

Full Stack iOS Entwicklung mit Swift

WPF FSIOS
Alexander Dobrynin, M.Sc.

Heute

Funktionen und Closures

Multithreading

CollectionView

Zusammenfassung

Funktionen und Closures

- In Swift wird zwischen Funktionen und Methoden unterschieden
- 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

Funktionen und Closures

```
// functions as a property
var number: Int?
var toString: (Int) -> String

// treat functions as first class citizen
struct Inty {
    let int: Int

    init(int: Int) { self.int = int }

    init(buildFrom: () -> Int?) { self.int = buildFrom() ?? -1 }
}
```

Funktionen und Closures

- 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
- So wie **Int**, **String** und **Bool** Literale haben, gibt es auch Funktionsliterale, sog. **λ oder Closures**
 - Funktionsliterale werden über geschweifte Klammern definiert und als solches übergeben
 - Innerhalb der Klammern wird der Typ der Funktion mit dem Pattern “Parameter -> ReturnType **in** ...” verwendet
 - Da Swift eine stark typisierte Sprache ist, wird die Signatur der Funktion oftmals inferiert, weshalb diese nicht explizit typisiert werden muss
 - Zudem können die Parameternamen mit \$0, \$1, ... \$x generalisiert werden oder mit **_** unterdrückt werden

Funktionen und Closures

```
// functions as a property
var number: Int?
var toString: (Int) -> String

// treat functions as first class citizen
struct Inty {
    let int: Int

    init(int: Int) { self.int = int }

    init(buildFrom: () -> Int?) { self.int = buildFrom() ?? -1 }
}

// assign function to property
number = 5
toString = { (int: Int) -> String in // full signature
    return int.description
}
toString = { int in // inferred signature
    return int.description
}
toString = { $0.description } // anonymous arguments

// call functions
number // returns 5 as an int
toString(5) // returns "5" as a string

// assign existing functions to properties
func objectOrientedToString(int: Int) -> String {
    return int.description
}

toString = objectOrientedToString // treat them as the same, because both signatures are matching
```

Funktionen und Closures

- 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
- So wie **Int**, **String** und **Bool** Literale haben, gibt es auch Funktionsliterale, sog. **λ oder Closures**
 - Funktionsliterale werden über geschweifte Klammern definiert und als solches übergeben
 - Innerhalb der Klammern wird der Typ der Funktion mit dem Pattern “Parameter -> ReturnType **in** ...” verwendet
 - Da Swift eine stark typisierte Sprache ist, wird die Signatur der Funktion oftmals inferiert, weshalb diese nicht explizit typisiert werden muss
 - Zudem können die Parameternamen mit \$0, \$1, ... \$x generalisiert werden oder mit **_** unterdrückt werden
- Funktionen **können selbst** Funktionen zurückgeben oder Funktionen entgegennehmen
- Closures erfassen (capture) den Zustand von Variablen und machen ihn innerhalb der Funktion zugänglich, auch wenn die Funktion erst später ausgeführt wird

Funktionen und Closures

```
var toString: (Int) -> String
toString = { $0.description } // anonymous arguments

// function which takes a function (high order function)
func randomNumbertoStringPlease(f: (Int) -> String) -> String { // takes function `f`
    let randomNumber = Int(arc4random()) // e.g. 3089556946
    let stringVersion = f(randomNumber) // use function `f`. at this point we have no idea how 'f' is implemented, and we
don't have to
    return stringVersion
}

randomNumbertoStringPlease(f: { int in // implementation of 'f' is defined here
    return int.description
})

randomNumbertoStringPlease { $0.description } // trailing closure syntax
randomNumbertoStringPlease(f: toString) // passing another functions which matches signature

// functions is capturing variable
let upperBound = UInt32(100)

func capturingrandomNumbertoStringPlease(f: (Int) -> String) -> String { // `upperBound` is available here
    let randomNumber = Int(arc4random_uniform(upperBound)) // e.g. 50
    let stringVersion = f(randomNumber)
    return stringVersion
}

capturingrandomNumbertoStringPlease(f: toString)
```

Funktionen und Closures

```
var toString: (Int) -> String
toString = { $0.description } // anonymous arguments

// function which takes a function (high order function)
func randomNumbertoStringPlease(f: (Int) -> String) -> String { // takes function `f`
    let randomNumber = Int(arc4random()) // e.g. 3089556946
    let stringVersion = f(randomNumber) // use function `f`. at this point we have no idea how 'f' is implemented, and we
    don't have to
    return stringVersion
}

randomNumbertoStringPlease(f: { int in // implementation of 'f' is defined here
    return int.description
})

randomNumbertoStringPlease { $0.description } // trailing closure syntax
randomNumbertoStringPlease(f: toString) // passing another functions which matches signature

// functions is capturing variable
let upperBound = UInt32(100)

func capturingrandomNumbertoStringPlease(f: (Int) -> String) -> String { // `upperBound` is available here
    let randomNumber = Int(arc4random_uniform(upperBound)) // e.g. 50
    let stringVersion = f(randomNumber)
    return stringVersion
}

capturingrandomNumbertoStringPlease(f: toString)
```

Funktionen und Closures

```
// function which returns a function

struct Student { let name: String } // given this
struct Certificate { let date: Date; let student: Student } // and that

// 'certificates' takes 'Date' and returns a function which takes 'Student' and returns 'Certificate'
let certificates: (Date) -> (Student) -> Certificate = { date in
    return { student in Certificate(date: date, student: student) }
}

let now = Date() // of type `Date`
let grantCertificate = certificates(now) // of type `(Student) -> Certificate`

["A", "B", "C"] // Array<String>
    .map { s in Student(name: s) } // Array<String> -> Array<Student>
    .map { s in grantCertificate(s) } // Array<Student> -> Array<Certificate>
    .forEach { c in print(c) } // prints

// make a print function which matches the signature of `(Certificate) -> ()`
let customPrint = { (c: Certificate) in print(c) }

// inline functions which matches signature
["A", "B", "C"].map(Student.init).map(grantCertificate).forEach(customPrint)
```

Funktionen und Closures

- Closures werden häufig verwendet, um **zu beschreiben** was passieren soll, **wenn** etwas soweit ist
 - Demnach können Closures erst nach einer bestimmten Zeit aufgerufen werden, obwohl sie unmittelbar übergeben werden
 - Es kann sogar passieren, dass die aufgerufene Funktion, welche die Closure entgegennimmt, den Funktions-Stack schon lange verlassen hat, wenn die Closure aufgerufen wird
 - Solche Closures müssen als **escaping** deklariert werden. Der Default ist **non escaping**
- 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
 - Zudem kann der Funktionsblock einer Closure Properties mehrzeilig initialisieren. Häufig in Verwendung mit **Lazy-Initialisation**

Funktionen und Closures

```
// closures as completion handlers
let label = UILabel()

UIView.animate(withDuration: 0.1, animations: {
    label.alpha = 0.0 // label is captured here by the way
}, completion: { finished in // called when animation is finished
    label.removeFromSuperview()
})

func superHeavyCalculation(completion: () -> ()) {
    print("starting super heavy calculation")
    // calc ...
    completion()
}

superHeavyCalculation { // again, trailing closure syntax
    print("seems to be finished")
}

// lazy init blocks

let formatter = DateFormatter()
formatter.dateStyle = .short
formatter.timeStyle = .short

let formatter2: DateFormatter = {
    let f = DateFormatter()
    f.dateStyle = .short
    f.timeStyle = .short
    return f
}()
```

Heute

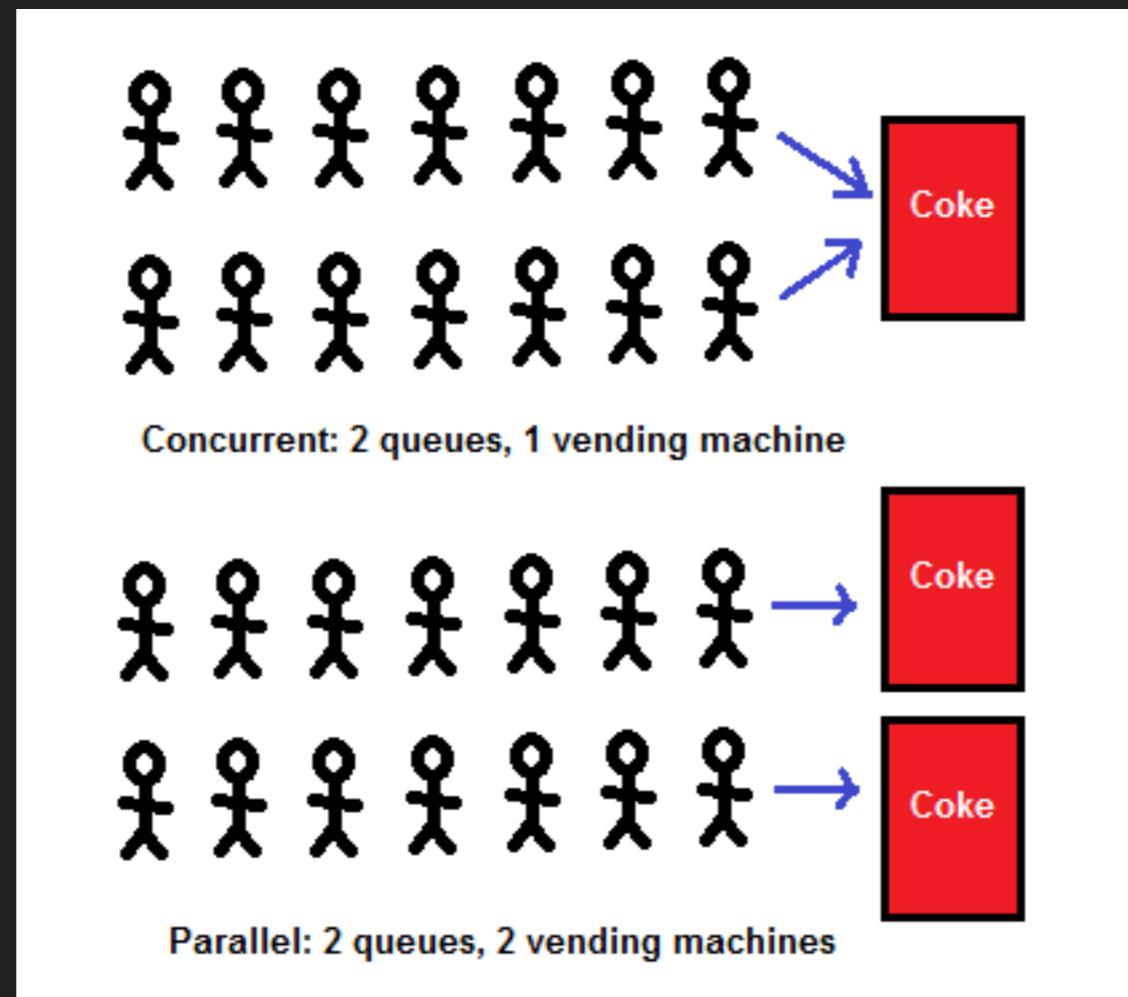
Funktionen und Closures

Multithreading

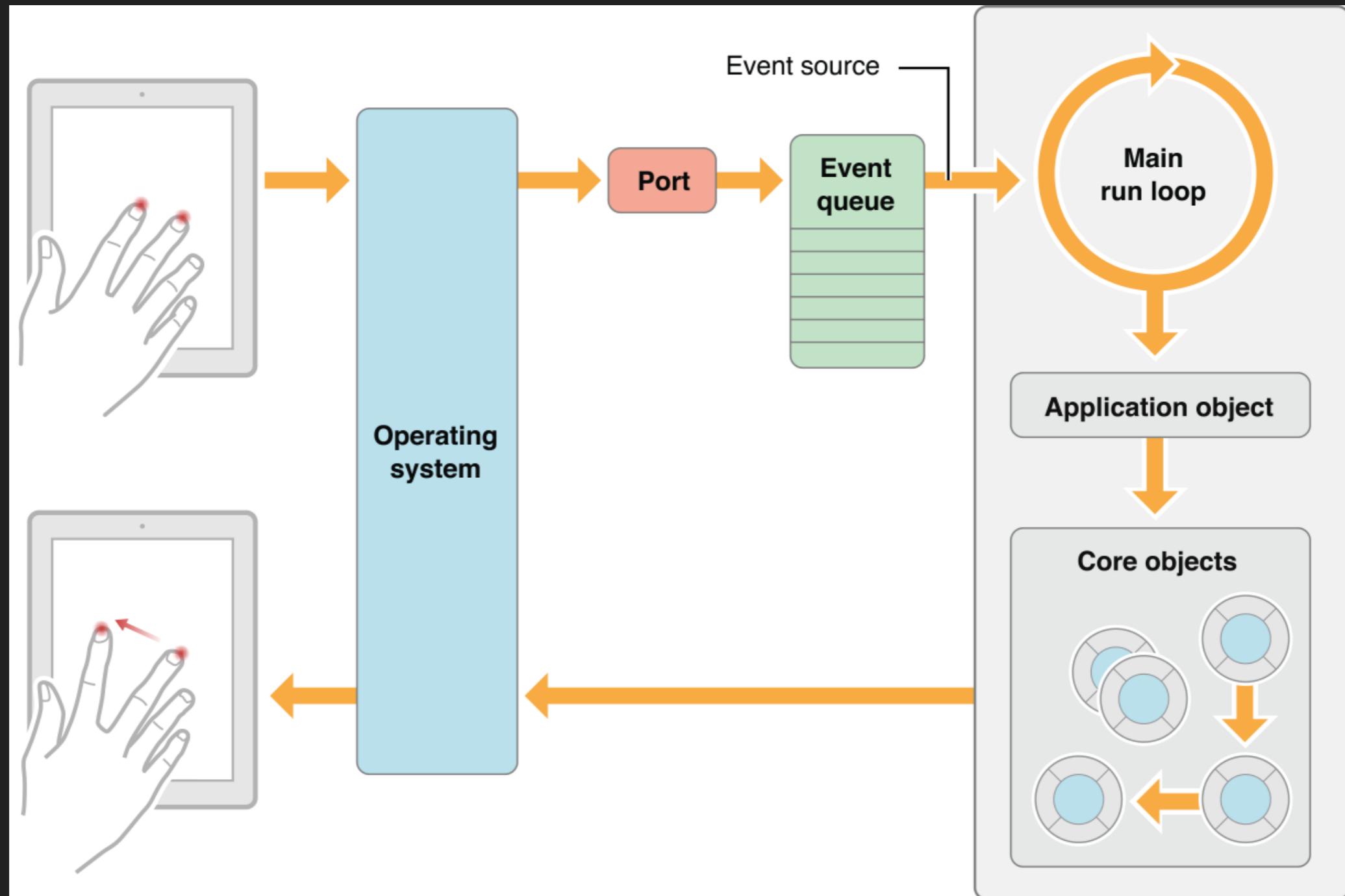
CollectionView

Zusammenfassung

Multithreading



Multithreading



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 **einziges 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 die 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
 - **Achtung:** die Main-Queue sollte niemals mit `sync` blockiert werden!
- Sobald die Funktionen ausgeführt wurde, kann die Main-Queue angefragt werden, um das Ergebnis auf dem Main-Thread weiter zu verarbeiten
- Die Klasse `Thread` kann im Kontext von GCD nützlich sein, z.B. die Property `Thread.isMainThread`
- **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

Multithreading

Main-Queue (sync)

Global-Queue (async)



Multithreading

```
let globalQueue = DispatchQueue.global(qos: .userInitiated) // request global queue which is user initiated (high priority)

globalQueue.async { // execute this closure asynchronously on `globalQueue` when possible

    // we are off the main thread now. this code block will be executed asynchronously

    // long running task ...

    // okay, i'm done here. ready to go on main thread

    let mainQueue = DispatchQueue.main // request (the one and only) main queue

    mainQueue.async { // execute this closure asynchronously on `mainQueue` when possible

        // we are back on main thread now. use your result here to update the UI
    }
}
```

Multithreading

```
print("1")

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

    DispatchQueue.main.async {
        // we are back on main thread now. use your result here to update the UI
        print("4")
    }

    print("5")
}

print("6")
```

Print Version 1

```
"1"
"6"
this could take some time ...
"2"
"3"
yep, this too
"5"
"4"
```

Print Version 2

```
"1"
"6"
this could take some time ...
"2"
"3"
nope, returning instant to main queue
"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

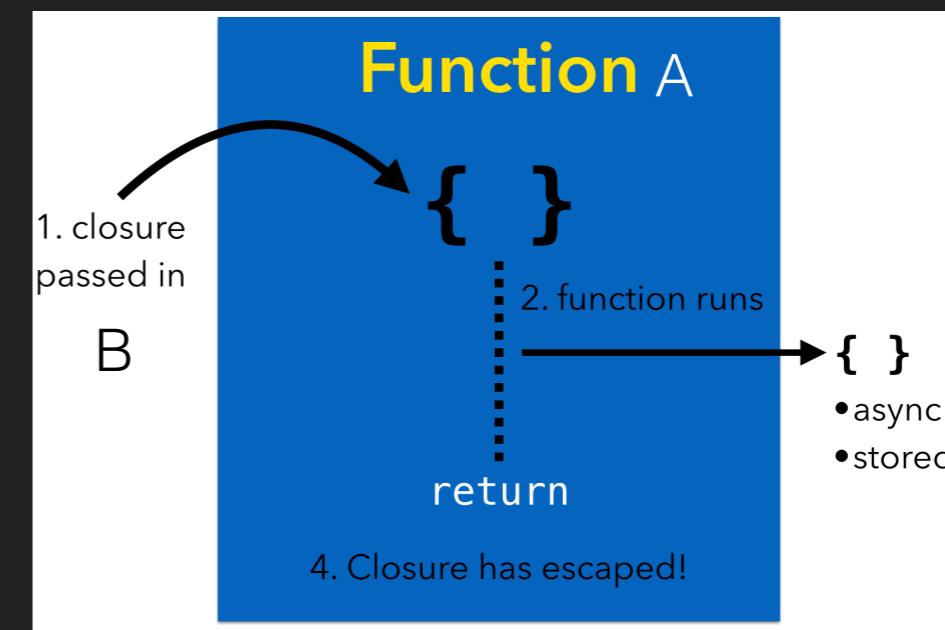
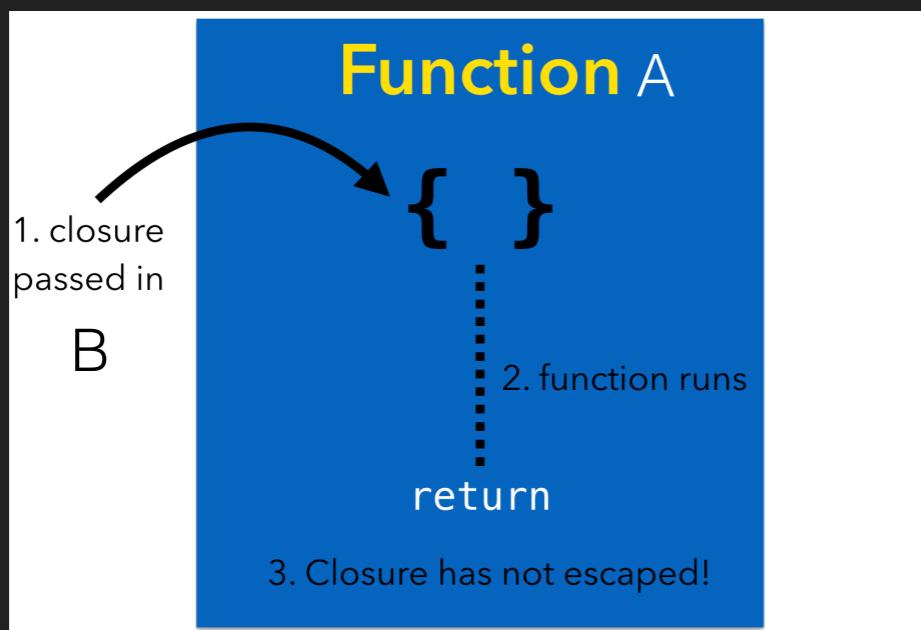
DispatchQueue.global(qos: .userInitiated).async {
    print(Thread.isMainThread) // false

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

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

Multithreading

- Wiederholung: Escaping und Non-escaping (default) Closures
- Einer Funktion A wird Funktion B übergeben
 - Non-Escaping: 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/>

Heute

Funktionen und Closures

Multithreading

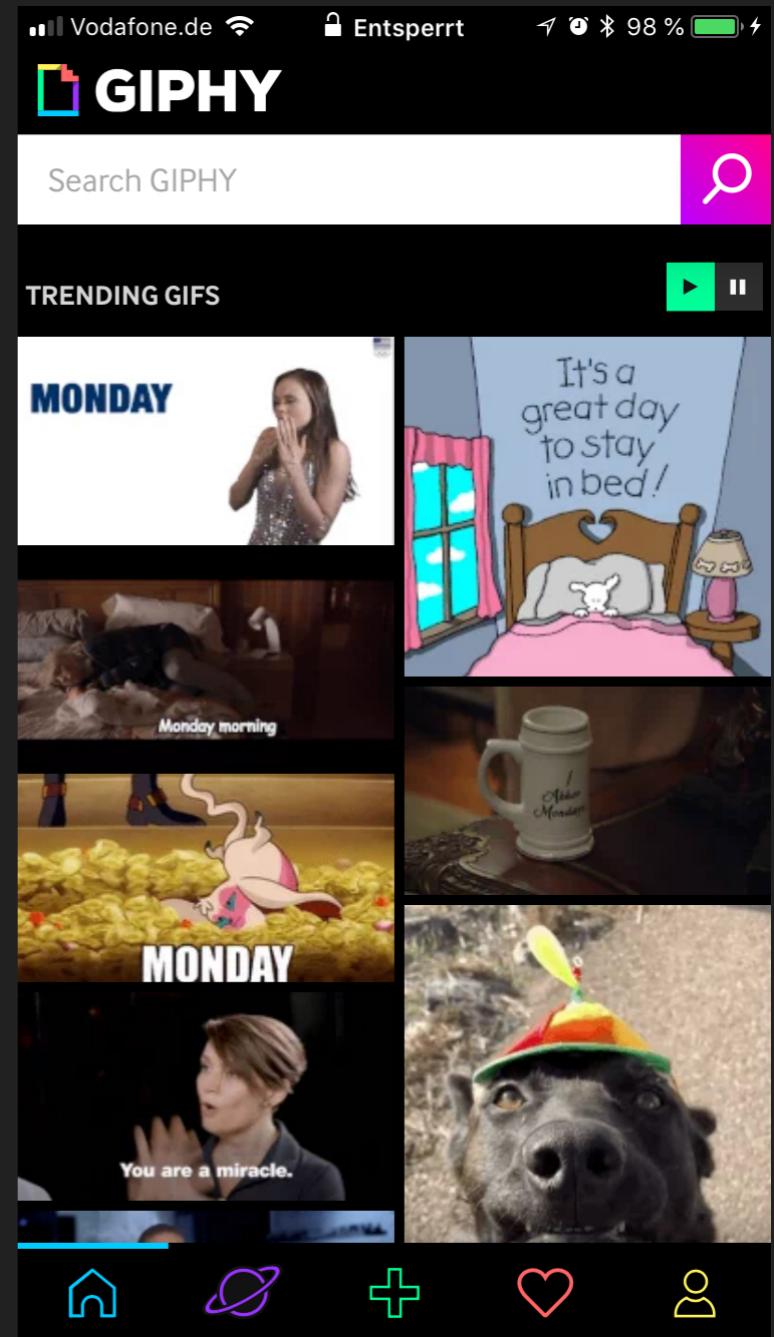
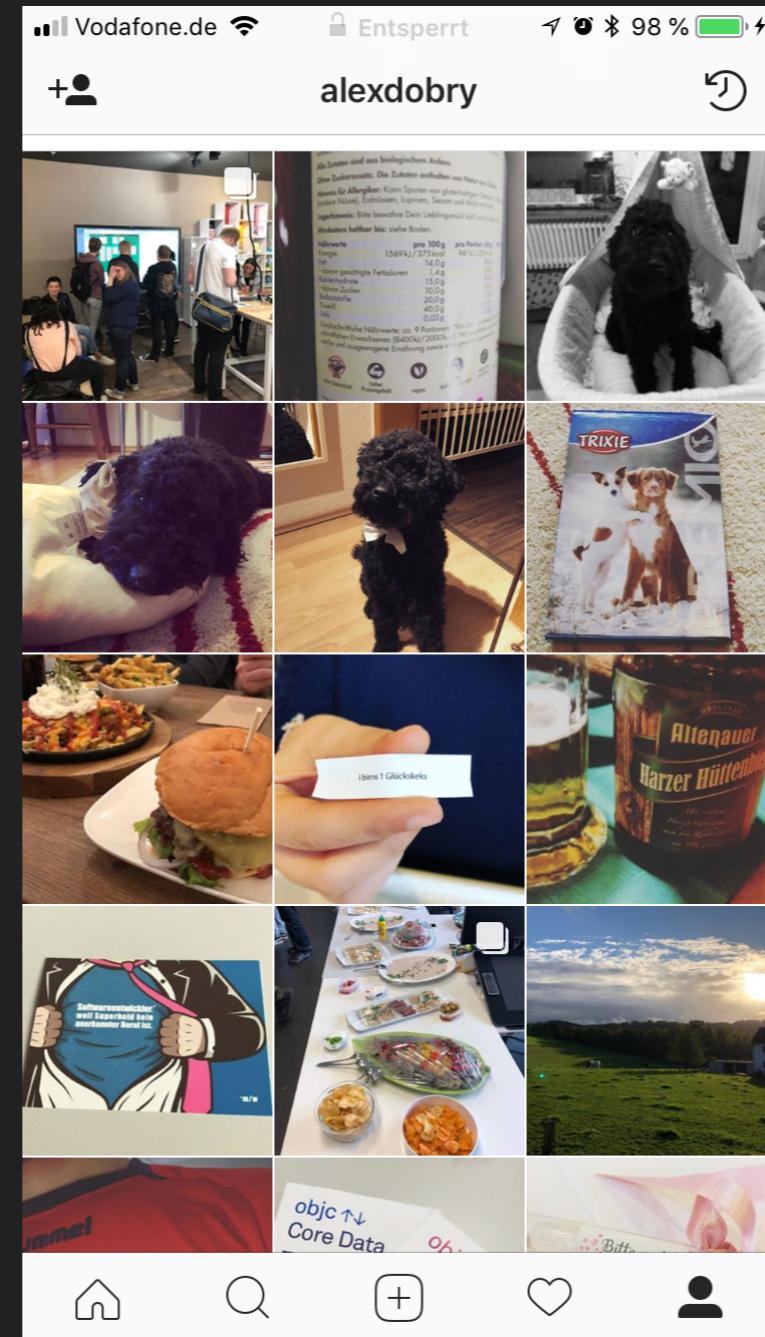
CollectionView

Zusammenfassung

CollectionView

- CollectionViews werden verwendet, um **große Mengen** an Daten in einem vertikal oder horizontal scrollbaren Grid zu präsentieren
- Ein Eintrag einer CollectionView ist eine **UICollectionViewCell**, welche als CustomView zu implementieren ist
- Die Daten können **Gruppiert** werden, um diese visuell oder semantisch aufzuteilen
 - Jede Gruppe präsentiert eine bestimmte Anzahl von Daten
 - Jede Gruppe kann eine eigene View als Header verwenden
- Die CollectionView fragt alle notwendigen Daten (Cell, Größe der Cell, Anzahl der Cells in einer bestimmten Section) mittels **DataSource** und **Delegate** an
 - UICollectionView abstrahiert die Kommunikation zu DataSource und Delegate über einen **IndexPath** (Kombination aus Section und Row)
 - DataSource und Delegate müssen die internen Daten (Array vom Model) mithilfe des IndexPath mappen. So kann **indexPath.row** der Index für das Array vom Model sein
 - Auf Basis des IndexPath werden Model (Array von Daten) und View (UICollectionView an der Stelle IndexPath) **synchronisiert**
- Im Storyboard kann man sowohl eine einzelne **UICollectionView** oder einen fertigen **UICollectionViewController** (superView ist eine UICollectionView und DataSource und Delegate sind gesetzt) verwenden

CollectionView



CollectionView

The image displays three screenshots from iOS devices demonstrating the use of CollectionView:

- Left Screenshot: Apple Watch Settings - Meine Uhr**

This screen shows the "Meine Uhr" section of the Apple Watch settings. It includes:
 - A preview of the "Apple Watch von Alexander" watch face.
 - A section for "MEINE ZIFFERBLÄTTER" with three watch faces: "Aktivität Analog", "Utility", and "10".
 - Links to "Komplikationen", "Mitteilungen", and "App-Layout".
- Middle Screenshot: Netflix App - Previews**

This screen shows the "Previews" section of the Netflix app. It includes:
 - Circular thumbnails for "HAUS DES GELDES", "LOST IN SPACE", and "THE RAIN".
 - A section titled "Beliebt auf Netflix" featuring "HAUS DES GELDES", "THE RAIN", and "RIVERDALE".
 - A section titled "Derzeit beliebt" featuring "LOST IN SPACE", "the Vampire Diaries", and "AVATAR".
 - A section titled "Beliebt bei Kindern auf Netflix".
- Right Screenshot: Online Store - BUSINESS**

This screen shows a product listing for men's shirts. It includes:
 - Two shirt models: one in white and one in dark blue.
 - Product details:
 - White shirt: "ELISHA SLIM FIT - Businesshemd" starting at 89,90 €.
 - Dark blue shirt: "Businesshemd - navy" starting at 16,95 €.

Heute

Funktionen und Closures

Multithreading

CollectionView

Zusammenfassung

Zusammenfassung

- Funktionen in Swift sind first class citizen
 - Closures sind **anonyme Funktionen** (Funktionsliterale), die alle Variablen in ihrem Scope erfassen (capture) und innerhalb der Closure verfügbar machen
 - Closures in iOS sind oftmals sog. **Completion-Handler** (z.B. completion: (Bool) -> ()) oder **Konkretisierungen** (z.B. animation: () -> ())
 - Es wird zwischen **Escaping** und **Non-Escaping** Closures unterschieden. Escaping Closures tendieren zum Memory-Leak, was allerdings leicht behoben werden kann
- Für asynchrone Operationen werden Grand-Central-Dispatch (GCD) oder OperationQueues verwendet
 - Es existieren **eine Main-Queue** und **mehrere Global-Queues**, die auf Basis einer Priorität angefragt werden
 - Wenn eine Funktion asynchron auf die Queue gelegt wird (95% der Benutzung), läuft das Programm **unmittelbar** weiter
 - Bei GCD ist der Workflow: request global queue -> *off main thread* perform heavy computation -> request main queue -> *on main thread* update UI
- CollectionViews präsentieren große Datenmengen in vertikal oder horizontal scrollbaren Grids
 - Hierbei werden eine **Menge von Daten** (Array vom Model) in **UICollectionViewCells** (View) **abgebildet** und mittels IndexPath synchron gehalten
 - Ein Großteil dieser Implementierung erfolgt über die Funktion von **DataSource** und **Delegate**