App-Entwicklung für iOS und OS X

Aktualisiert für iOS 10, Xcode 8 und Swift 3

SS 2017

Stephan Gimbel

Heute

- Mehr Swift und das Foundation Framework
  - Was genau sind Optionals?
  - Tuples
  - `Range<T>`
  - Datenstrukturen, Methoden und Properties
  - `Array<T>`, `Dictionary<K,V>`, `String`, und andere
  - Initialisierung
  - `AnyObject`, Introspection und Casting (`is` und `as`)
  - `User Defaults`
  - `assert`

Optional

- Ein Optional ist nur ein enum
  - Anders ausgedrückt...

```
enum Optional<T> { // das <T> ist ein Generic, wie z.B. in Array<T>
    case None
    case Some(T)
}
```

Optional

- Ein Optional ist nur ein enum
  - Anders ausgedrückt…

```
enum Optional<T> { // das <T> ist ein Generic, wie z.B. in Array<T>
   case None
   case Some(T)
}
```

```
let x: String? = nil
```

… ist …

```
let x = Optional<String>.None
```

```
let x: String? = “hello”
```

… ist …

```
let x = Optional<String>.Some(“hello”)
```

Optional

- Ein Optional ist nur ein `enum`
  - Anders ausgedrückt…

```
enum Optional<T> { // das <T> ist ein Generic, wie z.B. in Array<T>
   case None
   case Some(T)
}
```

```
let x: String? = nil
```

… ist …

```
let x = Optional<String>.None
```

```
let x: String? = “hello”
```

… ist …

```
let x = Optional<String>.Some(“hello”)
```

```
let y = x!
```

… ist …

```
switch x {
   case some(let value): y = value
   case none: // raise exception
}
```

Optional

- Ein Optional ist nur ein enum
  - Anders ausgedrückt…

```
enum Optional<T> { // das <T> ist ein Generic, wie z.B. in Array<T>
   case None
   case Some(T)
}
```

```
let x: String? = nil
```

… ist …

```
let x = Optional<String>.None
```

```
let x: String? = “hello”
```

… ist …

```
let x = Optional<String>.Some(“hello”)
```

```
let y = x!
```

… ist …

```
switch x {
   case some(let value): y = value
   case none: // raise exception
}
```

```
let x: String? = …

if let y = x {
   // do something with y
}
```

… ist …

```
switch x {
   case .some(let y):
      // do something with y
   case .none:
      Break
}
```

Optional

- Optional können “chained” werden
  - Zum Beispiel ist `hashValue` ein `var` in `String`
  - Was also, wenn wir ein `hashValue` von einem Optional `String` haben wollen?
  - Und was, wenn der Optional `String` selbst der `text` eines Optional `UILabel` ist?

```
var display: UILabel? // Stellen Sie sich dies als @IBOutlet ohne implicit unwrap vor !
if let temp1 = display {
   if let temp2 = temp1.text {
      let x = temp2.hashValue
         …
   }
}
```

… mit Optional Chaining unter Verwendung von `?` statt `!` zum unwrapping, wird dies zu…

```
if let x = display?.text?.hashValue { … }  // x ist ein Int
   let x = display?.text?.hashValue { … }  // x ist ein Int?
```

Optional

- Ebenfalls existiert ein Optional “defaulting” Operator ??
  - Was also, wenn wir einen `String` in ein `UILabel` stecken wollen, wenn es aber `nil` ist, stattdessen ein “ “ (space) in das `UILabel` stecken wollen?

```
let s: String? = … // kann nil sein
if s != nil {
   display.text = s
} else {
   display.text = “ “
}
```

… kann viel einfacher ausgedrückt werden …

```
display.text = s ?? “ ”
```

Tuples

- Was ist ein Tuple?
  - Eine Gruppierung von Values
  - Kann überall da verwendet werden, wo auch Typen verwendet werden können

```
let x: (String, Int, Double) = (“hello”, 5, 0.85) // Typ von x ist “ein Tuple”
let (word, number, value) = x // Benamung der Elemente erfolgt beim Zugriff auf das Tuple
print(word)   // hello
print(number) // 5
print(value)  // 0.85
```

... Elemente können aber auch bei der Deklaration benannt werden ...

```
let x: (w: String, i: Int, v: Double) = (“hello”, 5, 0.85)
print(x.w) // hello
print(x.i) // 5
print(x.v) // 0.85
let (wrd, num, val) = x // Auch ok (benennt Tuple Elemente bei Zugriff um)
```

Tuples

- Tuples als Rückkgabewerte
  - Wir können Tuples nutzen um mehrere Werte aus einer Funktion oder Methode zurückkzugeben…

```
func getSize() -> (weight: Double, height: Double) { return (250, 80) }

let x = getSize()
print("weight is \(x.weight)") // 250
```

… oder …

```
print("height is \(getSize().height)") // 80
```

Range

- Range
  - Eine `Range` sind zwei Endpunkte in Swift
  - Eine `Range` kann Dinge repräsentieren wie etwa eine Selektion in Text oder einen Bereich in einem `Array`
  - `Range` ist generisch (z.B. `Range<T>`), aber `T` ist eingeschränkt (z.B. vergleichbar)
  - Dies ist eine Pseudo-Repräsentation von `Range` ...

```
struct Range<T> {
   var startIndex: T
   var endIndex: T
}
```

  - So ist z.B. ein `Range<Int>` geeignet um einen Bereich in einem `Array` zu beschreiben
  - Es existieren andere Ranges mit höheren Fähigkeiten wie `CountableRange`
  - Eine `CountableRange` beinhaltet fortlaufende Werte, über die iteriert werden oder in die indexiert werden kann

Range

- Range
  - Es existiert eine spezielle Syntax zur Erstellung eines Range
  - Entweder `..<` (ohne die obere Grenze) oder `...` (inklusive beider Grenzen)

```
let array = [”a”,”b”,”c”,”d”]
let a = array[2...3] // Array welches [“c”,”d”] beinhaltet
let a = array[2..<3] // Array welches [“c”] beinhaltet
let a = array[6...8] // runtime crash (array index out of bounds)
let a = array[4...1] // runtime crash (untere Grenze muss kleiner als obere Grenze sein)
```

  - Eine String subrange ist kein `Range<Int>` (es ist `Range<String.Index>`)

```
let e = “hello”[2..<4]       // dies ist nicht (!=) “ll”, es kompiliert noch nicht einmal
let f = “hello”[start..<end] // dies ist möglich, start und end besprechen wir später
```

Range

- Range
  - Wenn der Typ der oberen/unteren Grenze ein Int ist, erstellt `..<` ein `CountableRange`
    (Um genau zu sein, hängt es davon ab ob die obere/untere Grenze "strideable by `Int`" ist)
  - `CountableRange` ist enumerierbar mit `for in`
  - Hier ein Beispiel wie wir ein C-Loop for (i = 0; i < 20; i++) wie durchführen können...

```
for i in 0..<20 {
}
```

  - Aber was ist z.B. mit etwas wie `for (i = 0.5; i <= 15.25; i += 0.3)`?
  - Floating Point Zahlen
  - So ist `0.5...15.25` nur eine Range, nicht eine `CountableRange` (welche gebraucht wird für `for in`)
  - Glücklicherweise gibt es eine globale Funktion, welche eine CountableRange aus Floating Point Werten erstellt

```
for i in stride(from: 0.5, through: 15.25, by: 0.3) {
}
```

  - Der Rückgabe Typ von `stride` ist `CountableRange` (genauer `ClosedCountableRange` in diesem Fall)

Datenstrukturen in Swift

- Classes, Structures und Enumerations
  - Dies sind die drei von view fundamentalen Grundbausteine für Datenstrukturen in Swift

- Gemeinsamkeiten
  - Syntax zur Deklaration…

```
class CalculatorBrain {

}
struct Vertex {

}
enum Op {

}
```

Datenstrukturen in Swift

- Classes, Structures und Enumerations
  - Dies sind die drei von view fundamentalen Grundbausteine für Datenstrukturen in Swift
- Gemeinsamkeiten
  - Syntax zur Deklaration…
  - Properties und Functions…

```
func doit(argx argi: Type) -> ReturnValue {

}

var storedProperty = <initial value> (kein enum)

}

var computedProperty: Type {
   get {}
   set {}
}
```

Datenstrukturen in Swift

- Classes, Structures und Enumerations
  - Dies sind die drei von view fundamentalen Grundbausteine für Datenstrukturen in Swift
- Gemeinsamkeiten
  - Syntax zur Deklaration…
  - Properties und Functions…
  - Initializers (nochmal, kein enum)…

```
init(arg1x arg1i: Type, arg2x arg2i: Type, …) {

}
```

Datenstrukturen in Swift

- Classes, Structures und Enumerations
  - Dies sind die drei von view fundamentalen Grundbausteine für Datenstrukturen in Swift
- Gemeinsamkeiten
  - Syntax zur Deklaration…
  - Properties und Functions…
  - Initializers (nochmal, kein enum)…
- Unterschiede
  - Vererbung (`class` only)
  - Value Type (`struct`, `enum`) vs. Reference Type (`class`)

Value vs. Reference

- Value (`struct` und `enum`)
  - Kopiert wenn als Parameter an eine Funktion übergeben
  - Kopiert wenn zu einer anderen Variable zugewiesen
  - Immutable wenn zu einer Variable mit `let` zugewiesen (Funktionsparameter sind `let`)
  - Jede `func`, die ein `struct`/`enum` ändern kann, muss mit dem Schlüsselwort `mutating` versehen werden
- Reference (`class`)
  - Gespeichert auf dem Heap und "reference counted" (automatisch)
  - Konstante Pointer zu einer Klasse (`let`) können immer noch geärndert werden durch Aufruf von Methoden und Änderung von Properties
  - Wenn als Parameter übergeben, wird keine Kopie gemacht (nur ein Pointer auf die gleiche Instanz übergeben)
- Was benutzen?
  - Haben wir bereits diskutiert (`class` vs. `struct`) und ist ebenfalls im Reading Assignment enthalten
  - Verwendung von `enum` ist Situationsabhängig (immer dann, wenn wir Daten mit diskreten Werten haben)

Methoden

- Parameter Namen
  - Alle Parameter zu einer Funktion haben einen internal und einen external Namen

```
func foo(externalFirst first: Int, externalSecond second: Double) {
   var sum = 0.0
   for _ in 0..<first { sum += second }
}

func bar() {
   let result = foo(externalFirst: 123, externalSecond: 5.5)
}
```

Methoden

- Parameter Namen
  - Alle Parameter zu einer Funktion haben einen internal und einen external Namen
  - Der internal Name ist der Name der lokalen Variable, die in der Methode verwendet wird

```
func foo(externalFirst first: Int, externalSecond second: Double) {
   var sum = 0.0
   for _ in 0..<first { sum += second }
}

func bar() {
   let result = foo(externalFirst: 123, externalSecond: 5.5)
}
```

Methoden

- Parameter Namen
  - Alle Parameter zu einer Funktion haben einen internal und einen external Namen
  - Der internal Name ist der Name der lokalen Variable, die in der Methode verwendet wird
  - Der external Name ist was der Caller nutzt, wenn die Methode aufgerufen wird

```
func foo(externalFirst first: Int, externalSecond second: Double) {
   var sum = 0.0
   for _ in 0..<first { sum += second }
}

func bar() {
   let result = foo(externalFirst: 123, externalSecond: 5.5)
}
```

Methoden

- Parameter Namen
  - Alle Parameter zu einer Funktion haben einen internal und einen external Namen
  - Der internal Name ist der Name der lokalen Variable, die in der Methode verwendet wird
  - Der external Name ist was der Caller nutzt, wenn die Methode aufgerufen wird
  - Wir können _ verwenden, wenn wir nicht wollen dass der Caller den external Namen für einen bestimmten Parameter verwendet (wird meist nur für den ersten Parameter so gemacht)

```
func foo(_ first: Int, externalSecond second: Double) {
   var sum = 0.0
   for _ in 0..<first { sum += second }
}

func bar() {
   let result = foo(123, externalSecond: 5.5)
}
```

Methoden

- Parameter Namen
  - Alle Parameter zu einer Funktion haben einen internal und einen external Namen
  - Der internal Name ist der Name der lokalen Variable, die in der Methode verwendet wird
  - Der external Name ist was der Caller nutzt, wenn die Methode aufgerufen wird
  - Sie können _ verwenden, wenn Sie nicht wollen dass der Caller den external Namen für einen bestimmten Parameter verwendet (wird meist nur für den ersten Parameter so gemacht)
  - Wenn wir nur einen Parameternamen nutzen, sind internal und external Name identisch

```
func foo(first: Int, second: Double) {
   var sum = 0.0
   for _ in 0..<first { sum += second }
}

func bar() {
   let result = foo(first: 123, second: 5.5)
}
```

Methoden

- Natürlich können Methoden/Properties überschrieben werden
  - Vor die `func` oder `var` einfach das Schlüsselwort `override` hinzufügen
  - Eine Methode kann als `final` markiert werden, welches Subclasses davon abhält diese zu überschreiben
  - Auch Klassen können als `final` markiert werden

- Sowohl Typen als auch Instanzen können Methoden/Properties haben
  - Für dieses Beispiel, betrachten wir die Verwendung des `structs Double` (ja, `Double` ist ein `struct`)

```
var d: Double = …
if d.isSignMinus {
   d = Double.abs(d)
}
```

  - `isSignMinus` ist ein Instanz Property eines `Doubles` (geschickt an ein bestimmtes `Double`)
  - `abs` ist eine Type Methode von `Double` (geschickt zum Typ selbst, nicht an ein bestimmtes `Double`)
  - Eine Typ Methode oder ein Property wird mittels `static` Prefix deklariert…

```
static func abs(d: Double) -> Double
```

Methoden

- Sowohl Typen als auch Instanzen können Methoden/Properties haben
  - Typ Methoden und Properties sind mit dem Schlüsselwort `static` bezeichnet
  - Zum Beispiel hat das `struct Double` `vars` und `funcs` auf seinen Typ
  - Dies sind keine Methoden oder `vars`, auf die wir mit einer Instanz eines `Double` zugreifen können (z.B. 53.2)
  - Der Zugriff erfolgt stattdessen durch referenzierung des `Double` Types selbstvar d: Double = ...

```
static func abs(d: Double) -> Double   { if d < 0 { return -d } else { return d} }
static var pi: Double                  { return 3.1415926 }
let d = Double.pi                      // d = 3.1415926
let d = Double.abs(-324.44)            // d = 324.44
Let x: Double = 23.85
let e = x.pi                           // pi ist keine instance var
let e = x.abs(-22.5)                   // abs ist keine instance method
```

Properties

- Property Observers
  - Wir können Änderungen eines Properties beobachten mittels `willSet` und `didSet`
  - Wird auch ausgelöst bei Änderung eines `struct` (z.B. hinzufügen zu einem Dictionary)
  - Ein Observer wird oft im Controller verwendet, um ein Update des User-Interfaces durchzuführen

```
var someStoredPropery: Int = 42 {
   willSet { newValue ist der neue Wert }
   didSet { oldValue ist der alte Wert }
}

override var inheritedProperty {
   willSet { newValue ist der neue Wert }
   didSet { oldValue ist der alte Wert }
}

var operations: Dictionary<String, Operation> = [ … ] {
   willSet { wird ausgeführt, wenn eine Operation hinzugefügtgt/entfernt wird }
   didSet { wird ausgeführt, wenn eine Operation hinzugefügtgt/entfernt wird }
}
```

Properties

- Lazy Initialization
  - Ein `lazy` Property wird nicht initialisiert, bis darauf zugegriffen wird
  - Wir können ein Objekt allokieren, ein Closure ausführen oder eine Methode aufrufen

```
lazy var brain = CalculatorBrain() // gut wenn CalculatorBrain viele Resourcen hat

lazy var someProperty: Type = {
    // erstellen eines Values für someProperty
    return <erstellter Value>
}()

lazy var myProperty = self.initializeMyProperty()
```

  - Stellt sicher, dass die “alle Properties müssen initialisiert sein” Regel erfüllt ist
  - Leider können Dinge die so initialisiert sind keine Konstanten sein (`var` ist ok, `let` nicht ok)
  - Kann genutzt werden um schwierige Abhängigkeiten bei der Initialisierung zu umgehen

Array

- Array

```
var a = Array<String>()
```

... ist das gleiche wie ...

```
var a = [String]()

let animals = [“Giraffe”, “Cow”, “Doggie”, “Bird”]
animals.append(“Ostrich”) // kompiliert nicht, da animals immutable ist (da let)
let animal = animals[4]   // crash (Array index out of bounds)

// enumeration eines Array (ist eine “sequenz” wie ein CountableRange
for animal in animals {
   print(animal)
}
```

Array

- Interessante `Array<T>` Methoden die ein Closure annehmen
  - Erstellt ein neues Array und filtert "unerwünschtes" heraus
  - Die Function welche als Parameter übergeben wird, gibt `false` zurück wenn ein Element unerwünscht ist

```
filter(includeElement: (T) -> Bool) -> [T]
let bigNumbers = [2,47,118,5,9].filter({ $0 > 20 }) // bigNumbers = [47,118]
```

  - Erstelle ein Array durch Transformation jeden Elementes zu etwas anderem
  - Das "Ding" in das es transformiert wird, kann ein anderer Typ sein als das was im Array ist

```
map(transform: (T) -> U) -> [U]
let stringified: [String] = [1,2,3].map { String($0) } // ["1","2","3"]
```

  - Reduktion eines kompletten Arrays zu einem einzigen Wert

```
reduce(initial: U, combine: (U, T) -> U) -> U
let sum: Int = [1,2,3].reduce(0) { $0 + $1 } // summiert alle Zahlen des Arrays
let sum = [1,2,3].reduce(0, +) // identisch, da + eine Funktion in Swift ist
```

Dictionary

- Dictionary

```
var pac12teamRankings = Dictionary<String, Int>()
```

... ist das gleiche wie ...

```
Var pac12teamRankings = [String:Int]()

pac12teamRankings = [“Stanford”:1, “USC”:11]
let ranking = pac12teamRankings[“Ohio State”] // wenn ranking ein Int? ist (wäre es nil)

// Verwendung eines Tuple mit for-in um ein Dictionary zu enumerieren
for (key, value) in pac12teamRankings {
   print(“Team \(key) ist auf Platz  \(value)”)
}
```

String

- Die Characters in einem String
  - Ein `String` besteht aus Unicodes, es existiert aber ebenfalls das Konzept eines `Character`
  - Ein `Character` ist, was ein Mensch als einzelner lexikalischer Character wahrnehmen würde
  - Dies trifft auch zu, wenn es aus mehreren Unicodes besteht
  - So kann Café aus 5 Unicodes bestehen, aber es sind 4 `Characters`

  - Wir können auf jeden Character (vom Typ `Character`) in einem Strunk mittels `[]` Notation zugreifen
  - Aber die Indizes in `[]` sind keine `Int`, sondern vom Typ `String.Index`

```
let s: String = “hello”                          // Was, wenn wir s[0] haben wollen (d.h. das “h”)?
let firstIndex: String.Index = s.startIndex //der Typ von firstIndex ist kein Int
let firstChar: Character = s[firstIndex]     // firstChar = der Character h
let secondIndex: String.Index = s.index(after: firstIndex)
let secondChar: Character = s[secondIndex]  // secondChar = e
let fifthChar: Character = s[s.index(firstIndex, offsetBy: 4)] // fifthChar = o
let substring = s[firstIndex…secondIndex]    // substring = “he”
```

String

- Die Characters in einem String
  - Auch wenn `String` indexierbar ist (mittels `[]`), ist es keine Collection oder Sequence (wie etwa ein `Array`)
  - Nur Sequences und Collections können Dinge wir `for in` oder `index(of:)`
  - Glücklicherweise gibt die `characters` `var` in `String` eine Collection von Characters des `String`s zurück

  - Damit können wir Dinge tun wie…

```
for c: Character in s.characters { } // iterieren durch alle Characters in s
let count = s.characters.count       // Wie viele Characters sind in s?
let firstSpace: String.Index = s.characters.index(of: " ") // Ein String.Index in die String
                                                           Characters matched ein String.Index in den String
```

String

- `String` ist ein Value Type (es ist ein `struct`)
  - Ob wir die Characters modifizieren können, hängt von `var` vs. `let` ab

```
let hello = “hello”     // immutable String
var greeting = hello    // mutable String
hello += “ there”       // Geht nicht, da hello immutable ist
greeting += “ there”    // greeting ist jedoch ein var und daher mutable
print(greeting)         // “hello there”
print(hello)            // “hello”
```

  - Natürlich können wir Strings auf noch viel kompliziertere Weise beeinflussen…

```
if let firstSpace = greeting.characters.index(of: “ “) {
   // insert)contentsOf:at:) fügt eine Collection von Characters am angegebenen Index ein
   greeting.insert(contentsOf: “ you”.characters, at: firstSpace)
}
print(greeting)         // “hello you there”
```

String

- Andere `String` Methoden

```
var endIndex: String.Index // Niemals ein valider Index in den String

hasPrefix(String) -> Bool
hasSuffix(String) -> Bool

var localizedCapitalized/Lowercase/Uppercase: String

func replaceSubrange(Range<String.Index>, with: String)
```

z.B. `s.replaceSubrange(s.startIndex..<s.endIndex, with: “new contents”)`

```
func components(separatedBy: String) -> [String]
```

z.B. `let array = “1,2,3”.components(separatedBy: “,”) // array = [“1”,”2”,”3”]`

- Und noch viel, viel mehr. Nachlesen in der Doku!

Andere Klassen

- NSObject
  - Basisklasse für alle Objective-C Klassen
  - Für einige fortgeschrittene Features müssen wir NSObject subclassen
- NSNumber
  - Generische Klasse zum speichern von Zahlen (d.h. Reference Type)

    `let n = NSNumber(35.5)` oder `let n: NSNumber = 35.5`

    ```
    let intified: Int = n.intValue // auch doubleValue, boolValue, etc.
    ```
- Date
  - Value Type, genutzt um das aktuelle Datum und Zeit zu ermitteln oder vergangene/zukünftige Dates zu speichern
  - Siehe auch Calendar, DateFormatter, DateComponents
  - Wenn wir ein Datum in UI anzeigen, dann beeinflusst dies die Darstellung
- Data
  - Value Type “bag o’ bits”. Verwendet um raw data im iOS SDK zu speichern/laden/übertragen

Initialization

- Wann wird eine `init` Methode benötigt?
  - `init` Methoden sind nicht so häufig, da Defaults für Properties mittels `=` gesetzt werden können
  - Manchmal sine Properties auch Optionals, sie also zu Beginn `nil`
  - Wir können Properties auch mittels eines Closures initialisieren
  - Oder `lazy` instantiation verwenden
  - Wir brauchen `init` also nur, wenn ein Property auf keine dieser Arten gesetzt werden kann
  - Wir können so viele `init` Methoden in einer `class` oder `struct` haben wie wir wollen
  - Natürlich hat jedes `init` andere Parameter

  - Caller können unser(e) init(s) verwenden durch Verwendung des Typ-Namens und zur Verfügung stellen der Parameter, welche wir benötigen

```
var brain = CalculatorBrain()
var pendingBinaryOperation = PendingBinaryOperation(function: +, firstOperand: 23)
let textNumber = String(45.2)
let emptyString = String()
```

Initialization

- Ein paar `init` Methoden bekommen wir "geschenkt"
  - Geschenktes `init()` (d.h. ein `init` ohne Parameter) ist gegeben für alle Base `class`es
  - Eine Base `class` hat keine superclass

  - Wenn ein `struct` keine Initializer hat, bekommt es per Default einen mit allen Properties als Parameter

```
struct PendingBinaryOperation {
   var function: (Double, Double) -> Double
   var firstOperand: Double

   init(function: (Double, Double) -> Double, firstOperand: Double) {
      // für uns geschenkt!
   }
}

// verwenden des geschenkten Initializers irgendwo in unserem Code
let pbo = PendingBinaryOperation(function: f, firstOperand: accumulator)
```

Initialization

- Was können wir innerhalb von `init` machen?
  - Wir können für jedes Property ein Value setzen, auch wenn es schon ein Default-Value hat
  - Konstante Properties (d.h. Properties die mit `let` deklariert wurden) können gesetzt werden
  - Wir können andere `init` Methoden in der eigenen `class` oder `struct` aufrufen mittels `self.init(<args>)`
  - In einer `class`, können wir ebenfalls `super.init(<args>)` aufrufen
  - Es existieren aber ein paar Regeln für den Aufruf von `init`s aus anderen `init`s in einer `class`...

Initialization

- Was wird von uns erwartet in `init` zu tun?
  - Zur Zeit wenn ein `init` fertig ist, müssen alle Properties ein Value haben (Optionals können `nil` sein)
  - Es existieren zwei Arten von `inits` in einer `class`: `convenience` und `designated` (d.h. nicht `convenience`)
  - Ein designated `init` muss (und kann nur) einen designated `init` aufrufen, welcher in seiner unmittelbaren `super`class ist
  - Wir müssen alle Properties, die von unserer eigenen Klasse vorgestellt wurden initialisieren bevor wir ein `init` der superclass aufrufen
  - Wir müssen ein `init` einer superclass aufrufen bevor wir einen Wert zu einem vererbten Property zuweisen
  - Ein `convenience init` muss (und kann nur) einen `init` in seiner eigenen class aufrufen
  - Ein `convenience init` muss diesen init aufrufen bevor ein Wert für ein Property gesetzt werden kann
  - Der Aufruf von anderen `inits` muss beendet sein, bevor wir auf Properties zugreifen oder Methoden aufrufen können

Initialization

- Vererbung von `init`
  - Wenn wir keinen designated init implementieren, erben wir alle designated inits unserer superclass
  - Wenn wir alle designated `init`s unserer superclass überschreiben, erben wir all ihre `convenience init`s
  - Wenn wir keine `init`s implementieren, erben wir alle `init`s aus der superclass
  - Jedes `init` welches wir durch diese Regeln erben, muss die Regeln auf der vorherigen Folie erfüllen

- Required `init`
  - Eine `class` kann eine oder mehrere seiner init Methoden als `required` markieren
  - Eine subclass muss dann die besagte `init` Methode implementieren (obwohl sie vererbt werden können, nach obigen Regeln)

Initialization

- Failable `init`
  - Wenn ein init mit ? nach dem Wort `init` deklariert ist, gibt es ein Optional zurück

```
init?(arg1: Type1, …) {
   // kann hier nil zurück geben (was bedeutet, dass init misslungen ist)
}
```

  - Beispiel…

```
let image = UIImage(named: “foo”) // image ist ein Optional UIImane (d.h. UIImage?)
```

Normalerweise würden if-let für diese Fälle nutzen…

```
if let image = UIImage(named: “foo”) {
   // image successfully reated
} else {
   // couldn’t create image
}
```

Any & AnyObject

- Any & AnyObjects sind spezielle Typen
  - Diese Typen wurden häufig aus Kompatibilitätsgründen mit alter Objective-C API verwendet
  - Nicht mehr so häufig in iOS 10, da die alten Objective-C APIs geupdated wurden
  - Variablen vom Typ `Any` können etwas von jedem beliebigen Typ speichern (`AnyObject` können nur `classes` speichern)
  - Swift ist eine streng getypte Sprache, also können wir keine Methode auf `Any` ausführen
  - Wir müssen dies zuerst in einen konkreten Type umwandeln
  - Eine der schönen Eigenschaften von Swift ist das strong Typing, so sollten wir generell `Any` vermeiden

Any & AnyObject

- Wo kommt es vor in iOS?
  - Manchmal (selten) ist es ein Parameter zu einer Funktion die verschiedene Dinge annehmen kann
  - Hier eine UIViewController Methode die einen sender beinhaltet (der von jedem Typ sein kann)
    ```
    func prepare(for segue: UIStoryboardSegue, sender Any?)
    ```
  - Der `sender` ist das Ding, welches den "Segue" ausgelöst hat (d.h. eine Bewegung zu einem anderen MVC)
  - Der `sender` könnte ein `UIButton` oder eine `UITableViewCell` oder ein selbsterstelltes Ding in unserem Code sein
  - Es ist ein Optional, da es absolut ok ist dass ein Segue stattfindet ohne dass ein `sender` spezifiziert ist

- Wo kommt es sonst noch vor?
  - Es könnte verwendet werden um ein Array von Dingen beliebigen Types zu beinhalten (z.B. [AnyObject])
  - Aber in Swift verwenden wir wahrscheinlich stattdessen eher ein `Array` von einem `enum` (wie in `CalculatorBrain`)
  - Also verwenden wir dies nur aus Kompatibilitätsgründen mit einiger Objective-C API

  - Wir könnten es auch nutzen um ein Objekt zurückzugeben, wenn wir nicht wollen dass der Caller dessen Typ kennt
    ```
    var cookie: Any
    ```

Any & AnyObject

- Wie verwenden wir eine Variable vom Typ `Any`?
  - Wir können sie normal nicht direkt verwenden (da wir den tatsächlichen Typ nicht kennen)
  - Stattdessen müssen wir sie in einen anderen, bekannten Typ umwandeln

  - Die Umwandlung in Swift wird mit dem Schlüsselwort `as?` gemacht
  - Die Umwandlung ist vielleicht nicht muoglich, dann wird ein Optional erstellt
  - Wir können auch prüfen ob eine Umwandlung möglich ist, mit dem Schlüsselwort `is` (`true`/`false`)

  - Wir verwenden `as?` meist mit `if let`...

```
let unknown: Any = … // wir können keine Message an unknown schicken, da es keinen Typ hat
if let foo = unknown as? MyType {
   // foo ist hier vom Typ MyType
   // also können wir MyType Methoden ausführen oder auf MyType vars in foo zugreifen
   // wenn unknown nicht vom Typ MyTpe war, dann kommen wir hier nie an
}
```

Casting

- Casting mit `as?` ist nicht nur für `Any` & `AnyObject`
  - Wir können jeden Typ in einen anderen Typ mittels `as?` casten, solange dies Sinn macht
  - Meistens wird es dann verwendet um ein Objekt von einer seiner superclasses zu einer subclass zu casten
  - Aber es könnte auch genutzt werden um einen Typ in ein Protocol zu casten, welches es implementiert (mehr dazu später)

  - Beispiel für Downcasting von einer superclass zu einer subclass…
    ```
    let vc: UIViewController = CalculatorViewController()
    ```
  - Der Typ von `vc` ist `UIViewController` (da wir es so explizit getyped haben)
  - Und die Zuweisung ist valide, da `CalculatorViewController` ein `UIViewController` ist (is a)
  - Aber wir können nun nicht, z.B., `vc.displayValue` nutzen, da `vc` als `UIViewController` getyped ist

  - Wenn wir jedoch vc in einen CalculatorViewController casten, dann können wir es verwenden…
    ```
    if let calcVC = vc as? CalculatorViewController {
       calcVC.displayValue = 3.1415 // dies ist ok
    }
    ```

UserDefaults

- Eine sehr leichtgewichtige und limitierte Datenbank
  - UserDefaults ist grundlegend eine sehr kleine Datenbank welche über App Starts hinweg bestehen bleibt
  - Großartig für Dinge wie “Settings” usw. Verwenden wir niemals für größere Dinge!

- Was können wir darin speichern?
  - Wir sind bzgl. der Speicherung in UserDefaults eingeschränkt: speichert nur Property List Daten
  - Eine Property List ist jede Kombination aus `Array`, `Dictionary`, `String`, `Date`, `Data` oder einer Zahl (`Int`, etc.)
  - Dies isty eine alte Objective-C API mit keinem Typ der all dies repräsentiert, also verwendet diese API `Any`
  - Wenn dies eine neue, Swift-style API wäre, würde es ziemlich sicher nicht `Any` verwenden
    (Wahrscheinlich gäbe es ein Protocol welches diese Typen implementieren würden)

- Wie sieht die API aus?
  - Die Kernfunktion ist einfach. Es werden Property Lists mittels Key gespeichert und abgefragt...

```
func set(Any? forKey: String)        // Any muss eine Property List sein (sonst crash)
func object(forKey: String) -> Any? // es ist garantiert dass Any eine Property List ist
```

UserDefaults

- Lesen und Schreiben
  - Normalerweise ersetllen wir keine dieser Datenbanken mit `UserDefaults()`
  - Stattdessen nutzen wir die `static` (Typ) `var` mit Namen `standard`...

    ```
    let defaults = UserDefaults.standard
    ```

  - Setzen eines Value in der Datenbank ist einfach, da die `set` Methode ein `Any?` annimmt

    ```
    defaults.set(3.1415, forKey: “pi”)          // 3.1415 ist ein Double, d.h. Property List Typ
    defaults.set([1,2,3,4,5], forKey: “My Array”) // Array und Int sind beides Property Lists
    defaults.set(nil, forKey: “Some Setting”)     // entfernt alle Daten für diesen Key
    ```

  - Es kann alles als erster Parameter übergeben werden, solange es ein Property List Typ ist

  - UserDefaults hat auch convenience API zum erhalten von vielen der Property List Typenragt...

    ```
    func double(forKey: String) -> Double
    func array(forKey: String) -> [Any]?              // gibt nil zurück, wenn non-Array bei Key
    func dictionary(forKey: String) -> [String:Any]? // Keys bei Rückgabe sind Strings
    ```

  - Das Any in den zurückgegebenen Values ist natürlich ein Property List Typ

UserDefaults

- Speichern der Datenbank
  - Die Änderungen werden ab und zu gespeichert
  - Wir können ein Speichern aber zu jeder Zeit erzwingen mittels `synchronize`...

    ```
    if !defaults.synchronize() { // failed! Ist aber nicht klar was wir dagegen machen können }
    ```

    (ist nicht “kostenlos” zu synchronisieren, ist aber auch nicht so teuer)

Assertions

- Debug Hilfe
  - Absichtlich unser Programm crashen, wenn eine bestimmte Bedingung nicht erfüllt ist (und eine Meldung ausgeben)
    `assert(() -> Bool, “message”)`
  - Der Funktionsparameter ist jedoch ein “Autoclosure”, daher brauchen wir die `{}` nicht

  - z.B. `assert(validation() != nil, “the validation function returned nil”)`
  - Crashed, wenn `validation() nil` zurückgibt (da wir sicherstellen dass `validation()` dies nicht tut)
  - Der `validation() != nil` Teil kann jeder beliebige Code sein

  - Wenn wir für Release bauen (z.B. für den AppStore), werden `asserts` komplett ignoriert