassung verursacht das Neuzeichnen des animierten Objektes.
  - Shapes und ViewModifiers, die animiert werden sollen, müssen das Animatable-Protokoll implementieren

```
var animatableData: Type
```

    - Type ist ein Generic, der aber das VectorAritmetic-Protokoll implementieren muss, damit die Schrittweisen Anpassungen durchgeführt werden können. Häufig ist Type ein Float, Double oder CGFloat.