Extensions Protocols

Lukas Pühringer genannt Rühri oder Pühri

Unsere Themen Extentions

- Allgemeine Syntax
- Extentions und Properties
- Extentions und Initializer
- Extentions und Methods
- Extentions und Subscripts

Unsere Themen Protocols

- Allgemeine Syntax
- Voraussetzungen
- Protocols als Type
- Delegation
- Protocols und Extentions
- Collections of Protocols
- Inheritance of Protocols
- Class-Only Protocols
- Protocol Composition
- Checking for Comformance
- Optional Protocol Requirements
- Protocol Extentions

Extentions

Allgemeine Syntax

- Kann existierenden Typen, Protocol, Klasse etc. **ergänzen/erweitern**
- **Funktionalität hinzufügen**
- Keyword: **extention**

```
1  extension SomeType {
2      // new functionality to add to SomeType goes here
3  }
```

Extentions und Properties

- Um Typen mit Properties zu **erweitern**
- Unten: Erweiterung der Double Klasse um „Längenumrechner“

```
1  extension Double {
2      var km: Double { return self * 1_000.0 }
3      var m: Double { return self }
4      var cm: Double { return self / 100.0 }
5      var mm: Double { return self / 1_000.0 }
6      var ft: Double { return self / 3.28084 }
7  }
8  let oneInch = 25.4.mm
9  print("One inch is \(oneInch) meters")
10 // Prints "One inch is 0.0254 meters"
11 let threeFeet = 3.ft
12 print("Three feet is \(threeFeet) meters")
13 // Prints "Three feet is 0.914399970739201 meters"
```

Extentions und Properties

- Aufruf dieser neuen Funktionalität mit **wert.funktion**
- Diese Properties sind **read-only.** d.h. ohne **get**

```
1  let aMarathon = 42.km + 195.m
2  print("A marathon is \(aMarathon) meters long")
3  // Prints "A marathon is 42195.0 meters long"
```

Extentions und Initializer

- Typen um **Initializer zu ergänzen**
- Um **eigene** Initializer zu bestehenden Klassen **hinzuzufügen z.B. für Anpassung**

```
1  struct Size {
2      var width = 0.0, height = 0.0
3  }
4  struct Point {
5      var x = 0.0, y = 0.0
6  }
7  struct Rect {
8      var origin = Point()
9      var size = Size()
10 }
```

Extentions und Initializer

- Struktur hat **default values** für alle Properites => default initializer

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
   size: Size(width: 5.0, height: 5.0))
```

- Rect Struktur wird nun **erweitert**

```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

- Neue Initializer für bestehende Struktur

Extentions und Methods

- neue Instanzen und Typ-Methoden zu **existierenden Typen ergänzen**

```
1  extension Int {
2      func repetitions(task: () -> Void) {
3          for _ in 0..<self {
4              task()
5          }
6      }
7  }
```

```
1  3.repetitions {
2      print("Hello!")
3  }
4  // Hello!
5  // Hello!
6  // Hello!
```

Mutating Instance Methods

- Um die **Instanzen** von Typen selbst zu **verändern**
- Stucts und Enums die **self müssen** mit **mutating**

```
1  extension Int {
2      mutating func square() {
3          self = self * self
4      }
5  }
6  var someInt = 3
7  someInt.square()
8  // someInt is now 9
```

Extentions und Subscripts

- Um Subscripts zu **bestehenden Typen** hinzuzufügen
- Gibt Nummer zurück, das **rechts vom eingegeben Index** steht

```
1  extension Int {
2      subscript(digitIndex: Int) -> Int {
3          var decimalBase = 1
4          for _ in 0..<digitIndex {
5              decimalBase *= 10
6          }
7          return (self / decimalBase) % 10
8      }
9  }
10 746381295[0]
11 // returns 5
```

Beispiel

- Schreibe eine Klasse **schclass** mit:
  1. Schüleranzahl
  2. Klassenname
  3. KV
  4. Schüler[]
- Erweitere die Klasse Traunlauf (von Max) **mit einer Extension** so dass die Läufer gleich zu Objekten in der Schulklasse werden.
- Erweitere die Klasse zzl. um eine Funktion die die Bestzeit herausfindet

Protocols

Allgemeine Syntax

- Protocols geben **„Blaupausen der Instanzen“** an. Passend für die aktuelle Aufgabe
- Kann dann von einer Klasse **adoptiert/genutzt** werden
- Protocols kann man auch mit Extentions erweitern

```
1  protocol SomeProtocol {
2      // protocol definition goes here
3  }
```

Allgemeine Syntax

- Protocol können genutzt werden:

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}
```

- Mehrere Protocols: **Trennung Semikolon** (wie in Kotlin)

- Bei Benutzung einer **abgeleiteten Klasse** zuerst Klasse dann Protocol

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

Voraussetzungen

- Wichtigste Vorraussetzung: conforming type für **Instance Property oder Type Property**
- Protocol spezifiert **nicht den Property-Type**

```swift
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

Voraussetzungen

- Verlangt Protocol **gettable oder settable** zu sein
  - -> Property Vorraussetzung **nicht** erfüllbar mit **stored oder readonly properties**
- Verlangt nur **gettable** -> jede Art von Property benutzen
- Vorraussetzungen (requirements) immer **{get set} (ähnlich C#)**

```
1  protocol SomeProtocol {
2      var mustBeSettable: Int { get set }
3      var doesNotNeedToBeSettable: Int { get }
4  }
```

Voraussetzungen

- Protocols mit **Type Property** with **static** keyword
- Erlaubt prefixen mit **class** oder **static**

```
181 struct Room {
182     static var squareMeters = 50
183 }
184
185 var roomSize = Room.squareMeters // 50
186
187 enum Square {
188     static var sides = 5
189     case side
190 }
191
192 var square = Square.sides //5
```

```
1  protocol AnotherProtocol {
2      static var someTypeProperty: Int { get set }
3  }
```

Voraussetzungen

- FullyNamed **braucht**:
  - Conforming type
  - Gibt nichts genaueres an
  - Nur dass es einen Namen braucht um das zu providen
  - **UND**: jede Klasse die einbindet, braucht eine Variable **fullName**:String

```
protocol FullyNamed {
    var fullName: String { get }
}
```

```
struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

Voraussetzungen

```swift
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```

Method Requirements

- Protocols **können** type methoden brauchen
- Werden als **Teil** der Protokolldefinition geschrieben
  - Wie normale Instanzen
- Type methods mit **static kennzeichnen**

```
1  protocol SomeProtocol {
2      static func someTypeMethod()
3  }
```

Method Requirements

```
1  protocol RandomNumberGenerator {
2      func random() -> Double
3  }
```

- **Abgeleitete** Klassen brauchen **Random()**
- Protocol keine Spezifikation der Erzeugun

Method Requirements

```
1  class LinearCongruentialGenerator: RandomNumberGenerator {
2      var lastRandom = 42.0
3      let m = 139968.0
4      let a = 3877.0
5      let c = 29573.0
6      func random() -> Double {
7          lastRandom = ((lastRandom * a +
     c).truncatingRemainder(dividingBy:m))
8          return lastRandom / m
9      }
10 }
11 let generator = LinearCongruentialGenerator()
12 print("Here's a random number: \(generator.random())")
13 // Prints "Here's a random number: 0.3746499199817101"
14 print("And another one: \(generator.random())")
15 // Prints "And another one: 0.729023776863283"
```

Mutating Method

- Auf deutsch: **Methoden ändern**
- Dafür: Instanz im **Protocol verändern**
- Bei protocol definition wo die Instanz verändert werden soll. Dann **mutating** Keyword nicht vergessen!

```
1  protocol Togglable {
2      mutating func toggle()
3  }
```

Mutating Methods

- Implementation von Toggleable
- Im Methodenkopf: **mutating**

```
1  enum OnOffSwitch: Togglable {
2      case off, on
3      mutating func toggle() {
4          switch self {
5          case .off:
6              self = .on
7          case .on:
8              self = .off
9          }
10     }
11 }
12 var lightSwitch = OnOffSwitch.off
13 lightSwitch.toggle()
14 // lightSwitch is now equal to .on
```

Initializer

- Spezielle Initializer (Konstruktoren).
- Als **Teil des Protocols**

```
1  protocol SomeProtocol {
2      init(someParameter: Int)
3  }
```

Initializer

- Implementierung:

```
1  class SomeClass: SomeProtocol {
2      required init(someParameter: Int) {
3          // initializer implementation goes here
4      }
5  }
```

- **required keyword**
- Legt fest: **explicit** oder **abgeleitete** Form des **init**
- Für alle Subklassen von SomeClass dann obligatorisch

Initializer

```
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from
  SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}
```

Protocols als Type

- Protocols **ohne Funktionalität**
- Nutzung als Instanz des Protocol Typen möglich

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

Protocols als Type

```swift
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    print("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

Delegation

- Ist Designpattern (DP)
- Gibt einige Verantwortlichkeiten ab
- DP: Protocol definieren welches die Grundfunktionalität hat so dass abgeleiteter Type diese implementieren kann

Delegation

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate: AnyObject {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll:
  Int)
    func gameDidEnd(_ game: DiceGame)
}
```

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll:
  Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

Delegation

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Started a new game of Snakes and Ladders
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
// Rolled a 5
// The game lasted for 4 turns
```

Protocols und Extentions

- **Existierende** Typen anpassen für Protocol
- Auch ohne Source Code Zugang

- **Bereits erwähnt: Extentions hinzufügen von:**
  - Properties
  - Methods
  - Subscripts
  - ➔ für **jede Anforderung** Anpassung möglich

Protocols und Extentions

- Kann von jedem Typ implementiert werden

```
protocol TextRepresentable {
    var textualDescription: String { get }
}
```

- Dice Beispiel von vorher:

```
extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}
```

Extentions und Protocols

```
1  let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
2  print(d12.textualDescription)
3  // Prints "A 12-sided dice"
```

- **Implementiert** Protocol gleich

Extentions und Protocols

- Ein eigener Type: Spezifikationen eines Protocols unter gewissen Voraussetzungen erfüllen
- Type **erweitern** -> Anforderung erfüllt

```
1  extension Array: TextRepresentable where Element: TextRepresentable {
2      var textualDescription: String {
3          let itemsAsText = self.map { $0.textualDescription }
4          return "[" + itemsAsText.joined(separator: ", ") + "]"
5      }
6  }
7  let myDice = [d6, d12]
8  print(myDice.textualDescription)
9  // Prints "[A 6-sided dice, A 12-sided dice]"
```

Extentions und Protocols

- Erfüllt Type alle Voraussetzungen **ohne** Protocol-Einbindung
- Protocol mit **empty extention**

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
print(somethingTextRepresentable.textualDescription)
// Prints "A hamster named Simon"
```

Collections of Protocol Types

- Protocol als Type **verwendbar**

```
let things: [TextRepresentable] = [game, d12, simonTheHamster]
```

- **Iteration** über Protocol-Array/Collection möglich

```
for thing in things {
    print(thing.textualDescription)
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

Protocol Inheritance

- Protocol kann **ein bis mehrere** Protocols erben
- D.h.: **weitere** Voraussetzungen werden dann **hinzugefügt**

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

```
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

Class-Only Protocols

- **Limitieren erbende** Protocols (nicht Strukturen und Enums)

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

- Kann **nur** von Class-Types geerbt werden
- Sonst **Compile-Time-Error**

Protocol Composition

- Sinnvoll ist einen Typen zu **verlangen** wenn der oft verlangt wird
- Mit **Protocol Composition**: mehrere **zusammenfassen**
- Verhält sich wie **temporäres lokales protocol**
- Definiert **kein** neuen Protocol typen

- **Hat funktion die Übergabe Werte des Composition Protocols dann ist jeder Type recht der den Anforderungen der Protocols passt!**

Protocol Composition

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(to celebrator: Named & Aged) {
    print("Happy birthday, \(celebrator.name), you're \
  (celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)
// Prints "Happy birthday, Malcolm, you're 21!"
```

Checking for Protocol Conformance

- **is** und **as** Operatoren für Protocol Conformance nutzen
- **Und** für Casten

- **is** liefert **true** wenn die Instanz konform des Protocols ist, sonst **false.**
- **as** liefert **nicht nil** (eig. Optional) wenn die abgeleitete Klasse dem entspricht
- **as!** liefert erzwingt den **cast nach unten** und **runtime error** wenns fehlschlägt.

Checking for Protocol Conformance

```
protocol HasArea {
    var area: Double { get }
}
```

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

Optional Protocol Requirements

- **Optional** Vorraussetzungen im Protocol möglich
- nicht unbedingt zu implementieren
- **Keyword: optional**
- Optional und protocol damit mit **@objc** deklarieren
- Für Code von Objective C

definition. Optional requirements are available so that you can write code that interoperates with Objective-C. Both the protocol and the optional requirement must be marked with the `@objc` attribute. Note that `@objc` protocols can be adopted only by classes that inherit from Objective-C classes or other `@objc` classes. They can't be adopted by structures or enumerations.

Optional Protocol Requirements

- Methode/property mit Optional Requirement -> Optional selbst
- (Int) -> String wird ((Int)->String)?
- **Ganze Funktion** als Optional kennzeichnen

```
1  @objc protocol CounterDataSource {
2      @objc optional func increment(forCount count: Int) -> Int
3      @objc optional var fixedIncrement: Int { get }
4  }
```

Optional Protocol Requirements

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount: count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

Protocol Extentions

- Protocol **erweitern** um Methoden, Initializer, Subscripts etc.
- **Verhalten** des Protocol selbst wird dadurch definiert
- **Besser wie in globalen Funktionen oder jeden Typen einzeln.**

```
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
```

```
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.3746499199817101"
print("And here's a random Boolean: \(generator.randomBool())")
// Prints "And here's a random Boolean: true"
```

Protocol Extentions

- Wenn man Protocol erweitert -> **Constraints** für die jeweiligen Typen hinzufügbar
- Diese müssen **zuerst** erfüllt werden!
- Constraints mit where keyword.

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

Ende

Danke!