Beta 2.0

Full Stack iOS Entwicklung mit Swift

WPF im MIM - WS 17/18

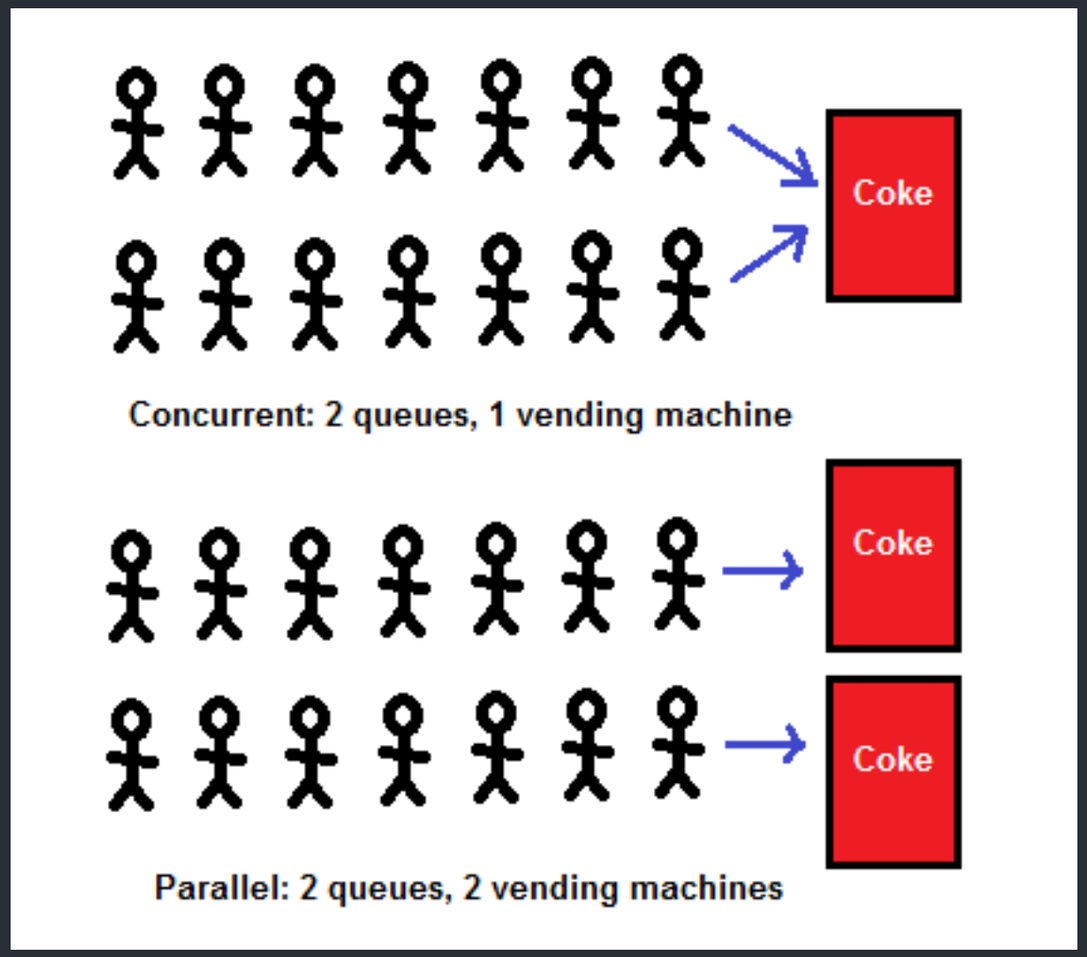
Alexander Dobrynin, M.Sc.

Heute

Multithreading
Networking
Json Parsing

Demo
Assignment

Multithreading



somewhere on the internet

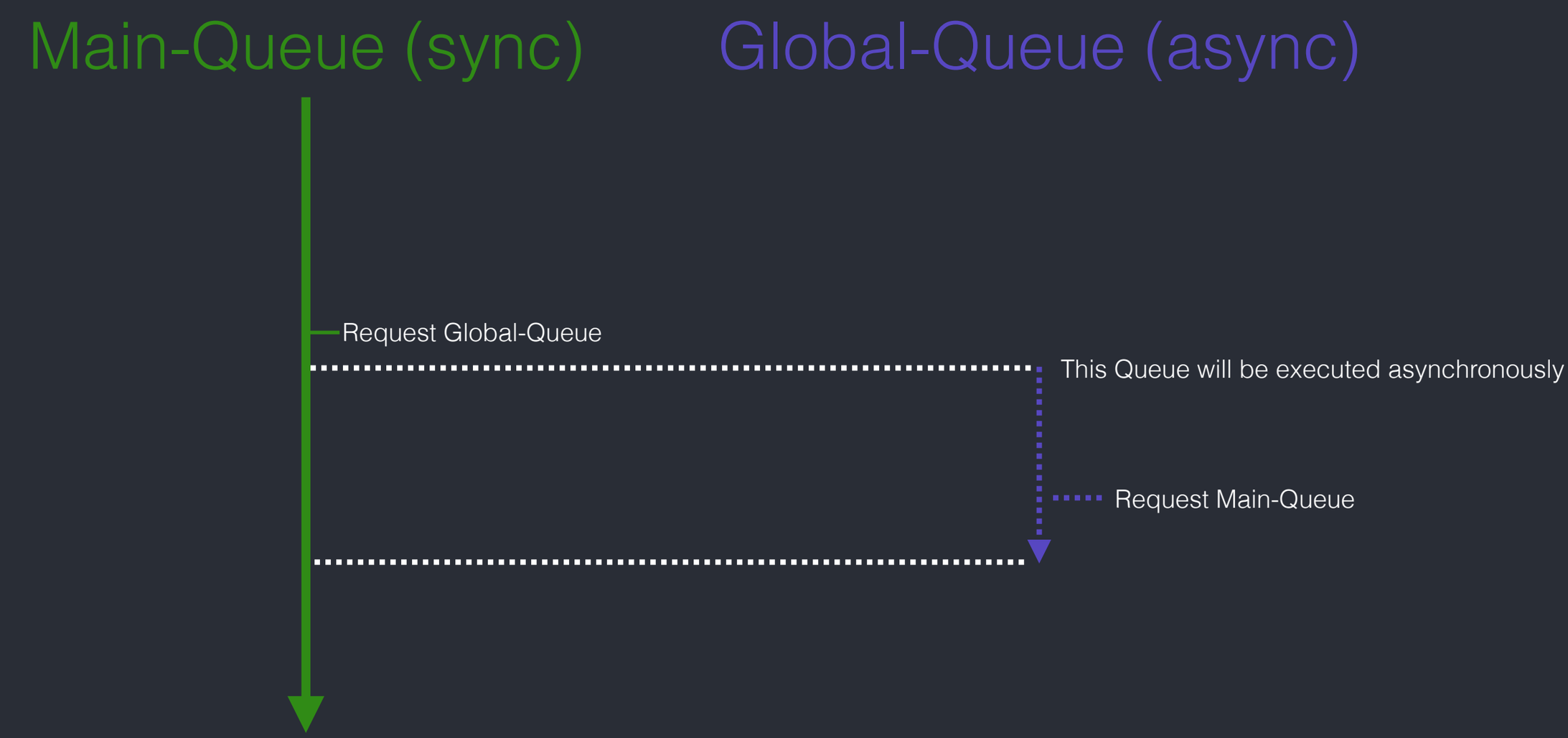
Multithreading

- Mobile Geräte unterliegen eingeschränkten Ressourcen
- Dennoch besteht der Bedarf Daten von Disk zu lesen, aus dem Internet zu laden oder grundsätzlich rechenaufwändige Operationen im Hintergrund durchzuführen
- Mobile Betriebssysteme haben einen Main-UI-Thread, welcher Touch- und sonstige Events verarbeitet und der **einzige Thread ist, welcher mit der UI arbeiten sollte**
- Multithreading in iOS basiert auf Funktionen, die auf unterschiedliche Queues gelegt und anschließend auf bestimmten Threads ausgeführt werden
    - Die einfachste Verwendung von Multithreading in iOS ist Grand-Central-Dispatch (GCD)
    - Eine mächtigere Alternative sind OperationQueues mit Operations. Hier können Abhängigkeiten zwischen Operations modelliert werden, einzelne Operations pausiert oder mit unterschiedlichen Prioritäten versehen werden. OperationQueue abstrahiert über GCD.
- Unabhängig davon existiert eine Main-Queue, auf der die “Hauptarbeit” ausgeführt wird
- Für alle nebenläufigen Aufgaben stehen Global-Queues zur Verfügung
- Queues können entweder seriell oder nebenläufig (concurrent) sein

Multithreading

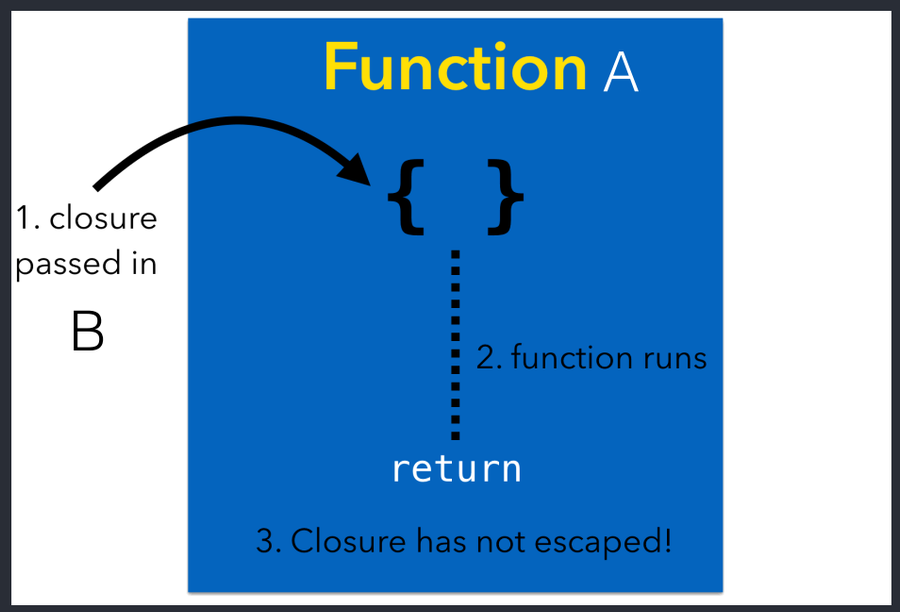
- Unter GCD wird eine Global-Queue mit einem DispatchQoS (Quality of Service) angefragt, welcher die Priorität bestimmt
- DispatchQoS ist ein Enum mit den Cases .userInteractive, .userInitiated, .background und .utility (Priorität absteigend)
- An die angefragte Global-Queue wird eine Funktion übergeben, die gemäß dem QoS in einem Background-Thread ausgeführt wird
- Die Funktion kann entweder sync oder async übergeben werden
    - Bei sync wartet (blockt) die aktuelle Queue solange, bis die Funktion verlassen wird (angefragte Queue hat Funktion synchron abgearbeitet)
    - Bei async wird die Funktion auf die Queue gelegt und sofort verlassen (escaping). Die aktuelle Queue blockiert nicht und “geht sofort weiter”. Erst zu einem späteren Zeitpunkt wird die übergebene Funktion von der angefragten Queue asynchron ausgeführt
- Sobald die Funktionen ausgeführt wurde, kann die Main-Queue angefragt werden, um das Ergebnis auf dem Main-Thread weiter zu verarbeiten
- Der Ablauf mit GCD sollte immer das folgende Pattern haben:
  request global queue -> … ’off main thread’ … -> do something asynchronously ->
  request main queue -> … ’on main thread’ … -> continue with result
- Achtung: für den Entwickler sieht es so aus, als würde alles prozedural ausgeführt werden. Aus diesem Grund basiert GCD (und viele andere asynchrone APIs) auf dem Übergeben und Verschachteln von (escaping) Closures
- Achtung: die Main-Queue sollte niemals mit sync blockiert werden!

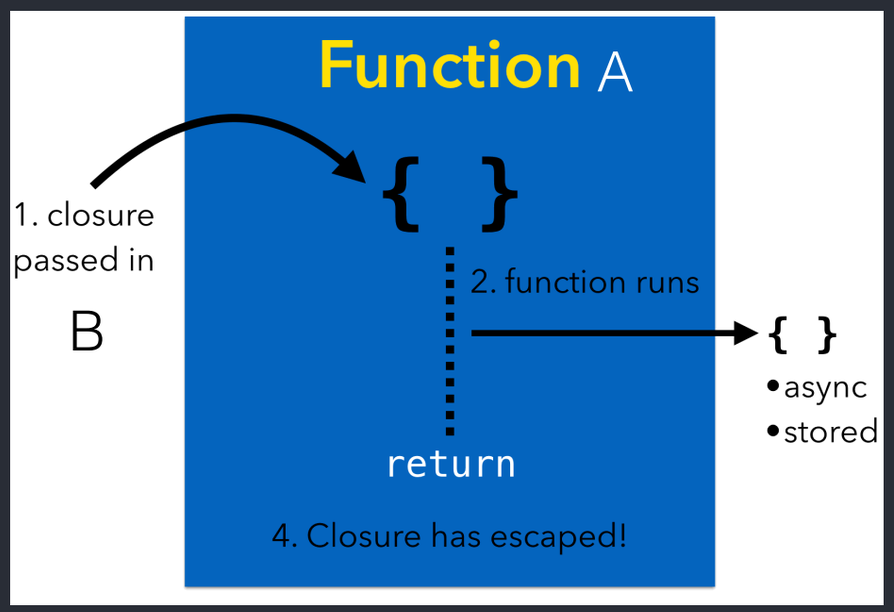
Multithreading



Multithreading

- Wiederholung: Escaping und Nonescaping (default) Closures
- Einer Funktion A wird Funktion B übergeben
    - Nonescaping: Funktion A führt Funktion B unmittelbar, im gleichen Scope wie Funktion A, aus.
      Nachdem Funktion A verlassen wird, wird Funktion B ebenfalls verlassen (vom Stack entfernt).
      Beispiel: alle High-Order Functions von Collections (map, flatMap, foreach, filter, contains, sortedBy, …)
    - Escaping: Funktion A führt Funktion B mittelbar aus. Funktion B wird von Funktion A lediglich angenommen und zwischengespeichert. Es ist nicht garantiert, dass Funktion B gemeinsam mit Funktion A vom Stack entfernt werden, weil Funktion B außerhalb des Scopes von Funktion A existieren kann
      Beispiel: asynchrone Funktionen, die zu einem späteren Zeitpunkt ausgeführt werden
- Escaping Closures tendieren dazu eine Memory-Leak zu verursachen, denn Closures erfassen (capture) alle Variablen!





https://swiftunboxed.com/lang/closures-escaping-noescape-swift3/

Multithreading

```
// request global queue wich is user initiated
DispatchQueue.global(qos: .userInitiated).async {
    // we are off the main thread now. this code block will be executed asynchronously

    // okay, iam done here. ready to go on main thread
    DispatchQueue.main.async {
        // we are back on main thread. lets continue by updating the UI e.g.
    }
}
```

Multithreading

```
print("1")
// request global queue wich is user initiated

DispatchQueue.global(qos: .userInitiated).async {
    // we are off the main thread now. this code block will be executed asynchronously
    print("2")
    // okay, iam done here. ready to go on main thread
    print(“3”)

    DispatchQueue.main.async {
        // we are back on main thread. lets continue by updating the UI e.g.
        print("4")
    }

    print("5")
}

print("6")
```

Print Version 1	Print Version 2
“1”	“1”
“6”	“6”
this could take some time …	this could take some time …
“2”	“2”
“3”	“3”
yep, this too	nope, returning instant to main queue
“5”	“4”
“4”	“5”

Multithreading

```
print(Thread.isMainThread)

DispatchQueue.global(qos: .userInitiated).async {
    print(Thread.isMainThread)

    DispatchQueue.main.async {
        print(Thread.isMainThread)
    }
}

print(Thread.isMainThread)
```

Multithreading

```
print(Thread.isMainThread) // “true”

DispatchQueue.global(qos: .userInitiated).async {
    print(Thread.isMainThread) // “false”

    DispatchQueue.main.async {
        print(Thread.isMainThread) // “true”
    }
}

print(Thread.isMainThread) // “true”
```

Heute

Multithreading
Networking
Json Parsing

Demo
Assignment

Networking

- Grundsätzlich bietet iOS zwei Möglichkeiten, um Daten aus dem Internet zu laden
    1. Data.init(contentsOf: URL)
    2. URLSession.shared.dataTask(with: URL, completionHandler: (Data?, URLResponse?, Error?) -> Void)
- Data.init sollte ausschließlich für File-URLs (z.B. Bilder) verwendet werden
- URLSession hingegen ist ein vollständiger HTTP-Client mit Unterstützung für HTTP-Verben, -Header, -Body, Query-Parameter, Caching, Timeouts, usw.
- In beiden Fällen ist das Ergebnis vom Typ Data?. Ab hier sind Parsen (XML, Json), Type-Casten oder Instantiieren von Objekten mit Data als Init-Parameter, wie z.B. bei UIImage, möglich
- Data.init ist synchron (blockierend) und wird auf der Main-Queue ausgeführt. Dieser Aufruf muss unbedingt auf einer Background-Queue erfolgen!
- Das Ergebnis (completionHandler) von URLSession wird bereits asynchron auf einer Background-Queue ausgeführt. Möchte man das Ergebnis im UI darstellen, so muss man die Main-Queue anfordern
- Achtung: Seit iOS 9 hat Apple ein Feature Namens App Transport Security eingeführt, wodurch non-HTTPS Requests nicht mehr erlaubt sind. Als Developer kann man entweder bestimmte Domains / Subdomains explizit erlauben (empfohlen) oder jegliche Anfragen (Arbitrary Loads) erlauben

Networking

```
func imageSync(by url: URL) {
    let data = try? Data(contentsOf: url) // bad! this line is blocking current (main) queue
    let img = data.flatMap(UIImage.init) // UIImage has an initializer which tries to build an image out of data

    imageView?.image = img
}

func imageAsync(by url: URL) {
    DispatchQueue.global(qos: .userInitiated).async { // request global queue and run asynchronously
        let data = try? Data(contentsOf: url) // this line will now block another queue than main queue
        let image = data.flatMap(UIImage.init)

        DispatchQueue.main.async { // finished async task... request main queue and show result
            self.imageView?.image = image // again: 'imageView' is UI code. thus we need main queue
        }
    }
}
```

- Wenn man durch eine Liste von Bildern scrollt, kann es vorkommen, dass jüngst heruntergeladene Bilder nicht mehr im UI dargestellt werden müssen, weil der User bswp. viel weiter ist oder neue Daten anfragt
- Deshalb kann es sinnvoll sein, self als schwache Referenz (weak) in die Closure zu übergeben, sodass self sich deallozieren kann, wenn self bswp. nicht mehr sichtbar ist, wodurch die Zuweisung der ImageView nicht mehr erfolgt
- Zudem kann es sinnvoll sein die angefragte URL mit der aktuellen URL zu vergleichen, falls der User zwischenzeitlich eine neue URL angefragt hat. Ansonsten wird “das alte Bild” angezeigt während das neue lädt

Networking

```
func requestGET(url: URL) {
    var request = URLRequest(url: url, cachePolicy: .reloadIgnoringCacheData, timeoutInterval: 10)
    request.httpMethod = "GET"

    let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
        // off main thread, managed by URLSession

        let mapped = data?.map {
            // do something with data … asynchronously
        }

        DispatchQueue.main.async { // super important: request main queue in order to update UI
            self.label.text = mapped?.description
        }
    }

    // dataTask(with:) creates a task and returns immediately …

    task.resume() // thus we have to resume (aka. start) the task afterwards
}
```

Networking

```
func requestPOST(url: URL, body: [String: Any]) {
    var request = URLRequest(url: url, cachePolicy: .reloadIgnoringCacheData, timeoutInterval: 10)
    request.httpMethod = "POST"
    request.addValue("application/json", forHTTPHeaderField: "Content-Type")
    request.httpBody = try? JSONSerialization.data(withJSONObject: body, options: [])

    let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
        // off main thread, managed by URLSession

        // if we don't do anything on the main queue anyway, we don't need it either
    }

    task.resume() // still important
}
```

Heute

Multithreading
Networking
Json Parsing

Demo
Assignment

Json Parsing

- Im Wesentlichen gibt es zwei Möglichkeiten, um Json aus Data? zu parsen und in ein eigenes Model zu konvertieren

    1. Data? zu Any (Json) überführen, danach zu einem Dictionary<String, Any> casten, iterieren und jedes einzelne Feld über die Subscript Schreibweise (dict[“key”]) ansprechen, um anschließend einen Typen zu bauen

    2. Einen Typen (Struct, Klasse, Enum) konform zu Codable (Encodable & Decodable) machen Neu in Swift 4

- Bei der ersten Lösung hat man am meisten Kontrolle, weil man selbst jedes einzelne Feld einließt und entscheidet, wie die Felder zu interpretieren sind. Jedes Feld kann validiert werden, bevor es in ein Model überführt wird

- Bei der zweiten (empfohlenen) Lösung gibt es 3 Ebenen der Anpassbarkeit

    1. 1-zu-1 Abbildung von Model zu Json. Matching von Variablennamen und Typen

    2. Unterschiedliche Variablennamen zwischen Model und Json inkl. der Auswahl von Json-Felder (CodingKey)

    3. Überschreiben der Serialisierung (func encode(to encoder:)) und/oder Deserialisierung (init(from decoder:))

- Da das automatische Parsen auf Typenebene implementiert ist, kann man nicht nur Basis-Datentypen, sondern auch eigene Typen verschachteln, solange sie ebenfalls Codable sind

- Zudem bekommt man bei Codable sowohl Encodable (Model -> Json) als auch Decodable (Json -> Model) geschenkt. Das manuelle Parsen ist lediglich der Weg von Json -> Model

- Bonus: in jedem Falle ist Data? die Serialisierung, welche ein legitimes Format (sog. Property List) für die Persistenz mit UserDefaults oder dem Schreiben auf Disk ist

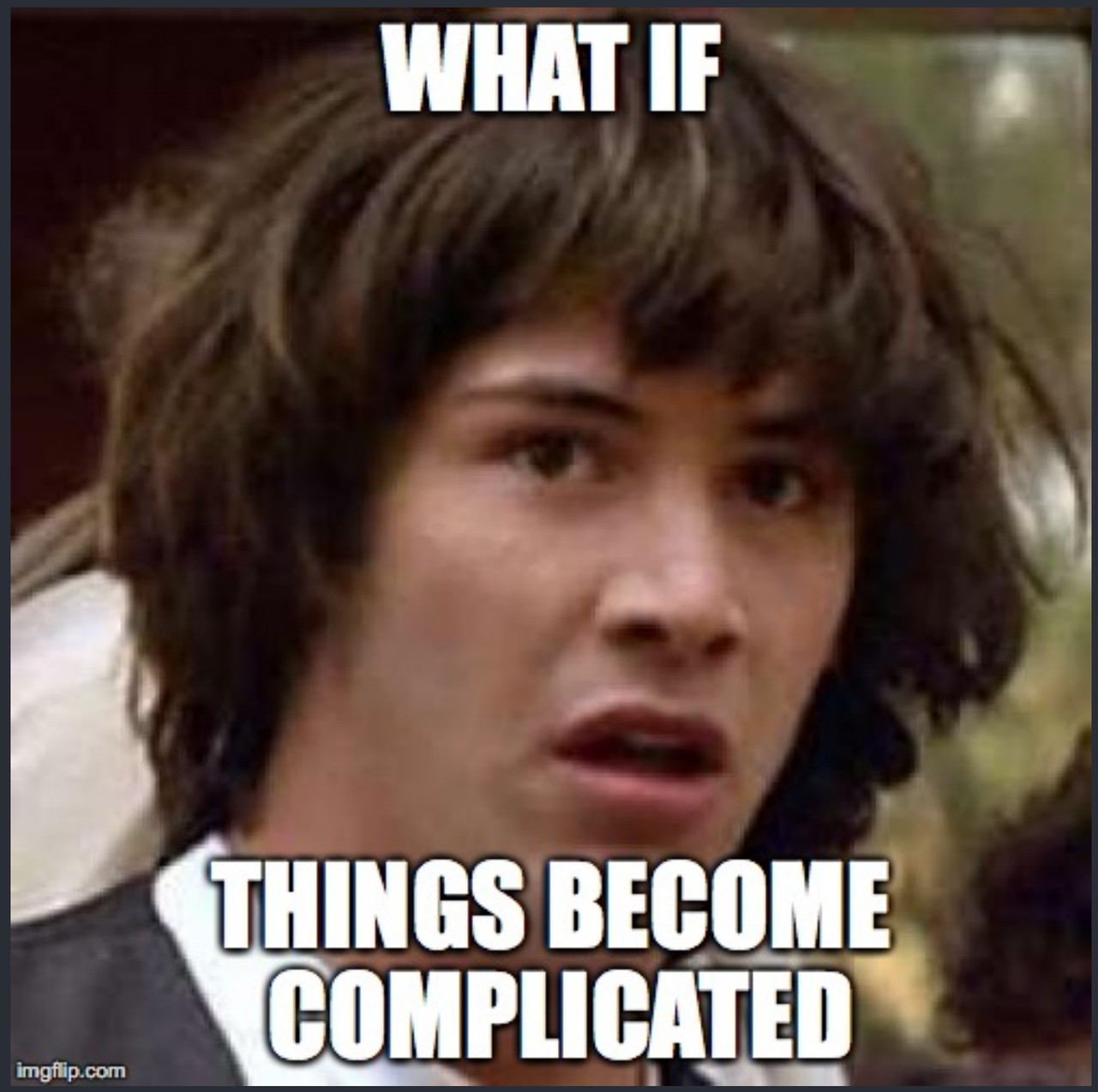
Json Parsing

```swift
struct Person {
    let name: String
    let birthDay: Date
}

/* JSON to parse
 [
    {
       "name": "alex",
       "birthDay": 634172400
    },
    {
       "name": "oskar",
       "birthDay": 1431986400
    },
    {
       "name": "pjotr",
       "birthDay": 0
    }
]
*/
```

Json Parsing

```
typealias JsonArray = [[String: AnyObject]] // nice little trick

fileprivate static func parseJson(_ data: Data) -> [Person] { // manuel parsing
    let json = try? JSONSerialization.jsonObject(with: data, options: .allowFragments) as? JsonArray // cast

    let persons: [Person]? = json??.flatMap { json -> Person? in // iterate through dict and try to build model
        guard let name = json["name"] as? String,
            let seconds = json[“birthDay"] as? Int,
            let birthDay = … /* seconds since 1970 to Date */
        else { return nil }

        return Person(name: name, birthDay: birthDay)
    }

    return persons ?? []
}


struct Person: Codable { … } // make your model conform to ‘Codable’

fileprivate static func parseJson(_ data: Data) -> [Person] {
    let decoder = JSONDecoder()
    decoder.dateDecodingStrategy = .secondsSince1970 // .iso8601, .formatted(dateFormatter), and much more

    let persons = try? decoder.decode([Person].self, from: data) // automatic conversion, error otherwise

    return persons ?? []
}
```

Json Parsing



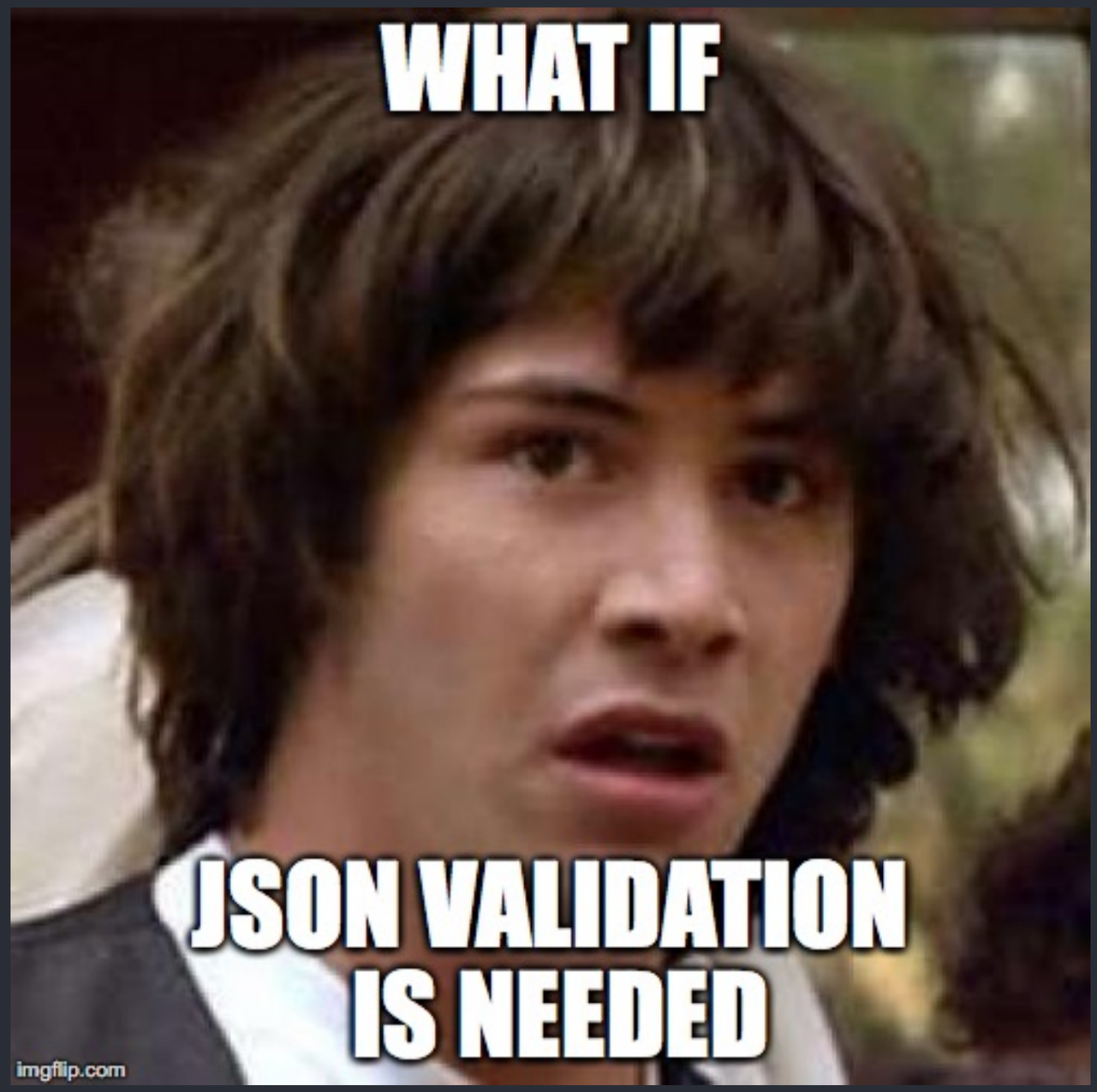
https://imgflip.com/memegenerator/Conspiracy-Keanu

Json Parsing

```
/* JSON with nested objects
 [
     {
        "name": "alex",
        “birthDay": 634172400,
         "address": {
            "street": "steinmuellerallee 1”,
            “zip_code": “51643”,
            "city": "gummersbach"
         }
     },
     { ... },
     { ... }
 ]
 */

struct Address: Codable { // conform to ‘Codable’
    let street: String
    let city: String
    let zip_code: String
}

struct Person: Codable {
    let name: String
    let birthDay: Date
    let address: Address // nested type
}
```

```
/* JSON with different attribute names
 [
     {
         ...
         "address": {
            "street": "steinmuellerallee 1”,
            “zip_code": “51643”,
            "city": "gummersbach"
         }
     },
     { ... },
     { ... }
 ]
 */

struct Address: Codable {
    let city: String; let zipCode: String

    enum CodingKeys: String, CodingKey {
        /* case street */ // exclude street
        case zipCode = "zip_code"
        case city
    }
}

struct Person: Codable {
    /* ... */ let address: Address
}
```

Json Parsing



https://imgflip.com/memegenerator/Conspiracy-Keanu

Json Parsing

```
struct Address: Codable {
    let city: String; let zipCode: Int // zipCode is now of type int

    enum CodingKeys: String, CodingKey { /* like from before */ }
}

extension Address {

    enum AddressCodableError: Error { // custom error
        case invalidCast(of: String)
    }

    init(from decoder: Decoder) throws { // override serialization
        let container = try decoder.container(keyedBy: CodingKeys.self)
        let city = try container.decode(String.self, forKey: .city)
        let zip = try container.decode(String.self, forKey: .zipCode)

        if let zipCode = Int(zip) { // cast zipCode from String to Int
            self.init(city: city, zipCode: zipCode)
        } else {
            throw AddressCodableError.invalidCast(of: zip) // throw otherwise
        }
    }

    func encode(to encoder: Encoder) throws { // override deserialization
        var container = encoder.container(keyedBy: CodingKeys.self) // ‘container’ has to be mutable!
        try container.encode(city.lowercased(), forKey: .city) // lowercased also
        try container.encode(zipCode.description, forKey: .zipCode) // encode string
    }
}
```

Json Parsing

```swift
let address = Address(city: "Gummersbach", zipCode: 51643)

do {
    let json: Data = try JSONEncoder().encode(address)
    let jsonString: String? = String(data: json, encoding: .utf8) // json is usually utf-8 encoded
    print("encoded json", jsonString)

    let addrrs: Address = try JSONDecoder().decode(Address.self, from: json)
    print("decoded address", addrrs)
} catch {
    print(error.localizedDescription)
}
```

```
encoded json Optional("{\"zip_code\":\"51643\",\"city\":\"gummersbach\"}")
decoded address Address(city: "gummersbach", zipCode: 51643)
```

Heute

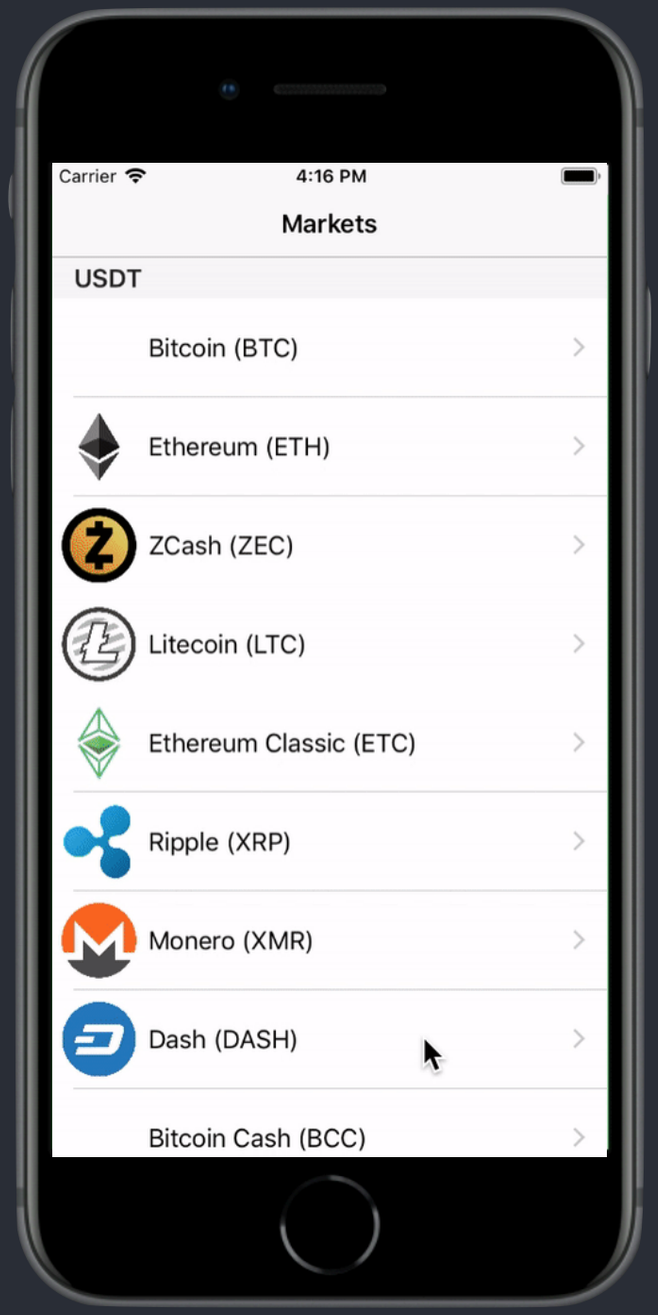
Multithreading
Networking
Json Parsing

Demo
Assignment

Cryptomarket - Demo

- Die App zeigt Crypto-Currency Märkte an
- Beim Tap auf ein Market, werden die letzten Kurse angezeigt
- Aktuell sind alle Daten hartkodiert. Diese sollen nun von einem RESTful-Webservice bezogen werden
- Recap: MVC, Segues, TableViewController

- GCD, Image Downloading, (escaping) Closures
- URLRequest, URLSession, URLSessionDataTask
- Codable, CodingKeys, JSONDecoder
- Generics, Algebra, API-Design, Encapsulation, Service-Klassen
- Optional: NSCache und Singletons



Heute

Multithreading
Networking
Json Parsing

Demo
Assignment

Cryptomarket - Assignment

- MarketSummary soll über den Webservice angefragt werden
  - Ressource ist /getmarketsummary?market=$MARKET_NAME
  - Achtung: es wird vermutlich folgender Fehler auftauchen:

    "The data couldn't be read because it isn't in the correct format."

    Dem JSONDecoder muss man einen DateFormatter übergeben, damit er Dates korrekt parsen kann. Nehmt meinen JSONDecoder hierfür.

    - Passt den MarketSummaryService an (ähnlich zum MarketService)
    - Benutzt zwischendurch debugPrints(#file, #function, $VAR), um eure Ergebnisse zu verifizieren
- Optional: das JSON von MarketSummary gibt immer nur den letzten Kurs zurück. Um eine bessere UX zu schaffen, kann man die vorherigen MarketSummaries in einem globalen Array sammeln und das neuste jedes mal anhängen (ähnlich zu Tags). Richtige Persistenz gibt es beim nächsten Termin!
- Sonstige Änderungen und Verbesserungen sind Willkommen
- Bis zum 16.01.18, 13:59 Uhr per Pull-Request einreichen