App-Entwicklung für iOS und OS X

SS 2016

Stephan Gimbel

Memory Management

Automatic Reference Counting

- Referenz-Typen (Klassen) werden auf dem Heap gespeichert.
- Woher weiss das System, wann es diesen Speicher auf dem Heap wieder freigeben soll?
- Es zählt Referenzen zu Instanzen und wenn keine existiert, werden diese verworfen.
- Dies geschieht automatisch.
- Dies ist als “Automatic Reference Counting” (kurz ARC) bekannt und ist KEINE Garbage Collection.

Beeinflussen des ARC

- ARC kann beeinflusst werden, indem eine Referenz-Typ `var` mit diesen Schlüsselworten deklariert wird...

```
strong
weak
unowned
```

Memory Management

strong

`strong` ist “normales” Reference Counting.

Solange wie von irgendwoher ein strong Pointer auf eine Instanz existiert, bleibt die Instanz auf dem Heap.

weak

`weak` bedeutet “wenn sich niemand dafür interessiert, dann interessiere ich mich auch nicht dafür, also auf `nil` setzen”.

Weil es `nil`-bar sein muss, betrifft `weak` nur Optional Pointer auf Referenz-Typen.

Ein `weak` Pointer hält NIEMALS ein Objekt auf dem Heap.

Beispiel: Outlets (`strong` von der View Hierarchie gehalten, daher kann ein Outlet `weak` sein).

unowned

`unowned` bedeutet “keine Referenz darauf zählen; crash wenn ich mich irre”.

Wird sehr selten verwendet.

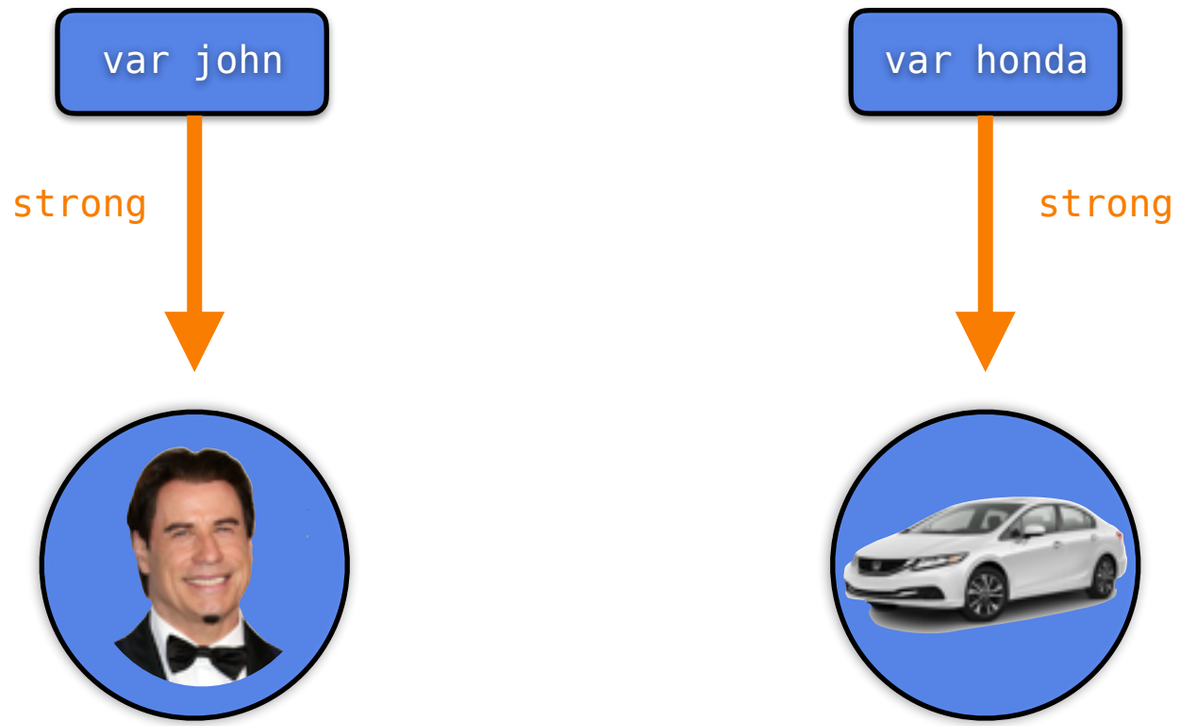
Normalerweise nur um Memory Cycles zwischen Objekten aufzubrechen.

Memory Management

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var car: Car?
    deinit { print("\(name) is being deinitialized") }
}
```

```
class Car {
    let model: String
    init(model: String) { self.model = model }
    var driver: Person?
    deinit { print("Car \(model) is being deinitialized") }
}
```

```
var john: Person?; var honda: Car?
john = Person(name: “John”); honda = Car(model: “Honda Civic”)
```

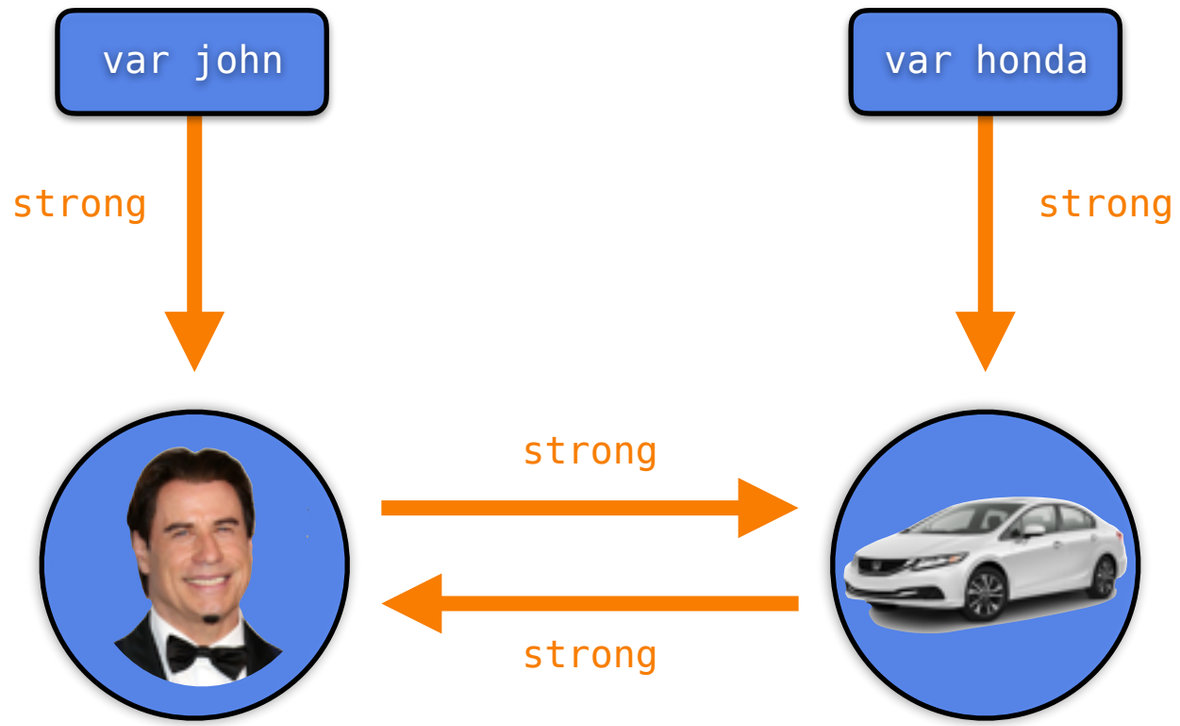


Memory Management

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var car: Car?
    deinit { print("\(name) is being deinitialized") }
}
```

```swift
class Car {
    let model: String
    init(model: String) { self.model = model }
    var driver: Person?
    deinit { print("Car \(model) is being deinitialized") }
}
```

```swift
john!.car = honda
honda!.driver = john
```

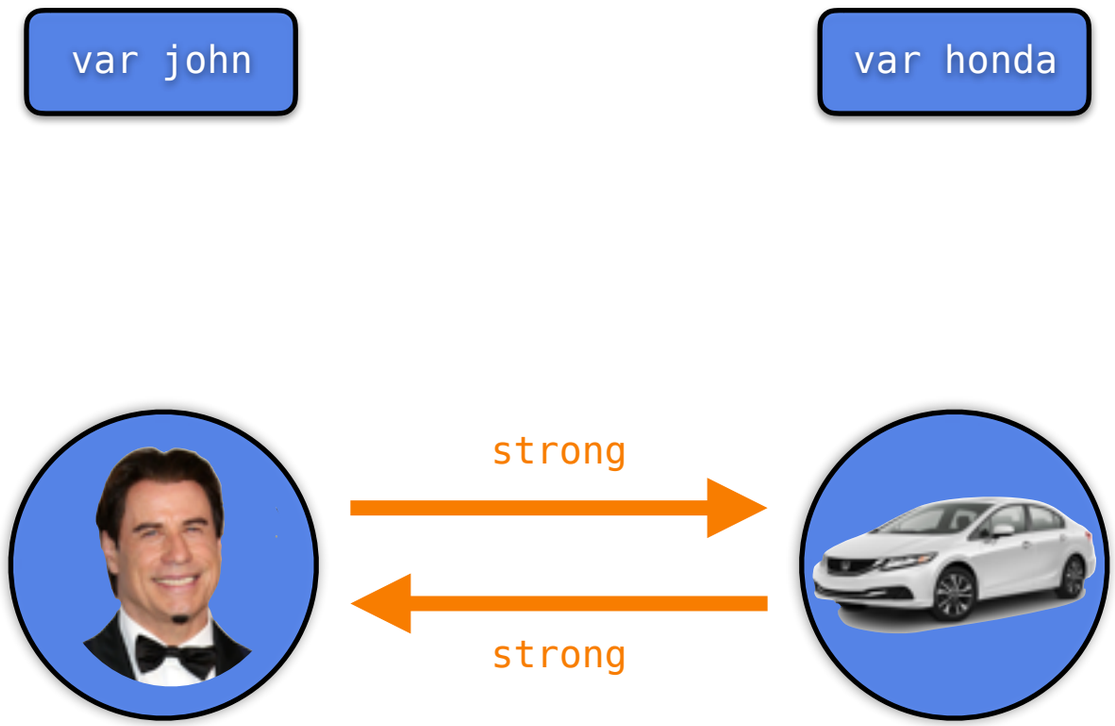


Memory Management

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var car: Car?
    deinit { print("\(name) is being deinitialized") }
}
```

```
class Car {
    let model: String
    init(model: String) { self.model = model }
    var driver: Person?
    deinit { print("Car \(model) is being deinitialized") }
}
```

```
john = nil
honda = nil
```



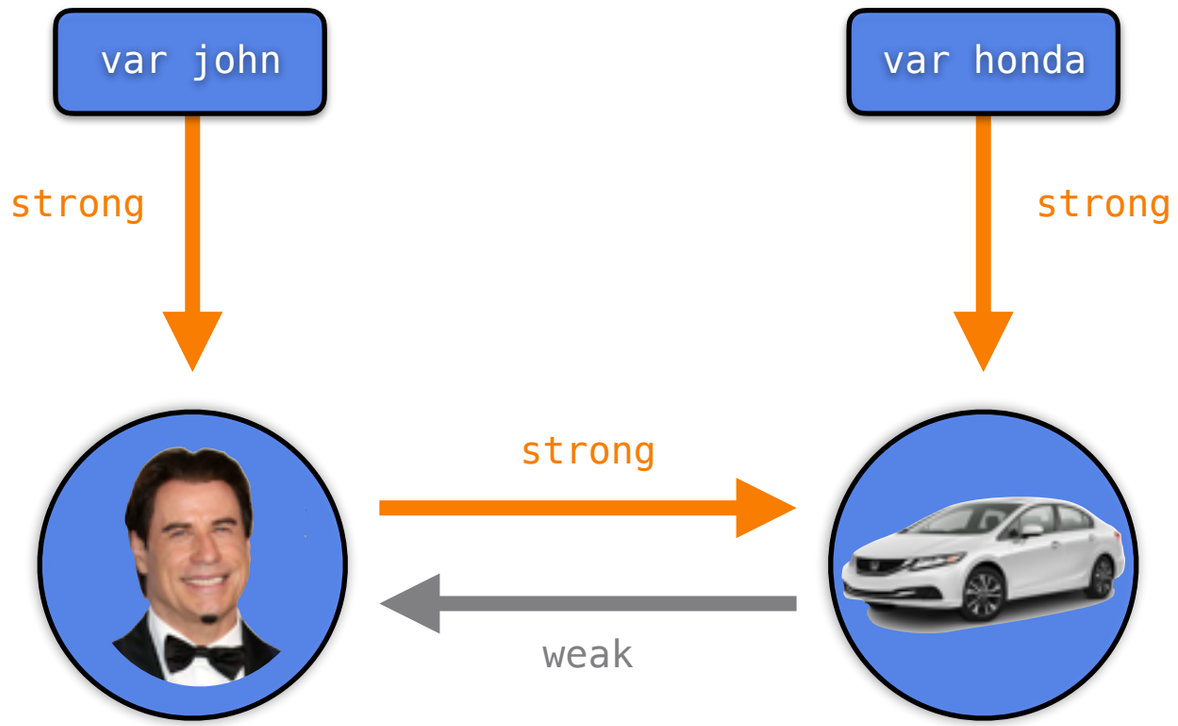
Memory Management

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var car: Car?
    deinit { print("\(name) is being deinitialized") }
}
```

```swift
class Car {
    let model: String
    init(model: String) { self.model = model }
    weak var driver: Person?
    deinit { print("Car \(model) is being deinitialized") }
}
```

```swift
var john: Person?; var honda: Car?
john = Person(name: “John”); honda = Car(model: “Honda Civic”)

john!.car = honda
honda!.driver = john
```

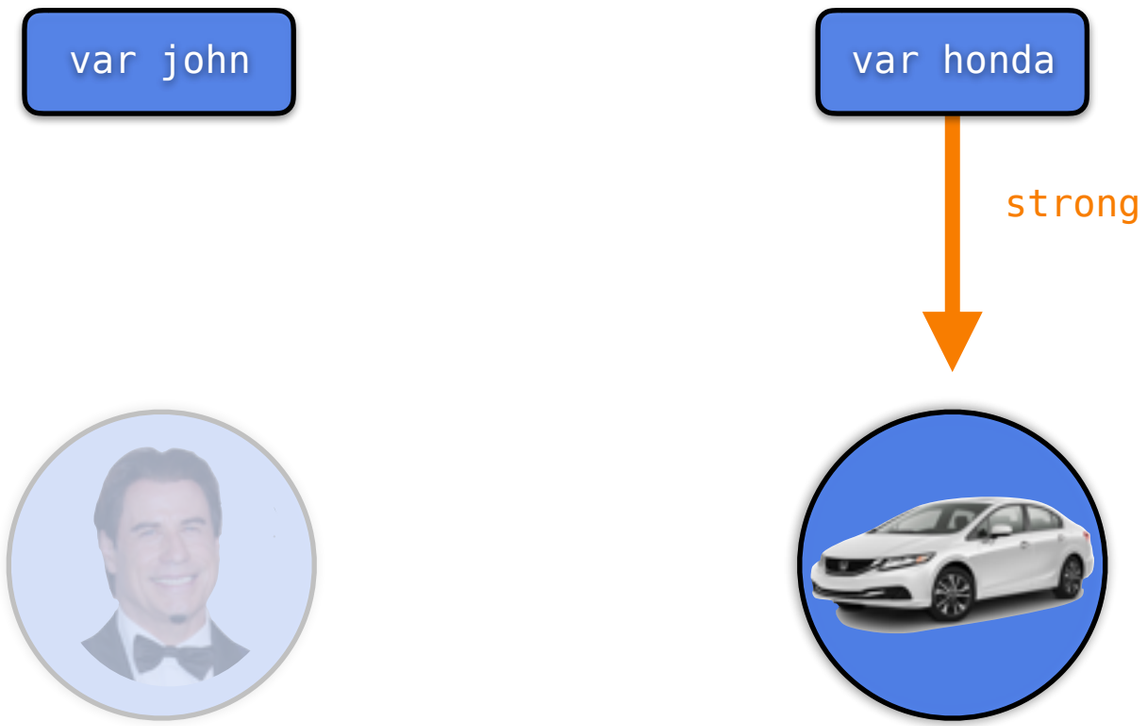


Memory Management

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var car: Car?
    deinit { print("\(name) is being deinitialized") }
}
```

```swift
class Car {
    let model: String
    init(model: String) { self.model = model }
    weak var driver: Person?
    deinit { print("Car \(model) is being deinitialized") }
}
```

```swift
john = nil
```



Memory Management

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var car: Car?
    deinit { print("\(name) is being deinitialized") }
}
```

```swift
class Car {
    let model: String
    init(model: String) { self.model = model }
    weak var driver: Person?
    deinit { print("Car \(model) is being deinitialized") }
}
```

```
john = nil
honda = nil
```

var john

var honda

Closures

Closures sind geschlossene Blöcke von Funktionalität, die übergeben und im Code genutzt werden können. Closures in Swift sind ähnlich zu Blocks in C und Objective-C, sowie Lambdas in anderen Programmiersprachen.

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

func backwards(s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversed = names.sort(backwards)
// reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

Closure Expression Syntax

```
{ (parameters) -> return type in
    statements
}
```

Closures

```
reversed = names.sort({ (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Kurze Closures werden in einer Zeile geschrieben

```
reversed = names.sort( { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

Aus dem Context lässt sich auf den Typ schließen (Inferring Type)

```
reversed = names.sort( { s1, s2 in return s1 > s2 } )
```

Implicit Returns aus Single-Expression Closures

```
reversed = names.sort( { s1, s2 in s1 > s2 } )
```

Shorthand Argument Names

```
reversed = names.sort( { $0 > $1 } )
```

Operator Functions

```
reversed = names.sort(>)
```

Closures

Trailing Closures

Wenn ein Closure an eine Funktion als letzter Parameter übergeben wird und das Closure lang ist, dann bietet sich die Verwendung eines Trailing Closure an.

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure({
    // closure's body goes here
})

// here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

```
reversed = names.sort() { $0 > $1 }

reversed = names.sort { $0 > $1 }
```

Closures

Capturing

Closures werden ebenfalls auf dem Heap gespeichert (sie sind Referenz-Typen).

Sie können in Arrays, Dictionaries, etc. abgelegt werden. Sie sind first-class Typen in Swift.

Weiterhin “capturen” sie Variablen die sie nutzen aus dem umgebenden Code auf den Heap.

Diese gecaptureten Variablen müssen auf dem Heap bleiben, solange das Closure auf dem Heap bleibt.

Dies kann einen Memory Cycle erzeugen...

Closures

Angenommen wir haben public API, um eine `UnaryOperation` zu erlauben, zu einen `CalculatorBrain` hinzugefügt zu werden.

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

Diese Methode macht nichts anderes als eine `UnaryOperation` zu einem Dictionary von `enum` hinzuzufügen (siehe Demo).

Angenommen ein View Controller fügt nun die Operation “red square dot” hinzu.

Die Operation berechnet die Wurzel, gibt das Ergebnis aber in rot aus.

```
addUnaryOperation(“🔴√”, operation: { (x: Double) -> Double in
    display.textColor = UIColor.redColor()
    return sqrt(x)
})
```

Closures

Angenommen wir haben public API, um eine `UnaryOperation` zu erlauben, zu einen `CalculatorBrain` hinzugefügt.

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

Diese Methode macht nichts anderes als eine `UnaryOperation` zu einem Dictionary von `enum` hinzuzufügen (siehe Demo).

Angenommen ein View Controller fügt nun die Operation “red square dot” hinzu.

Die Operation berechnet die Wurzel, gibt das Ergebnis aber in rot aus.

```
addUnaryOperation("🔴√", operation: { (x: Double) -> Double in
    display.textColor = UIColor.redColor()
    return sqrt(x)
})
```

Closures

Angenommen wir haben public API, um eine `UnaryOperation` zu erlauben, zu einen `CalculatorBrain` hinzugefügt.

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

Diese Methode macht nichts anderes als eine `UnaryOperation` zu einem Dictionary von `enum` hinzuzufügen (siehe Demo).

Angenommen ein View Controller fügt nun die Operation “red square dot” hinzu.

Die Operation berechnet die Wurzel, gibt das Ergebnis aber in rot aus.

```
addUnaryOperation(“🔴√”) { (x: Double) -> Double in
    display.textColor = UIColor.redColor()
    return sqrt(x)
}
```

Closures

Angenommen wir haben public API, um eine `UnaryOperation` zu erlauben, zu einen `CalculatorBrain` hinzugefügt.

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

Diese Methode macht nichts anderes als eine `UnaryOperation` zu einem Dictionary von `enum` hinzuzufügen (siehe Demo).

Angenommen ein View Controller fügt nun die Operation “red square dot” hinzu.

Die Operation berechnet die Wurzel, gibt das Ergebnis aber in rot aus.

```
addUnaryOperation(“🔴√”) { (x: Double) -> Double in
    display.textColor = UIColor.redColor()
    return sqrt(x)
}
```

Closures

Angenommen wir haben public API, um eine `UnaryOperation` zu erlauben, zu einen `CalculatorBrain` hinzugefügt.

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

Diese Methode macht nichts anderes als eine `UnaryOperation` zu einem Dictionary von `enum` hinzuzufügen (siehe Demo).

Angenommen ein View Controller fügt nun die Operation “red square dot” hinzu.

Die Operation berechnet die Wurzel, gibt das Ergebnis aber in rot aus.

```
addUnaryOperation(“🔴√”) {
    display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Dies kompiliert aber nicht.

Closures

Angenommen wir haben public API, um eine `UnaryOperation` zu erlauben, zu einen `CalculatorBrain` hinzugefügt.

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

Diese Methode macht nichts anderes als eine `UnaryOperation` zu einem Dictionary von `enum` hinzuzufügen (siehe Demo).

Angenommen ein View Controller fügt nun die Operation "red square dot" hinzu.

Die Operation berechnet die Wurzel, gibt das Ergebnis aber in rot aus.

```
addUnaryOperation("🔴√") {
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Swift zwingt uns hier `self.` anzugeben, um ins daran zu erinnern dass `self` hier gecaptured wird.

Das Model und der Controller zeigen nun aufeinander, durch das Closure.

Daher kann weder das Model, noch der Controller den Heap verlassen. Dies ist ein Memory Cycle.

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ <special variable Declarations> ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ me = self ] in
    me.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ unowned me = self ] in
    me.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ unowned self = self ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ unowned self ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ weak self ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ weak self ] in
    self?.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Wie können wir dies lösen?

Swift lässt uns das Capture Verhalten kontrollieren…

```
addUnaryOperation("🔴√") { [ weak weakSelf = self ] in
    weakSelf?.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

Versuchen Sie es doch mal selbst und modifizieren Sie die Calculator Demo mit dem hier gezeigten Beispiel...

Extensions

Erweitern von existierenden Datenstrukturen

Wir können Methoden/Properties zu `class/struct/enum` hinzufügen (auch wenn wir den Quellcode nicht haben).

Zum Beispiel fügt dies die Methode `contentViewController` zu `UIViewController` hinzu.

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

... kann verwendet werden um `prepareForSegue` Code zu vereinfachen...

```
var destination: UIViewController? = segue.destinationViewController
if let navcon = destination as? UINavigationController {
    destination = navcon.visibleViewController
}
if let myvc = destination as? MyVC { ... }
```

Extensions

Erweitern von existierenden Datenstrukturen

Wir können Methoden/Properties zu `class/struct/enum` hinzufügen (auch wenn wir den Quellcode nicht haben).

Zum Beispiel fügt dies die Methode `contentViewController` zu `UIViewController` hinzu.

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

... kann verwendet werden um `prepareForSegue` Code zu vereinfachen...

```
if let myvc = segue.destinationViewController.contentViewController
                    as? MyVC { ... }
```

Extensions

Erweitern von existierenden Datenstrukturen

Wir können Methoden/Properties zu `class`/`struct`/`enum` hinzufügen (auch wenn wir den Quellcode nicht haben).

Zum Beispiel fügt dies die Methode `contentViewController` zu `UIViewController` hinzu.

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

Die Referenz auf `self` bezieht sich auf das was es erweitert (`UIViewController`)

Extensions

Zusammenfassung (Wiederholung)

Erweitern von existierenden Datenstrukturen

- Wir können Methoden/Properties zu `class`/`struct`/`enum` hinzufügen (auch wenn wir den Quellcode nicht haben).

Ein paar Einschränkungen

- Keine Re-Implementierung von Methoden oder Properties die schon existieren (nur hinzufügen von neuen)
- Die Properties die hinzugefügt werden, dürfen keinen Storage haben

Wird leider oft missbraucht

- Sollte für Klarheit und Lesbarkeit sorgen, nicht zur Verschleierung.
- Nicht als Ersatz für gutes objekt-orientiertes Design verwenden.
- Am besten (für Beginner) für kleine, gekapselte Hilfsfunktionen verwenden.
- Kann verwendet werden um Code zu organisieren, benötigt aber eine gute Software-Architektur.
- Im Zweifelsfall (jetzt noch) nicht benutzen.

UIScrollView

Hinzufügen eines Subviews zu einem normalen UIView ...

```
logo.frame = CGRect(x: 300, y: 50, width: 120, height: 180)
view.addSubview(logo)
```

UIScrollView

Hinzufügen eines Subviews zu einem UIScrollView ...

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
logo.frame = CGRect(x: 2700, y: 50, width: 120, height: 180)
scrollView.addSubview(logo)
```

UIScrollView

Hinzufügen eines Subviews zu einem UIScrollView ...

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
logo.frame = CGRect(x: 2700, y: 50, width: 120, height: 180)
scrollView.addSubview(campus)
```

UIScrollView

Hinzufügen eines Subviews zu einem UIScrollView ...

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
logo.frame = CGRect(x: 2700, y: 50, width: 120, height: 180)
scrollView.addSubview(campus)
```

UIScrollView

Hinzufügen eines Subviews zu einem UIScrollView ...

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
logo.frame = CGRect(x: 2700, y: 50, width: 120, height: 180)
scrollView.addSubview(campus)
```

UIScrollView

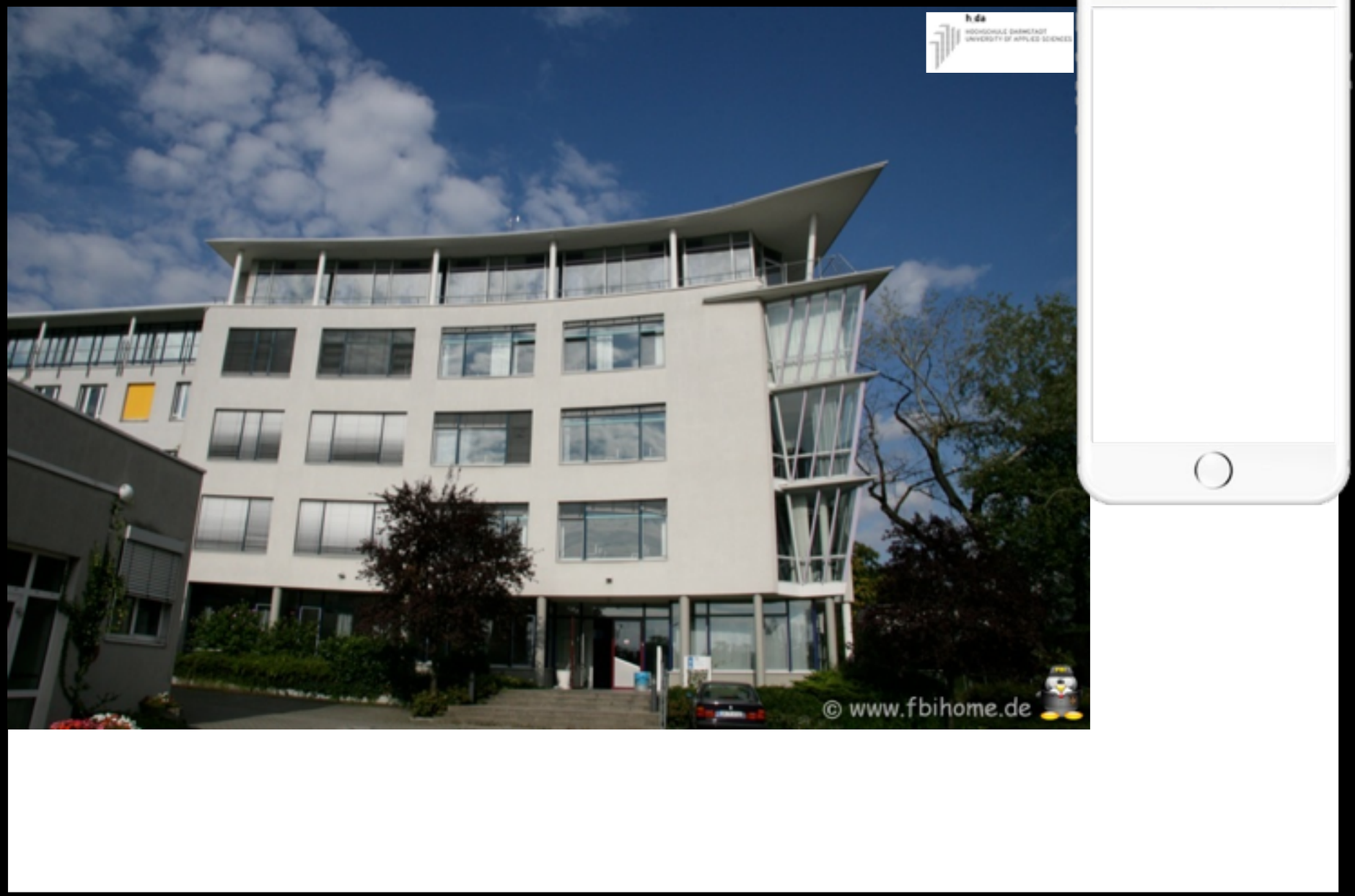
Positionierung eines Subviews in einem UIScrollView ...

```
campus.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
```

UIScrollView

Positionierung eines Subviews in einem UIScrollView ...

```
campus.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)
```



UIScrollView

Positionierung eines Subviews in einem UIScrollView …

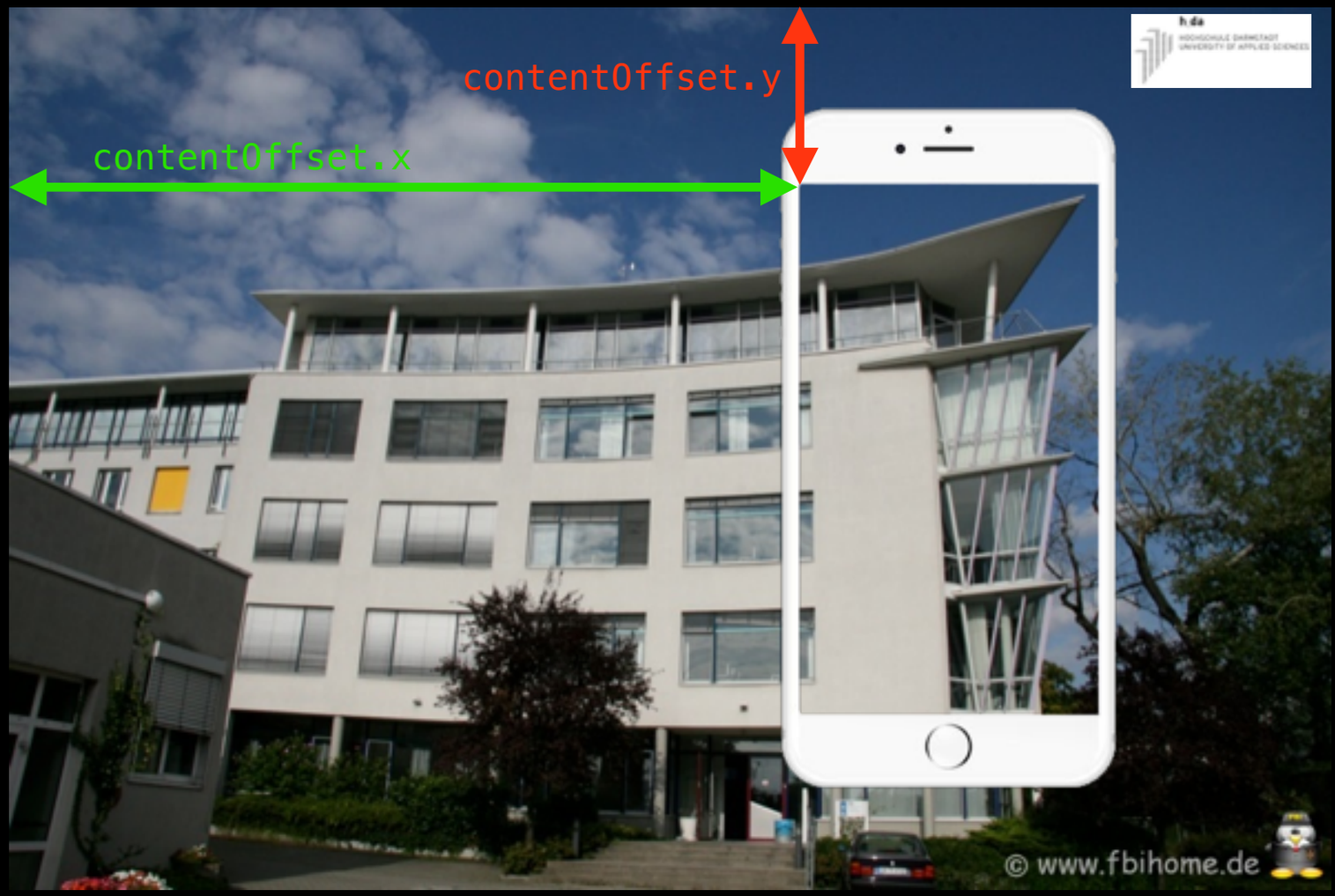
```
campus.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)
scrollView.contentSize = CGSize(width: 2500, height: 1600)
```

UIScrollView

Positionierung des Scrollview im Content?

```
let upperLeftOfVisible: CGPoint = scrollView.contentOffset
```

Im Koordinatensystem der Content-Fläche.



UIScrollView

Welche Fläche in einem Subview ist aktuell sichtbar?

```
let visibleRect: CGRect = campus.convertRect(scrollView.bounds, fromView:
                                                                scrollView)
```

Im Koordinatensystem der Content-Fläche.

Wozu `convertRect`? Weil die `scrollView` `bounds` im `scrollView` Koordinatensystem sind. Vielleicht wurde im `scrollView` auch gezoomt…

UIScrollView

Wie können wir einen Scrollview erstellen?

Wie jeden anderen `UIView` auch. Drag-Out in Storyboard oder verwenden von `UIScrollView(frame:)`.

Oder Auswahl eines `UIView` im Storyboard und dann "Embed In → Scroll View" aus dem Editor Menü.

Um einen "zu großen" `UIView` in Code mittels `addSubview` hinzuzufügen...

```
let image = UIImage(named: "bigimage.jpg")
let iv = UIImageView(image: image) // iv.frame.size = image.size
```

Bei Bedarf mehrere Subviews hinzufügen.

Alle Subview Frames sind im `UIScrollView`'s Content Area Koordinatensystem (also (0,0) für oben links und width & height von `contentSize.width & .height`)

Danach nicht vergessen die `contentSize` zu setzen

Es wird gerne vergessen diesen letzten Schritt zu machen:

`scrollView.contentSize = imageView.bounds.size` (Beispiel)

UIScrollView

Scrollen in Code

```
func scrollRectToVisible(CGRect: animated: Bool)
```

Andere Dinge, die in einem Scrollview gemacht werden können

- Angeben ob Scrolling enabled ist.
- Scrollrichtung einschränkten auf den ersten “Move” des Users.
- Style des Scroll-Indicators (`flashScrollIndicators` wenn der Scrollview auftaucht)

UIScrollView

Zooming

Alle `UIView`'s haben ein Property (`transform`), welches eine affine Transformation (Translation, Skalierung, Rotation) ist.

Scrollviews modifizieren diese Transformation, wenn gezoomt wird.

Zoom beeinflusst ebenfalls die Scrollview's `contentSize` und `contentOffset`.

Funktioniert nicht, wenn die minimum/maximum Zoom Scale nicht gesetzt ist

```
scrollView.minimumZoomScale = 0.5 // halbe normale Größe
scrollView.maximumZoomScale = 2.0 // doppelte normale Größe
```

Funktioniert nicht ohne die delegate Methode die den View spezifiziert der gezoomt werden soll

```
func viewForZoomingInScrollView(sender: UIScrollView) -> UIView
```

Wenn der Scrollview nur einen Subview hat, diesen zurückgeben.

Mehr als einer? Können wir frei wählen.

Zoomen in Code

```
var zoomScale: CFFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGFloat, animated: Bool)
```

UIScrollView

```
scrollView.zoomScale = 1.2
```

UIScrollView

```
scrollView.zoomScale = 1.0
```

UIScrollView

```
scrollView.zoomScale = 1.2
```

UIScrollView

```
zoomToRect(CGRect, animated: Bool)
```

UIScrollView

```
zoomToRect(CGRect, animated: Bool)
```

UIScrollView

Viele delegate Methoden!

Scrollview hält uns auf dem laufenden über was gerade passiert.

Beispiel: Delegate Methode benachrichtigt uns, wenn das Zoomen endet

```
func scrollViewDidEndZooming (UIScrollView,
                              withView: UIView, // vorh. delegate Meth.
                              atScale: CGFloat)
```

Wenn der View mit der neuen Scale neu gezeichnet wird, sicherstellen dass die Transformation auf die Identität zurückgesetzt wird.

Multithreading

Queues

Bei Multithreading geht es meist um “Queues” in iOS.

Funktionen (normalerweise Closures) werden in einer Queue angeordnet.

Diese Funktionen werden dann aus der Queue geholt und in einem zugehörigen Thread ausgeführt.

Queues können “serial” (eines auf einmal) oder “concurrent” (mehrere Dinge zur gleichen Zeit) sein.

Main Queue

Es existiert eine spezielle serielle Queue, genannt “Main Queue”.

Alle UI Aktivität MUSS in dieser und nur in dieser Queue geschehen.

Im Gegenzug darf alle non-UI Aktivität die zeitaufwendig ist, NICHT in dieser Queue stattfinden.

Wir wollen dies, damit unser UI immer schnell reagiert!

Und ebenfalls da wir alles was im UI passiert vorhersehen wollen (seriell).

Funktionen werden nur in der Main Queue geholt und abgearbeitet, wenn es “ruhig” ist.

Andere Queues

iOS hat andere Queues, die wir für non-UI Dinge nutzen können (gleich mehr dazu).

Multithreading

Ausführen einer Funktion in einer anderen Queue

```
let queue: dispatch_queue_t = <get the queue we want>
dispatch_async(queue) { /* do your stuff here */ }
```

Die Main Queue (eine serielle Queue)

```
let mainQ: dispatch_queue_t = dispatch_get_main_queue()
```

Alle UI-Dinge müssen in dieser Queue erledigt werden.

Und alle zeitaufwendigen (und noch schlimmer, vielleicht blockierenden) Dinge müssen außerhalb dieser Queue erfolgen.

Häufig sieht dies so aus...

```
dispatch_async(not main queue) {
    // non UI-stuff that takes a while
    dispatch_async(dispatch_get_main_queue()) {
        // call UI functions with the results above
    }
}
```

Wie erhalten wir eine non-Main Queue? ...

Multithreading

Andere (concurrent) queues (also nicht die Main Queue)

Die meisten non-main-queue Dinge passieren in einer concurrent Queue mit einem bestimmten Quality of Service.

```
QOS_CLASS_USER_INTERACTIVE  // schnelle und hohe Priorität
QOS_CLASS_USER_INITIATED    // hohe Priorität, braucht viel Zeit
QOS_CLASS_USER_UTILITY      // lange Laufzeit
QOS_CLASS_USER_BACKGROUND   // User interessiert sich dafür nicht
                               (prefetching, etc.)
let queue = dispatch_get_global_queue(<one of above>, 0)
                          // wobei 0 reserviert ist für zukünftige Dinge
```

Dies sind wahrscheinlich die Queues, die wir benutzen werden, wenn wir etwas machen wollen ohne die Main Queue zu blockieren.

Wir können unsere eigenen seriellen Queues erstellen, wenn wir Serialisierung brauchen

```
let serialQ = dispatch_queue_create("name", DISPATCH_QUEUE_SERIAL)
```

Vielleicht zum Download von einer Webseite, aber ohne die Website mit parallelen Anfragen zu spammen.

Oder vielleicht für Dinge, die voneinander abhängig sind und in einer bestimmten Reihenfolge erledigt werden müssen.

Multithreading

- Wir kratzen nur an der Oberfläche
  - Es gibt noch viel mehr in GDC (Grand Central Dispatch)
  - Wie z.B. Locking, Protection von Critical Sessions, Reader & Writer, synchronous dispatch, usw.
  - Bei Interesse Blick in die Dokumentation werfen.

- Es existiert ebenfalls eine objekt-orientierte API für all dies
  - `NSOperationQueue` und `NSOperation`
  - Erstaunlicherweise wird die nicht-objekt-orientierte API sehr häufig verwendet.
  - Dies liegt daran, dass sich das Nesting von Dispatching im Code sehr gut lesen lässt.
  - Die objekt-orientierte API ist aber ebenfalls nützlich (speziell für komplizierteres Multithreading)

Multithreading

Multithreaded iOS API

Einige Dinge in iOS machen ihre Arbeit außerhalb der Main Queue.

Sie stellen uns vielleicht auch die Möglichkeit zur Verfügung etwas außerhalb der Main Queue zu machen.

Wir können Funktionen (normalerweise ein Closure) übergeben, welche manchmal außerhalb des Main Threads ausgeführt wird.

Nicht vergessen, wenn wir hier UI-Dinge machen wollen, dann muss ein dispatch zurück zur Main Queue erfolgen.

Multithreading

Beispiel für Multithreaded iOS API

Diese API erlaubt uns etwas von einer http URL zu einem lokalen File zu empfangen.

Offensichtlich können wir das nicht im Main Thread machen.

```
let session = NSURLSession(configuration:
                NSURLSessionConfiguration.defaultSessionConfiguration())
if let url = NSURL(string: “http://url”) {
    let request = NSURLRequest(URL: url)
    let task = session.downloadTaskWithRequest(request) { (localURL, response, error) in
        /* do the UI thing here with the result of the download? */
    }
    task.resume()
}
```

Die Antwort zum Kommentar oben ist “nein”.

Weil der Block nicht in der Main Queue läuft.

Wie können wir dies beheben?

Eine Möglichkeit besteht darin eine Variante der API zu nutzen, die es erlaubt die Queue zu spezifizieren.

Ein anderer Weg ist…

Multithreading

Beispiel für Multithreaded iOS API

Einfach zur Main Queue zurück dispatchen…

```
let session = NSURLSession(configuration:
                NSURLSessionConfiguration.defaultSessionConfiguration())
if let url = NSURL(string: “http://url”) {
    let request = NSURLRequest(URL: url)
    let task = session.downloadTaskWithRequest(request) { (localURL, response, error) in
        dispatch_async(dispatch_get_main_queue()) {
        /* do the UI thing here with the result of the download? */
        }
    }
    task.resume()
}
```

Die Antwort zum Kommentar oben ist “ja”.

Da der UI Code der ausgeführt werden soll zurück zur Main Queue dispatched wurde.

Wir müssen aber beachten, dass der Code vielleicht MINUTEN nach dem ausführen des Requests ausgeführt wird.

Der User hat sein Vorhaben vielleicht schon lange aufgegeben.