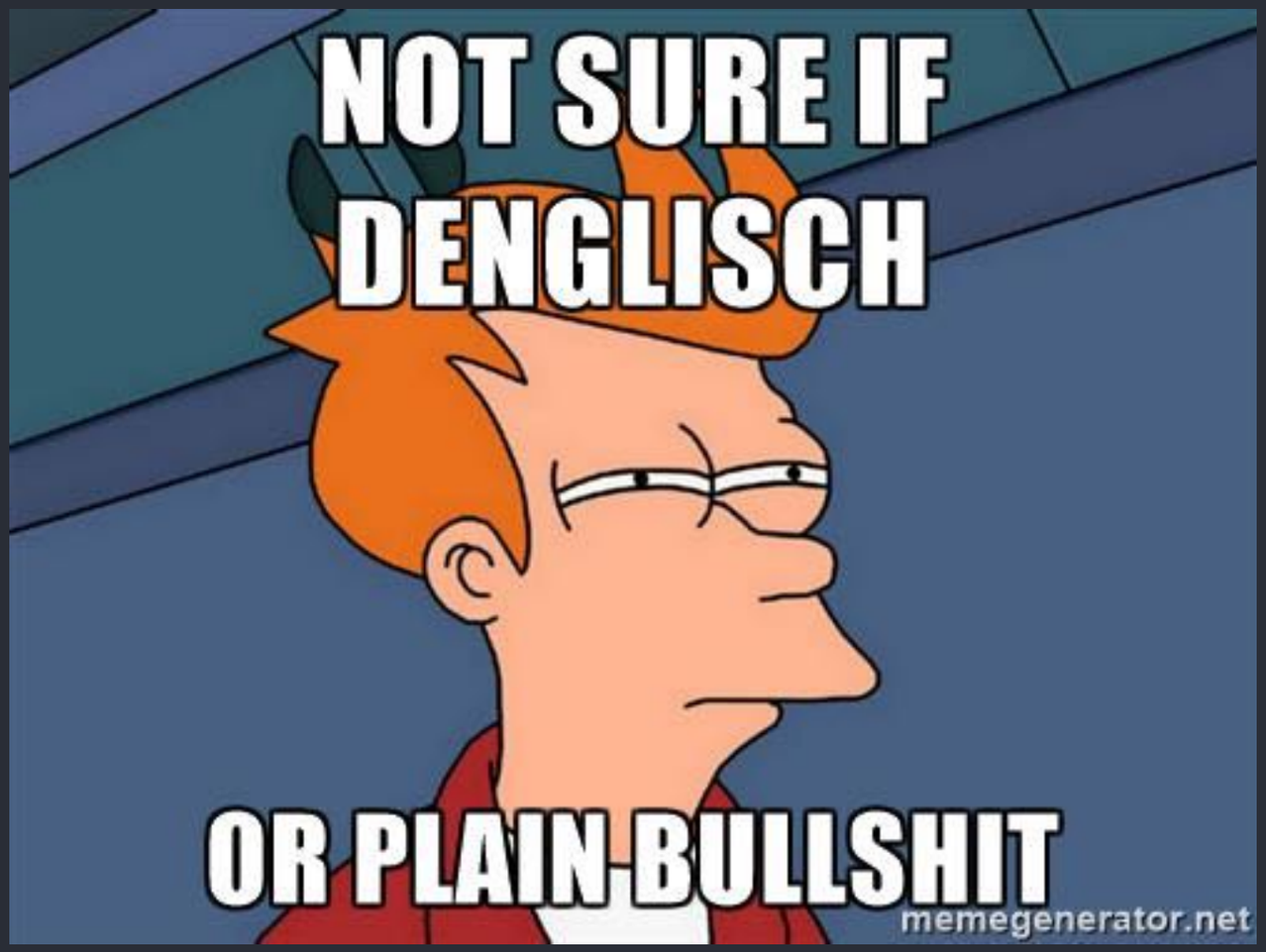
Full Stack iOS Entwicklung mit Swift

WPF im MIM - SS 17

Alexander Dobrynin M.Sc.

Disclaimer



https://cdn.meme.am/instances/500x/63231968.jpg

Heute

Swift und Foundation
Swift Programming Language Guide

Inhalt

- Structs, Klassen und Enums
- Methoden und Properties
- Tuples
- Optionals
- Error Handling
- Generics
- Funktionen
- Range
- Any und AnyObject
- Weitere nützliche Typen
- Collections
- Extensions
- Protocols und Extensions

Structs, Klassen und Enums

- Die wichtigsten Datenstrukturen in Swift (neben Collections)
- Gemeinsamkeiten
  - Deklaration
  - Methoden und Properties
  - Initialisierung
- Unterschiede
  - Semantik
  - Vererbung (class only)
  - Value (struct, enum) und Reference (class) Type

```
// declaration
class MatchingCardGameController { }
struct MatchingCardGame { }
enum Result { }

// properties
let deck: Deck // stored property, not enum!

var pendingCards: [Card] { // computed property
   set {
      pendingCards = newValue
   }
   get {
      return deck.cards.filter { … }
   }
}

var image: Bool { // enum
   switch self {
      case .J, .Q, .K, .A: return true
      case _: return false
   }
}

// methods
mutating func revealCard(at index: Int) -> Result // struct
func cardViewMatching(card: Card) -> CardView // class

// initializers
Card(suit: suit, rank: rank) // struct - memberwise init
init(suit: Suit, rank: Rank) { } // class - explizit init

Result.pendingWith(card) // enum - init with associated type
Rank.A // enum - init with case
Rank(rawValue: “A") // enum - init with rawValue (String)
```

Methoden und Properties

- Methoden
  - Interne und externe Parameternamen
  - Falls nur ein Name angegeben wird, dann ist intern = extern
  - Unterdrücken von externen Namen mit _
  - Idiom: Swift ist eine verbose Sprache, wodurch sich Methoden und Funktionen wie Sätze lesen lassen
- Properties
  - willSet und didSet zum Observieren
  - get und set für Getter und Setter
  - lazy var für Lazy Initialization
- override und final für Vererbung
- open, public, internal, fileprivate, private für Sichtbarkeit

```
// internal and external name
func matchingCardGameScoreDidChange(to score: Int)
.matchingCardGameScoreDidChange(to: 100)

// suppressed external name
func previousCard(_ previous: Card, isMatchingWith other: Card) ->
Bool
previousCard(previous, isMatchingWith: card)

// one name to rule them all
init(numberOfCards: Int)
Deck(numberOfCards: 12)

// sentences all over the place
allCardViews.first(where: { $0.currentTitle == card.desc }
matchedCards.append(contentsOf: [previous, card])
```

```
// property observation
var score: Int = 0 {
   willSet {
      print("setting \(score) to \(newValue)")
   }
   didSet {
      if score != oldValue {
         delegate?.matchingCardGameScoreDidChange(to: score)
      }
   }
}

// property get and set
var score: Int {
   get {
      return UserDefaults.standard.integer(forKey: "score")
   }
   set {
      UserDefaults.standard.set(newValue, forKey: "score")
   }
}

// property read only (get)
var facedUp: Bool {
   return currentTitle != nil
}
```

Tuples

- Gruppierung von mehreren Werten als first class type
- Nichts anderes als unnamed structs
- Häufige Verwendung, wenn mehrere Werte zurückgegeben werden

```
// unnamed tuple
let cardWithScore = (Suit.club, Rank.A, 10)
cardWithScore.0 // ♣ string
cardWithScore.1 // A string
cardWithScore.2 // 10 int

// named tuple
let (suit, rank, score) = (Suit.club, Rank.A, 10)
suit // ♣ string
rank // A string
score // 10 int

// examples
// execute the given `request` asynchronously.`completionHandler` is called afterwards
func dataTask(with request: URLRequest, completionHandler: (Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask

// try to get an image by the given `url` asynchronously. `completion` is called afterwards
func imageBy(url: URL, completion: (UIImage?, URL) -> ()) { … }
```

Optionals

- Ein beliebiger Typ in einem Kontext
- Kontext beschreibt, ob der assoziierte Typ entweder vorhanden (some) oder abwesend (none) ist
- Optional ist ein Enum und first class citizen
- Optional ist ein Monad
- null bzw. nil werden explizit mit einem Typen modelliert
- Unterschiedliche Strategien, um an den Wert innerhalb des Optionals zu kommen
- Konsekutive Optionals können mit Optional Chaining behandelt werden
- Fallbacks Behandlung mit ?? (get or else)

```swift
enum Optional<T> { // generic over any type
    case some(T) // associated type
    case none // nothing, which is mapped to nil
}

// pattern matching, thanks to enum type
switch previousCard {
   case .some(let card):
   case .none:
}

// both are the same
var previousCard: Card?
var prevCard: Optional<Card>.none

// optional inits
let image: UIImage? = UIImage(named: “back”)

// optional checking
if previousCard != nil {
    // .some(previousCard), go ahead
} else {
    // .none, too bad
}

// safe unwrapping
guard let i = cardViews.index(of: sender) else { return }
if let i = cardViews.index(of: sender) { } else { }

// force unwrapping
game!.delegate = self
allCardViews.first(where: { $0.currentTitle == card.description })!

// drawRandomCard() returns Card?
// flatMap removes nested optionals
let cards = (0..<12).flatMap { _ in drawRandomCard() }

// optional chaining, abort when nil occurs
let maybeWinner = scoreLabel?.text?.contains(“100”)

// get or else
let winner = scoreLabel?.text?.contains("100") ?? false
```

Error Handling

- Ähnlich zu Java, können Methoden in Swift Errors werfen
- Ein Error kann mit throws an die nächste Aufrufende Instanz delegiert werden
- Ein sicherer Umgang mit “throwable Methoden” ist es, diese mit einem try Prefix innerhalb eines do-catch-blocks auszuführen
- Ähnlich zu Optionals kann ein try-Aufruf mit try! forciert werden …
- Oder mit try? in ein Optional umgewandelt werden
- Unterschiedliche Errors können mit einem Enum modelliert werden, welches konform zum Error Protokoll ist
- Anschließend kann man im catch-block über den Error pattern matchen

```
// creating a directory can throw, obviously
func createDirectory(at url: URL) throws

// rethrow
init(url: URL) throws {
    try createDirectory(at: url)
}

// try something which can throw and catch the error (safe)
do {
    try createDirectory(at: url)
} catch let error {
    print(error)
}

// force try will crash if it throws an error
try! createDirectory(at: url)

// optional try will return an optional
try? createDirectory(at: url)

// error cases
enum ApiError: Error {
    case emptyData
    case serverError(Error)
}

// given createDirectory throws an ApiError, you can catch
those different error cases
do {
    try createDirectory(at: url)
} catch ApiError.emptyData {
    print(“empty data, sorry”)
} catch let ApiError.serverError(error) {
    print("server error \(request)")
}
```

Generics

- Generics sind ein sehr (sehr) mächtiges Konzept von Programmiersprachen und deshalb auch in Swift vertreten
- Generics sollten nur dann eingesetzt werden, wenn sie einen echten Mehrwert bieten…
- Und nicht um sich selbst zu profilieren
- Structs, Klassen und Enums können mit <T> generisch typisiert werden
- Methoden und Properties können ebenfalls generisch typisiert werden
- Generische Typen können durch Constraints (where Klause) konkretisiert werden
- Dadurch kann man bestimmte Properties oder Methoden des generischen Typen voraussetzen
- Oder den Definitionsbereich der Funktion festlegen

```swift
// generic typed
struct DbRequest<T> {
    let new: [T]
    let deleted: [T]
    let all: [T]
}

enum Result<F, S> {
    case success(value: S)
    case failure(error: F)
}

protocol UniqueEntity {
    var id: Int { get }
}

// T must to be a valid subtype of `UniqueEntity`
func createOrUpdate<T>(entities: [T]) -> DbRequest<T> where T:
UniqueEntity {
   entities.forEach { entity in
        entity.id
   }
}

// given Array<T>, where T is `Element`, `Element` must be a
valid subtype of CardView
extension Array where Element: CardView {
    func cardViewMatching(card: Card) -> CardView { }
}
```

Funktionen

- Wir unterscheiden zwischen Funktionen und Methoden
- Funktionen in Swift sind first class citizen
- Alles, was man mit einer Variable machen kann, kann man auch mit Funktionen machen
- Überall, wo man eine Variable verwenden kann, kann man auch eine Funktion verwenden
- Variablen haben einen Namen und einen Typ
- Funktionen haben auch einen Namen und einen Typ
- Namenlose Funktionen, ƛ, sind gleichzeitig Closures
- Closures erfassen (capture) den Zustand von Variablen und machen ihn innerhalb der Funktion zugänglich, auch wenn die Funktion erst später ausgeführt wird
- Closures können escaping oder non escaping (default) sein
- Foundation und andere iOS Libraries/Frameworks verwenden Closures häufig als completionHandler(:)
- Oder um den Inhalt eines generischen Ablaufes von der aufrufenden Instanz spezialisieren zu lassen
- Dadurch wird ausführbarer Code von der aufrufenden Instanz innerhalb der implementierten Funktion aufgerufen
- Ist ein Closure das letzte Argument, kann die Trailing-Closure Syntax verwendet werden

```swift
// functions as a property
var event: Event?
var resolveCategoryIdToName: ((Int) -> String)?

// assign a function to a property
resolveCategoryIdToName = { id: Int -> String in
   return eventCategories.first(where: { $0.id == id })?.name
)

// you can also assign an existing function to a property
func resolveCategoryName(_ id: Int) -> String {
   return eventCategories.first(where: { $0.id == id })?.name
}

let resolveCategoryIdToName = self.resolveCategoryName

// use the function
let name: String = resolveCategoryIdToName?(event.id)
```

```swift
// one example, where you are able to perform a code block on given actions
let permissionAlert = UIAlertController(
    title: "permission needed",
    message: "\"location - always\" is required in order to continue",
    preferredStyle: .alert
)

let cancelAction = UIAlertAction(
   title: "cancel",
   style: .cancel,
   handler: { action in
    // user tapped “cancel” action, closure is called
    print("user wont give us access to location")
   }
)

let grantAction = UIAlertAction(title: "ok", style: .default) { action in
    // user tapped “ok” action, closure is called
    self.requestLocation()
}

alert.addAction(grantAction)
alert.addAction(cancelAction)

present(permissionAlert, animated: true, completion: { _ in
    // `permissionAlert`, `cancelAction`, `grantAction` and all other
   variables are captured, regardless where defined, thus they can be
   accessed right here
    print(“\(permissionAlert) alert is requested")
})
```

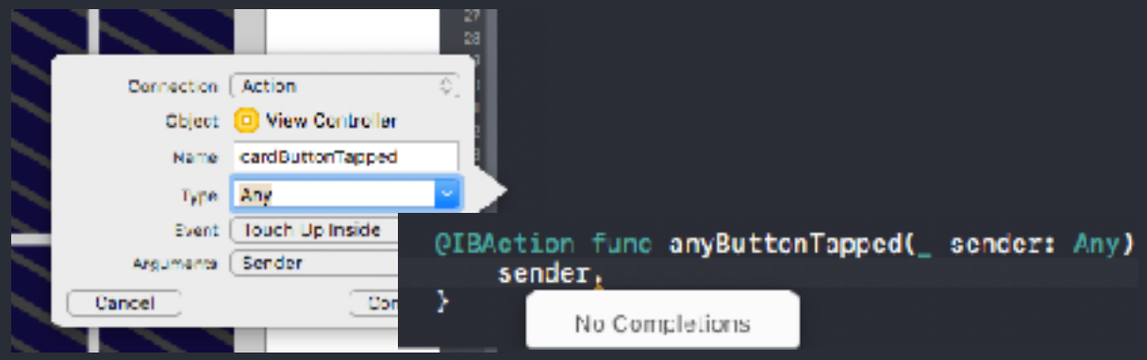
Range

- Range nimmt einen generischen Typen <T> where T: Comparable und baut eine Spanne zwischen lowerBound: T und upperBound: T
- lowerBound muss immer kleiner als upperBound sein, ansonsten gibt es einen Runtime Error
- Über die Spanne kann man anschließend mit high order functions operieren
- Ranges eignen sich um Testdaten zu generieren

```
// give a range of ints …
let numbers: CountableRange<Int> = (0..<100)

// we can apply high order functions
let strings: [String] = numbers.map { i in "\(i)" }
let multipliedByTwo: [Int] = numbers.map { i in i * 2 }
let evenNumbers: [Int] = numbers.filter { i in i % 2 == 0 }
let has10: Bool = numbers.contains(10)
let overlapdsWithRange: Bool = numbers.overlaps((0..<10)

// given following semester dates …
let now = Date()
let semesterStart: Date = …
let semesterEnd: Date = …

// we can check whether we are in the middle of semester
let bool: Bool = (semesterStart..<semesterEnd).contains(now)

// given following group schedules
let groupAStart: Date = …
let groupAEnd: Date = …

let groupBStart: Date = …
let groupBEnd: Date = …

// we can check whether they collide
let collision: Bool =
(groupAStart..<groupAEnd).overlaps((groupBStart..<groupBEnd))

// populate 1000 Cards
let manyCards: [Card] = (0..<1000).map { i in
    Card(suit: Suit.randomOf(i), rank: Rank.randomOf(i))
}
```

Any und AnyObject

- Swift ist (im Gegensatz zu Objective-C) eine stark typisierte Programmiersprache
- Aus Gründen der Kompatibilität zu Objective-C gibt es Any und AnyObject als spezielle Typen



- Auf diese Typen kann man keine Methoden, Funktionen und Properties ausführen
- Man muss sie zunächst in einen konkreten Typen konvertieren (Type-Casting)
- Manchmal kann gibt es dennoch den Anwendungsfall, dass ein Argument tatsächlich Any sein kann

```swift
// first, cast any to a certain type in order to perform actions
@IBAction func anyButtonTapped(_ sender: Any) {
    guard let cardView = sender as? CardView else { return }

    cardView.currentTitle
}

// sometimes, something could really be any
func post(
   name aName: NSNotification.Name,
   object anObject: Any?,
   userInfo aUserInfo: [AnyHashable : Any]? = nil
)

NotificationCenter.default.post(
   name: NSNotification.Name(“ItemsDidChange”),
   object: self, // for now, self (database) is the sender, but it could
   be any
   userInfo: [ // same here, userInfo is just an any dictionary. you can
   post any data you want
    "Updated": updated,
    "Existing": existing
   ]
)

// another example is performing a segue
func performSegue(
   withIdentifier identifier: String,
   sender: Any?
)

performSegue(
   withIdentifier: "ShowSettings",
   sender: self // for now, the vc who is calling this function is sender
)
performSegue(
   withIdentifier: "ShowSettings",
   sender: tableView.indexPathForSelectedRow // or the selected cell
)
performSegue(
   withIdentifier: "ShowSettings",
   sender: "something from the storyboard” // or something in the
   storyboard
)
```

Weitere nützliche Typen

- In Objective-C wurden Typen innerhalb von Modulen (CoreLocation) mit einem identifizierenden Prefix (CL) assoziiert. Der häufigste Prefix ist dabei NS, um sich als Superset von C abzusetzen
- Auch in Swift Projekten tauchen hin und wieder solche Prefix-APIs auf, auch wenn das nicht “swifty” ist. Ist ein entsprechender Typ in Swift verfügbar, beispielsweise NSString -> String, so ist der Swift-Typ zu verwenden

- String, NSLocalizedString, NSAttributedString
- Date, Calendar, DateComponents, Calendar.Component
- DateFormatter, NumberFormatter
- URL, URLRequest, URLSession, Error, NSError
- Void, Data
- CLRegion, CLAuthorizationStatus, UNNotificationRequest, UNNotificationTrigger

Collections

- Swift kennt Array, Dictionary und Set
- Collections sind generisch typisiert, wobei
    - Dictionary<Key, Value> where Key: Hashable und
    - Set<Element> where Element: Hashable sind
- Interessanterweise sind Collections als Structs implementiert
- Zudem implementieren sie unterschiedliche Protokolle (die wiederum Protokolle implementieren), wodurch sie bestimmte Funktionen erhalten, die über alle Collections gleich sind
- Viele dieser Funktionen sind generische high order functions und operieren auf den jeweiligen Collections
- Nahezu alle Funktionen, die high order functions verwenden, sind pure Funktionen
- Das Verwenden solcher Funktionen ist “swifty” und wird ausdrücklich empfohlen, weil
    - sehr kluge Ingenieure die Iterationen um die jeweiligen Collections bereits performant und (hoffentlich) fehlerfrei implementiert haben, wodurch man als Benutzer der API lediglich den eigentlichen Inhalt ausdrücken muss
    - der Code von einem selbst nur das beschreibt, was passiert, und nicht wie es passiert
    - bestimmte Probleme plötzlich lösbar erscheinen, die ansonsten nur äußerst mühsam mit viel boilerplate zu lösen sind
    - die Vorteile von puren Funktionen nicht von der Hand zu weisen sind
    - die high order functions dynamisch zur Laufzeit injiziert werden (Strategy-Pattern) können, da die Funktionen eine definierte Signatur haben

Collections

```
// declaring an array of cards
var cards: Array<Card> = []
var cards: [Card] = [
    Card(suit: Suit.club, rank: Rank.A),
    Card(suit: Suit.club, rank: Rank.Q)
]

// appending/inserting cards
let card = Card(suit: Suit.spade, rank: Rank.A)
cards[0] = card
cards.insert(card, at: 0)
cards.append(card)

// iterating
cards.forEach { (card: Card) in }

// declaring a dictionary of cardViews to cards
var dict: Dictionary<CardView, Card> = [:]
var dict: [CardView : Card] = [
    sender: card
]

// insert a value for a unique key
dict[sender] = card

// iterating
dict.forEach { (key: CardView, value: Card) in }

// return all keys/values as an array
dict.keys
dict.values

// there is a lot more to explore by yourself
```

Collections

```
// given Array with `Element`s inside (simplified code)
struct Array<Element> {

    ///     let cast = ["Vivien", "Marlon", "Kim", "Karl"]
    ///     let shortNames = cast.filter { $0.characters.count < 5 } // ["Kim", "Karl"]
   func filter(_ isIncluded: (Element) -> Bool) -> [Element]

    ///     let allCardButtons: [CardView] = …
    ///     let cardToMatch: CardView = …
    ///
    ///     let first = allCardButtons.first(where: { card in
    ///         card.currentTitle == cardToMatch.description
    ///     })
   func first(where predicate: (Element) -> Bool) -> Element?

   func contains(where predicate: (Element) -> Bool) -> Bool

   func forEach(_ body: (Element) -> Void)

    ///     let cast = ["Vivien", "Marlon", "Kim", "Karl"]
    ///     let lowercaseNames = cast.map { $0.lowercaseString } // ["vivien", "marlon", "kim", "karl"]
    ///     let letterCounts = cast.map { $0.characters.count } // [6, 6, 3, 4]
   func map<T>(_ transform: (Element) -> T) -> [T]

    ///     let numbers = [1, 2, 3, 4]
    ///
    ///     let mapped = numbers.map { Array(count: $0, repeatedValue: $0) } // [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]
    ///
    ///     let flatMapped = numbers.flatMap { Array(count: $0, repeatedValue: $0) } // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
   func flatMap<SegmentOfResult>(_ transform: (Element) -> SegmentOfResult) -> [SegmentOfResult.Iterator.Element] where SegmentOfResult :
   Sequence

    ///     let possibleNumbers = ["1", "2", "three", "///4///", "5"]
    ///
    ///     let mapped: [Int?] = possibleNumbers.map { str in Int(str) } // [1, 2, nil, nil, 5]
    ///
    ///     let flatMapped: [Int] = possibleNumbers.flatMap { str in Int(str) } // [1, 2, 5]
   func flatMap<ElementOfResult>(_ transform: (Element) -> ElementOfResult?) -> [ElementOfResult]

    ///     let numbers = [1, 2, 3, 4]
    ///     let numberSum = numbers.reduce(0, { x, y in
    ///         x + y
    ///     }) // numberSum == 10
   func reduce<Result>(_ initialResult: Result, _ nextPartialResult: (Result, Element) -> Result) -> Result
}
```

Collections

```swift
// we can write our own high order functions
extension Array {

    func groupBy<K : Hashable>(key: (Self.Iterator.Element) -> K) -> [K: [Self.Iterator.Element]] {
        var dict: [K: [Self.Iterator.Element]] = [:]

        forEach { elem in
            let k = key(elem)
            if case nil = dict[k]?.append(elem) { dict[k] = [elem] }
        }

        return dict
    }
}
```

```swift
// one cool thing you can do by combining functions
let atLeastOneMatchWithRanks: Bool = pending.groupBy(key: { card -> Rank in
    return card.rank
}).contains(where: { ranks -> Bool in
    return ranks.value.count >= 2
})

// a more complex example which filter, transform and sort an given array
let transformed: [DateComponents: [Event]] = self.events.filter { event -> Bool in
    guard let date = event.start else { return false }

    return date.isInTheFuture

}.sorted { (l, r) -> Bool in
    return l.start! < r.start!

}.groupBy { event -> DateComponents in
    return calendar.dateComponents([.month, .year], from: event.start!)

}.sorted { (left, right) -> Bool in
    guard let l = calendar.date(from: left.key),
        let r = calendar.date(from: right.key)
    else { return false }

    return l < r
}
```

Extensions

- Extensions erweitern Datenstrukturen wie Structs, Klassen, Enums und Protokolle um Methoden, Funktionen oder Computed-Properties
- Das gilt sogar für Datenstrukturen auf dessen Implementierung man keinen Zugriff hat
- Man kann nichts Überschreiben oder Stored Properties definieren
- Extensions tendieren dazu der goldene Hammer für alle Nägel zu werden. Viele Probleme lassen sich bereits mit guter Objektorientierung lösen
- Zudem lassen sich Extensions nicht “faken” oder “mocken”, da sie unmittelbar an der Implementierung der Datenstruktur hängen
- Ein guter Anwendungsfall für Extensions sind convenience Funktionen/Properties
- Des Weiteren kann man mit Extensions die Sichtbarkeit von Funktionen/Properties steuern

```swift
extension Array {
    func toDictionary<K, V>(key: @escaping (Element) -> K, value: @escaping (Element) -> V) -> [K: V]

    func forAll(predicate: (Element) -> Bool) -> Bool

    mutating func remove(predicate: (Element) -> Bool) -> Element?
}

extension Dictionary {
    func map<K: Hashable, V>(transform: (Key, Value) -> (K, V)) -> Dictionary<K, V>

    func mapKeys<U>(_ transform: (Key) -> U) -> [U: Value]

    func toArray() -> Array<(key: Key, value: Value)>
}

extension Date {
    var isInThePast: Bool { return timeIntervalSinceNow.isLess(than: 0.0) }

    var isInTheFuture: Bool { return !isInThePast }

    var isAtLeastOneDayAgo: Bool { return timeIntervalSince(Date()).isLessThanOrEqualTo(-(60*60*24)) }

    var isAtLeastOneHourAgo: Bool { return timeIntervalSince(Date()).isLessThanOrEqualTo(-(60*60)) }

    var isWithoutTime: Bool {
        let comps = Calendar.current.dateComponents([.hour, .minute], from: self)

        guard let hour = comps.hour,
            let minute = comps.minute
        else { return true }

        return hour == 0 && minute == 0
    }

    func set(hour: Int, minute: Int = 0, second: Int = 0) -> Date {
        return Calendar.current.date(bySettingHour: hour, minute: minute, second: second, of: self)!
    }

    func set(weekday: Int, hour: Int, minute: Int = 0, second: Int = 0) -> Date {
        let date = Calendar.current.date(bySetting: .weekday, value: weekday, of: self)!
        return date.set(hour: hour, minute: minute, second: second)
    }

    func add(day: Int) -> Date {
        return Calendar.current.date(byAdding: .day, value: day, to: self)!
    }

    func add(month: Int) -> Date {
        return Calendar.current.date(byAdding: .month, value: month, to: self)!
    }
}
```

Protocols und Extensions

- Protocol's in Swift sind ähnlich zu Interfaces in Java
- Sie ermöglichen ein lose gekoppeltes API Design
- Ein Protokoll ist ein Typ und erhält somit alle Eigenschaften dessen
- Protokolle können Funktionen, Properties und init-Funktionen definieren und assoziierte Typen enthalten
- Klassen, Structs und Enums können Protokolle implementieren (bzw. zu ihnen "konform" werden)
- Protokolle selbst können konform zu anderen Protokollen sein, wodurch type mixins modellieren werden können
- Zudem sind Default Implementierungen mithilfe von Protocol Extensions möglich
- Des Weiteren kann man mit Protocol Extensions die generischen Typen der Protokolle auf spezifische Typen einschränken, ähnlich zu den generischen Funktionen
- Auf diese Weise kann eine Algebra auf Typenebene für einen bestimmte Definitionsbereich beschrieben und beliebig erweitert werden
- Stichwort Protocol-Oriented Programming

```swift
// a protocol defining a read only property
protocol CustomStringConvertible {
    var description: String { get }
}


extension Card: CustomStringConvertible {
    var description: String {
        return suit.rawValue.appending(rank.rawValue)
    }
}

// a protocol defining a function
protocol MatchingCardGameDelegate {
    func matchingCardGameScoreDidChange(to score: Int)
}
```

```swift
// a protocol defining an init function
protocol JsonConvertable {
    typealias Json = [String: Any]

    init?(json: [String: Any])
}

// a protocol with an associated type
protocol Storage {
    associatedtype Item

    func create(item: Item)
    func get() -> Item
}

struct EventStorage: Storage {
    typealias Item = Event

    func create(item: Event) { }

    func get() -> Event { }
}

struct CardStorage: Storage {
    typealias Item = Card

    func create(item: Card) { }

    func get() -> Card { }
}
```

Protocols und Extensions

```swift
// given the following protocols
enum RequestPolicy {
    case full, network, local, memory
}

protocol ModelProvider {
    func requestModel(requestPolicy: RequestPolicy)
}

protocol Networking {
    func request(category: URLCategory, completion: @escaping (Result<Error, [String: AnyObject]>) -> Void)
}

protocol AbstractCRUDStorage {
    func create<T>(entity: T, withKey key: StorageKey, completion: EntityCompletion<T>?)
    func get(entitiesByKeys keys: [StorageKey], completion: @escaping ([Any?]) -> ())
}

protocol UserDefaultStorage {
    func set(date: Date, forKey key: UserDefaultDateKey)
}

protocol DataRequester {
    associatedtype Model: NSCoding

    var networking: Networking { get }
    var storage: AbstractCRUDStorage { get }
    var defaults: UserDefaultStorage { get }
    var parseJson: (JsonObject) -> [Model] { get }
}

// for those who are conforming to both `ModelProvider` and `DataRequester`, these two functions comes for free. relying on `DataRequester` is crucial in order to access
`networking: Networking`, `storage: AbstractCRUDStorage`, `defaults: UserDefaultStorage` and some kind of json parsing
extension ModelProvider where Self: DataRequester {
    typealias Completion = (Result<Error, [Model]>) -> ()

    func modelByBackgroundUpdate(category: URLCategory, storageKey: StorageKey, defaultsKey: DefaultKey, completion: @escaping Completion) {
        networking.request(category: category) { result in
            let transformed = result.map(self.parseJson).sideEffect { model in
                self.storage.create(entity: model, withKey: storageKey) { _ in
                    self.defaults.set(date: Date(), forKey: defaultsKey)
                }
            }

            completion(transformed)
        }
    }

    func modelByLocalStorage(storageKey: StorageKey, completion: @escaping ([Model]) -> ()) {
        storage.get(entitiesByKeys: [storageKey]) { entities in
            let fromDb = entities.first?.flatMap { $0 as? [Model] } ?? []

            completion(fromDb)
        }
    }
}

struct EventService: ModelProvider, DataRequester { } // you can be this guy
```

Danach

Application- und ViewController-Lifecycle
Segues (Detail, Modal, Popover, Unwind, Embedded)

Demo - CardGames