Fachhochschule
Nordwestschweiz

iOS Modul
Reactive UI Protocols Layout

Protocol
Enum

Protocol

- Wie eine reduzierte Klasse / Struct, ähnlich dem Interface in Java / C#
- Protocols bestehen aus Properties (var) und Functions (func), enthalten aber keine Implementierung oder Speicher.
- Ein Protocol wird ähnlich einer Klasse / Struct deklariert:

```
protocol Moveable{
      func move(by: Int)
      var hasMoved: Bool { get }
      var distanceFromStart: Int { get set }
}
```

Protocol

```
protocol Moveable{
      func move(by: Int)
      var hasMoved: Bool { get }
      var distanceFromStart: Int { get set }
}
```

Jeder andere Typ kann Movable implementieren:

```
struct PortableThing: Moveable {
      // muss move(by:), hasMoved und distanceFromStart
            implementieren
}
```

Protocol Inheritance

Protokolle können auch andere Protokolle erben:

```
protocol Moveable{
      func move(by: Int)
      var hasMoved: Bool { get }
      var distanceFromStart: Int { get set }
}
```

```
protocol Vehicle: Moveable {
      var passengerCount: Int { get set }
}
```

```
class Car: Vehicle{
      // muss move(by:), hasMoved, distanceFromStart und
         passengerCount implementieren
}
```

Implementieren mehrerer Protokolle

Structs und Klassen können gleichzeitig mehrere Protokolle implementieren:

```
class Car: Vehicle, Impoundable, Leasable {
      // muss move(by:), hasMoved, distanceFromStart
         und passengerCount von Vehicle implementieren
      // zusätzlich müssen auch alle vars / funcs von
         Impoundable und Leasable implementiert werden
}
```

Implementieren mehrerer Protokolle

Protokolle dürfen auch als Argument in Funktionen oder als Variable von Properties verwendet werden:

```
var m: Moveable
var car: Car = new Car()
var portable: PortableThing = PortableThing()
m = car // valider Code
m = portable // valider Code
```

- Valide, da Car und PortableThing jeweils Movable implementieren

Nicht gültig wäre jedoch die Zuweisung:

```
portable = car
```

- Grund: portable ist vom Typ PortableThing (nicht Moveable) und Car ist kein PortableThing (Am besten in Xcode nachvollziehen)

Protocol extension

Protokol Extensions erweitern Swift um ein wichtiges Feature und sind ein wichtiger Aspekt der funktionalen Programmierung in Swift. Das View-Protokoll von SwiftUI ist ein sehr prominentes Beispiel.

```
struct Tesla: Vehicle{
      //Implementierung aller in Vehicle definierten vars / funcs
}
```

```
extension Vehicle{
      func registerLicencePlate(){ //Implementierung }
}
```

- Hierdurch kann auf jedem Tesla-Objekt (und jedem Vehicle) `registerLicencePlate()` aufgerufen werden, ohne dass die Funktion „selbst“ implementiert werden muss.

Protocol extension

Protocol Extensions lassen sich zum Beispiel dazu verwenden um eine Default-Implementation umzusetzen. Die Pflicht die func / var dann noch selbst zu implementieren wird dadurch optional.

```
protocol Moveable {
      func move(by: Int)
      var hasMoved: Bool { get }
      var distanceFromStart: Int { get set }
}
extension Moveable {
      var hasMoved: Bool { return distanceFromStart > 0 }
}
```

```
struct ChessPiece: Movable{
      // move und distanceFromStart muss implementiert werden
      // hasMoved darf überschrieben werden
}
```

Extensions für Klassen und Structs

Durch Extensions lassen sich Typen wie Structs und Klassen erweitern. Das ist insbesondere für fremden Code interessant, wie der von Apple.

```
struct Boat {
      ...
}

extension Boat {
      func sailAroundTheWorld() {
            /* implementation */
      }
}
```

Motivation für Protocols

- Typen können dadurch Ausdrücken, welche Fähigkeiten sie haben. Dabei spielt es keine Rolle, welcher konkrete Typ genau verwendet wird.
- Auch lassen sich von der anderen Seite damit Anforderungen an Typen formulieren.
- Dies ist ein Kernelement der Funktionalen Programmierung: Es wird auf die Funktionalität fokussiert, die Details der Implementierung aber versteckt.

Generics und Protocols in Kombination

```
protocol Greatness {
      func isGreaterThan(other: Self) -> Bool
}
```

`Self` entspricht dem Objekt, das das Protokol implementiert z.B. Car.

```
extension Array where Element: Greatness{
      var greatest: Element {
            //for-loop durch alle Elemente und nutzen der
              Funktion isGreaterThan(other: Self) zur Ermittlung
              des grössten Elementes
      }
}
```

Enumeration

- Ein Enum ist ein Typ, der einen diskreten Wert enthält.
- Enums gibt es in vielen Sprachen, aber Swift-Enums können auch funcs / computed vars enthalten.
- Ausserdem kann jeder diskreter Wert zugeordnete Daten enthalten.

```
enum VehicleCondition{
      case accidentFree
      case damaged(Double)
}
```

- Details später

Layout

Wie wird einem View der Platz auf dem Bildschirm zugeteilt?

1. Ein Container-View bieten den beinhaltenden Views Fläche an, diese einzunehmen.
2. Views berechnen darauf die Fläche, die sie tatsächlich benötigen.
3. Der Container-View positioniert drauf die Views innerhalb seiner Fläche.

Container-Views

- HStacks und VStacks teilen den Platz, der Ihnen zur Verfügung steht unter Ihren Sub-Views auf.
- Mit ForEach teilt man die Kinder in einem Container auf.
- Modifiers (z.B. padding()) enthalten den View, den sie anpassen.

HStack und VStack

- Stacks teilen die ihnen zur Verfügung stehende Fläche auf, indem sie Ihren Sub-Views Fläche anbieten. Dabei wird den Elementen zuerst ein Angebot gemacht, die „am wenigsten flexibel“ sind.
  - Unflexible Elemente: Image (feste Bildgrösse), Text (Fläche soll dem benötigten Inhalt entsprechen)
  - Flexible Elemente: RoundedRectangle (passt sich der angebotenen Fläche an)
- Nachdem ein View die benötigte Fläche kommuniziert hat, wird diese von der zur Verfügung stehenden Fläche des Stacks abgezogen und zum nächsten Element übergegangen.
- Sind die Flächen aller Sub-Views bestimmt, kommuniziert der Stack die benötigte Fläche an seinen Parent-View.

HStack und VStack

- Es gibt einiges Views, die oft in Stacks verwendet werden
  - Spacer(minLength: CGFloat)
    - Nimmt die ganze angebotene Fläche
    - Ist selbst nicht sichtbar
    - minLength hat als Standardwert die für die Plattform übliche Länge (iOS, iPadOS, TVOS, …)
  - Divider()
    - Zeichnet eine Linie als Trennelement in der Ausrichtung des jeweiligen Stacks
    - Verwendet den minimal nötigen Platz um die Breite bzw. Höhe der Linie zu zeichnen

HStack und VStack

Das Verhalten der Flexibilität kann überschrieben und dadurch das Layout gesteuert werden:

```
HStack {
      Text(“Important”).layoutPriority(100) // Double
      Image(systemName: “arrow.up”) // default layout priority: 0
      Text(“Unimportant”)
}
```

- Zuerst bekommt “Important“ Fläche zugewiesen, die es benötigt
- Als nächstes Image, da es weniger Flexibilität hat als Text
- Zuletzt muss „Unimportant“ mit der übrigen Fläche auskommen. Genügt die Fläche nicht, wird der Text abgeschnitten und zum Beispiel durch drei Punkte ersetzt.

HStack und VStack

- Gelegentlich ist es notwendig die Ausrichtung (Aligment) des Stacks zu bestimmen.
- In einem VStack also die vertikale Ausrichtung:

```
VStack(alignment: .leading){}
```

  - Dabei wird `.leading` oder `.trailing` erwendet, damit auch in Sprachen wie Arabisch und Hebräisch die Ausrichtung korrekt ist.
- In einem HStack lässt sich zum Beispiel die Textgrundlinie anpassen:

```
HStack(alignment: .firstTextBaseline){}
```

- Es lassen sich auch eigene Ausrichtungen definieren.

Modifier

- View Modifier wie `.padding()` geben selbst wieder einen View zurück. Dieser enthält den View, den sie modifizieren.
- Viele Modifier reichen die zur Verfügung stehende Fläche im Container an die modifizierte View weiter, aber einige sind auch direkt am Layout-Prozess beteiligt:
  - `.padding(10)` gibt zum Beispiel die zugewiesene Fläche an die View weiter, reduziert aber an jeder Seite 10 Point, sodass sie grösser ist, als die View die sie beinhaltet.
  - `.aspectRatio(…)` ist ein anderes Beispiel, dass ja nach `(.fill)` oder `(.fit)` die beinhaltete View anpasst.
- Views ist es auch erlaubt nach aussen mehr Fläche zu wählen, als Ihr vom Container angeboten wurde.

Beispiel

```
HStack { // default alignmen .center
      ForEach(viewModel.cards) { card in
            CardView(card: card)
                  .aspectRatio(2/3, contentMode: .fit)
      }
}.foregroundColor(Color.orange)
 .padding(10)
```

- Die View, die vom Container ein Angebot zur Fläche bekommt ist die View, die padding(10) erzeugt.
- Diese bietet der Fläche für die View von `.foregroundColor(Color.orange)` die angebotene Fläche an minus 10 Point an jeder der vier Seiten
- Diese Fläche wird dann dem HStack angeboten
- Der HStack teilt die Fläche gleichmässig zwischen den `.aspectRatio`-Views auf. Dabei wird als Breite die Breite des HStacks verwendet und als Höhe ein Wert gewählt, der dem 2/3-Verhältnis gerecht wird. Sollte das nicht passen, wird die Höhe fest gesetzt und die Breite an das Verhältnis angepasst.
- CardView erhält am Ende eine Fläche vorgeschlagen, die diese ausfüllt.

Views, die die ganze Fläche einnehmen, die ihnen angeboten wird

- Die meisten Views nehmen die gesamte Fläche an, die ihnen angeboten wird. D.h. sie passen sich so an, dass sie die Fläche vollständig einnehmen
- CustomViews sollten dies ebenfalls tun, wenn es sinnvoll ist. Zum Beispiel sollte in einer CardView die Schriftgrösse des Emojis angepasst werden, damit die Symbolgrösse zur Kartenfläche passt.
  - Dies wird durch einen GeometryReader erreicht, der es ermöglicht die entsprechenden Berechnungen durchzuführen

```
var body: View {
      GeometryReader { geometry in
            CardView(…)
      }
}
```

Geometry Reader

Der `geometry` Parameter ist ein GeometryProxy:

```
struct GeometryProxy {
      var size: CGSize
      func frame(in: CoordinateSpace) -> CGRect
      var safeAreaInsets: EdgeInsets
}
```

- Size ist die vom Container angebotene Fläche
  - Dadurch kann in der CardView die Schriftgrösse bestimmt werden

Safe Area

- SafeAreas sind Bereiche, in denen nicht gezeichnet werden soll. Eine bekannte SafeArea ist die „Notch“-Fläche am iPhone.
- Wenn einem View eine Fläche angeboten wird, enthält diese in der Regel keine SafeArea.
- Umschliessende Flächen können eine SafeArea definieren, in die nicht gezeichnet werden soll. Daher der Parameter im GeometryProxy.
- Es ist jedoch möglich diese SafeAreas zu ignorieren und so zum Beispiel den Bereich der Nodge mit Inhalt zu füllen:

```
ZStack { ... }
      .edgesIgnoringSafeArea([.top])
```

Container

- Mithilfe des `.frame(…)`-Modifiers wird einer View die Fläche angeboten, die Ihr zur Verfügung steht.
- Sobald ein View die Grösse bestimmt hat, muss sie sich mithilfe des `.position`-Modifiers positionieren. Dabei ist die position das Zentrum des SubViews im Koordinatensystem des Containers.
- `.frame(…)` hat viele Parameter, die in der Dokumentation genauer beschrieben werden.