

# SwiftUI & Concurrency

Programmieren für iOS



# Data Flow – viele Möglichkeiten

- Struct
  - Enum
  - Class
  - Source of Truth
  - Read-only
  - Read-write
- 
- let/var
  - @State
  - @Binding
  - @Environment
  - @StateObject
  - @ObservedObject
  - @EnvironmentObject

# SwiftUI & Software-Architektur

- Die Wahl der richtigen Konstrukte ist entscheidend, aber oft schwierig.
- Es gibt keine offiziellen App-Architektur-Patterns zu SwiftUI
  - Aber genügend (unterschiedliche) Meinungen im Internet
- Viel wichtiger: Problem/Lösung verstehen und strukturieren können
  - Wer auf abstrakter Ebene versteht, welche Komponenten wie interagieren müssen, um ein Ziel zu erreichen, findet in der Regel auch eine gute Abbildung davon mit Swift

# POP

- Sehr viele Framework-Methoden von Swift und SwiftUI sind mit Protokollen und teilweise Standard-Implementationen in Extensions definiert
- Diese ersetzen oft die Basis-Klassen und bieten ähnliche Möglichkeiten wie Vererbung mit mehr Flexibilität
- Apple nennt dies **Protocol-Oriented Programming** als Pendant zu Object-Oriented-Programming
- Trotzdem gilt auch hier: Die Aufteilung vom Code in Protokolle soll nicht erzwungen werden

```
protocol Animal {  
    func speak()  
}  
  
extension Animal {  
    func speak() {  
        print("I don't know how to")  
    }  
}  
  
struct Dog: Animal {  
}  
  
class Human: Animal {  
    func speak() {  
        // ...  
    }  
}
```

# User Defaults

- Letzte Übung: Speichern von Teams als JSON-Datei
- Für kleine Daten-Mengen einfacher: UserDefaults
  - z.B. Benutzereinstellungen, App-Starts
  - Erscheinen nicht bei den App-spezifischen System-Einstellungen
- Meistens genügt das Singleton **standard**
  - Weitere Initializer für geteilte Einstellungen (z.B. App und Widget)



```
var appStarts = UserDefaults.standard.integer(forKey: "app-starts")
appStarts += 1
UserDefaults.standard.set(appStarts, forKey: "app-starts")
```

# AppStorage

- UserDefaults können direkt in SwiftUI verwendet werden, man verliert aber die eindeutige Source of Truth
- **@AppStorage** erlaubt den Zugriff einfach und korrekt

```
struct ContentView: View {  
    @AppStorage("counter") var counter = 1  
    var body: some View {  
        Stepper("Counter", value: $counter)  
    }  
}
```

# Setters & Getters

- Properties werden manchmal mit gettern und setttern von internem Wert abgeleitet, das kann zu viel Boilerplate-Code führen

Welchen Wert hat value?



```
private var _value: Int?  
  
public var value: Int {  
    get {  
        if let _value {  
            return _value  
        }  
        else {  
            return 42  
        }  
    }  
    set {  
        if _value == nil {  
            _value = newValue  
        }  
    }  
}
```

# Property Wrapper

- Mit der `@propertyWrapper` Annotation können Getter und Setter abstrakt definiert werden
- Die Variabel kann danach normal verwendet werden, entspricht aber dem `wrappedValue`

```
class MyClass {  
    @WriteOnce var myValue: Int  
    func foo() {  
        let n = myValue + 1  
        print(n) // 43  
        myValue = 50  
    }  
}
```

```
@propertyWrapper struct WriteOnce {  
  
    var _value: Int?  
  
    var wrappedValue: Int {  
        get {  
            if let _value {  
                return _value  
            }  
            else {  
                return 42  
            }  
        }  
        set {  
            if _value == nil {  
                _value = newValue  
            }  
        }  
    }  
}
```

# Projected Value

- Mit Underscore kann auf das Konstrukt hinter dem Property Wrapper zugegriffen werden
- Falls eine Property `projectedValue` definiert ist, kann diese mit dem Dollar-Zeichen verwendet werden



```
@propertyWrapper struct WriteOnce {  
    var _value: Int?  
  
    var wrappedValue: Int { ... }  
  
    var projectedValue: Int = 42  
    var title: String = ""  
}  
  
class MyClass {  
    @WriteOnce var myValue: Int  
  
    func changeDefault() {  
        _myValue.title = "foo"  
        $myValue = 100  
    }  
}
```

# Projected Value

- Mit Underscore kann auf das Konstrukt hinter dem Property Wrapper zugegriffen werden
- Falls eine Property `projectedValue` definiert ist, kann diese mit dem Dollar-Zeichen verwendet werden

Richtig, SwiftUI verwendet  
sehr viele solche  
Property Wrappers!



```
@propertyWrapper struct WriteOnce {  
  
    var _value: Int?  
  
    var wrappedValue: Int { ... }  
  
    var projectedValue: Int = 42  
    var title: String = ""  
}  
  
class MyClass {  
    @WriteOnce var myValue: Int  
  
    func changeDefault() {  
        _myValue.title = "foo"  
        $myValue = 100  
    }  
}
```

# SwiftUI – Das wars!

- Es gibt natürlich sehr, sehr viele weitere Views und Modifiers
  - (Z.B. [TextField](#), [Stepper](#), [ColorPicker](#), [ProgressView](#), [blur](#), [blendMode](#), ...)
  - Learning-by-doing: Aktuell gibt es ca. 70 verschiedene Views – es genügt, diese kennen zu lernen, wenn man ein konkretes UI-Problem lösen möchte
- Custom User Interfaces: Genau so möglich
  - Einfache Views kombinieren, mehr Overlays und Stacks, weniger Forms
- Wie geht's weiter?
  - SwiftUI, falls für andere Frameworks relevant
  - Nächste Woche: UIKit
  - Heute: Swift ohne UI (Concurrency)

# Concurrency & Parallelism

Programmieren für iOS

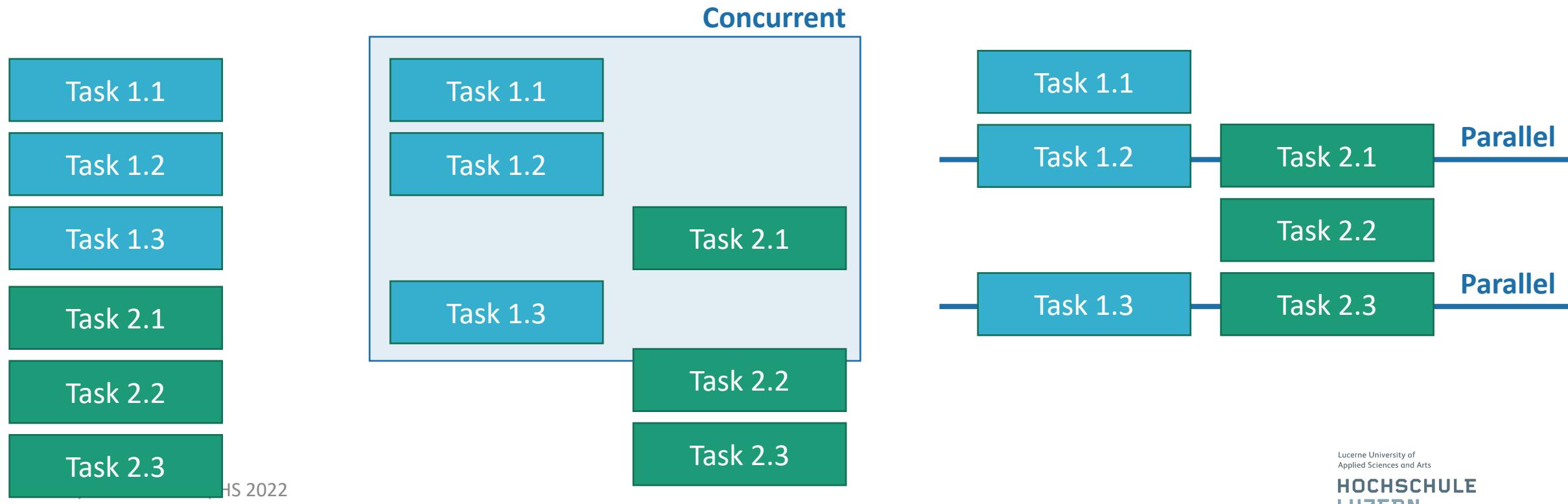


# Definitionen

- Die „**Tasks**“ (Aufgaben) einer iOS-App werden nur selten komplett seriell abgearbeitet. Zwei Begriffe sind für die Diskussion zu unterscheiden:
  - **Concurrency** (Gleichzeitigkeit, Nebenläufigkeit)
  - **Parallelism** (Parallelisierung)
- Viele Definitionen der beiden Begriffe tönen sehr ähnlich:
  - „Concurrency is about dealing with lots of things at once but parallelism is about doing lots of things at once“ (Rob Pike)

# Definitionen 2

- Ein Programm als **nebenläufig/concurrent** bezeichnet, wenn ein Tasks Fortschritte machen kann, bevor ein anderer abgeschlossen wurde.
- Die Ausführung wird als **parallel** bezeichnet, wenn mehrere Tasks im gleichen Moment laufen.
- Letzteres ist nur mit mehreren CPU-Kernen möglich.

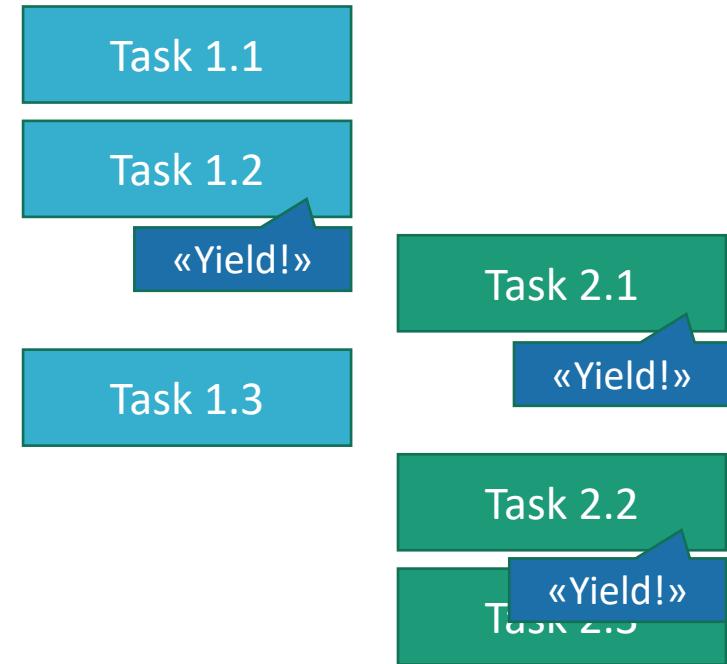


# Concurrency auf iOS

- Nebenläufigkeit ist auf iOS aus mehreren Gründen von Bedeutung:
  1. Die CPU ist nicht immer ausgelastet, das Bottleneck kann oft auch eine andere Hardware-Komponente sein.
    - Beispiele: Netzwerk-Antenne, Disk, GPS
  2. Gewisse Tasks, insbesondere UI-Updates, sollten regelmässig Fortschritte machen, damit das User Interface nicht einfriert.
  3. Ohne Nebenläufigkeit ist auch keine parallele Ausführung möglich, die modernen Multi-Core CPUs können nicht richtig genutzt werden.

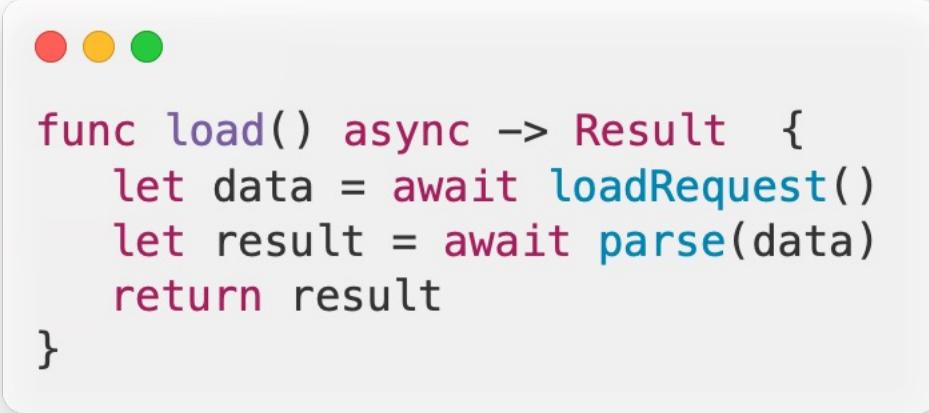
# Asynchron

- Damit Tasks nebenläufig durchgeführt, müssen Methoden asynchron ausgeführt werden können.
- Eine Methode wird als **asynchron** bezeichnet, wenn sie Möglichkeiten zum Aussetzen (**suspend**) bietet. Dies wird auch als aufgeben (**yield**) der Durchführung bezeichnet.



# Async Swift

- In Swift wird eine asynchrone Methode mit dem Keyword `async` markiert.
- Jeder Befehl, bei dem die Ausführung ausgesetzt werden kann, wird mit `await` markiert.



```
func load() async -> Result {
    let data = await loadRequest()
    let result = await parse(data)
    return result
}
```

# Parallelisierung

- Durch mehrere `await` kann eine Methode mehrmals nacheinander aussetzen. Die Reihenfolge der Durchführung bleibt aber gleich.
- In dem mehrere Variablen mit `async let` markiert werden und die Resultate mit einem `await` aggregiert werden, können mehrere Subtasks parallel ausgeführt werden.



```
let firstPhoto = await downloadPhoto(named: photoNames[0])
let secondPhoto = await downloadPhoto(named: photoNames[1])
let thirdPhoto = await downloadPhoto(named: photoNames[2])

let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```



```
async let firstPhoto = downloadPhoto(named: photoNames[0])
async let secondPhoto = downloadPhoto(named: photoNames[1])
async let thirdPhoto = downloadPhoto(named: photoNames[2])

let photos = await [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

# Networking

## Programmieren für iOS



# Networking

- Das Laden von Daten von einem Server ist eines der wichtigsten Beispiele von Nebenläufigkeit.
- Die System-API von `URLSession` bietet dafür (auch) asynchrone Methoden.

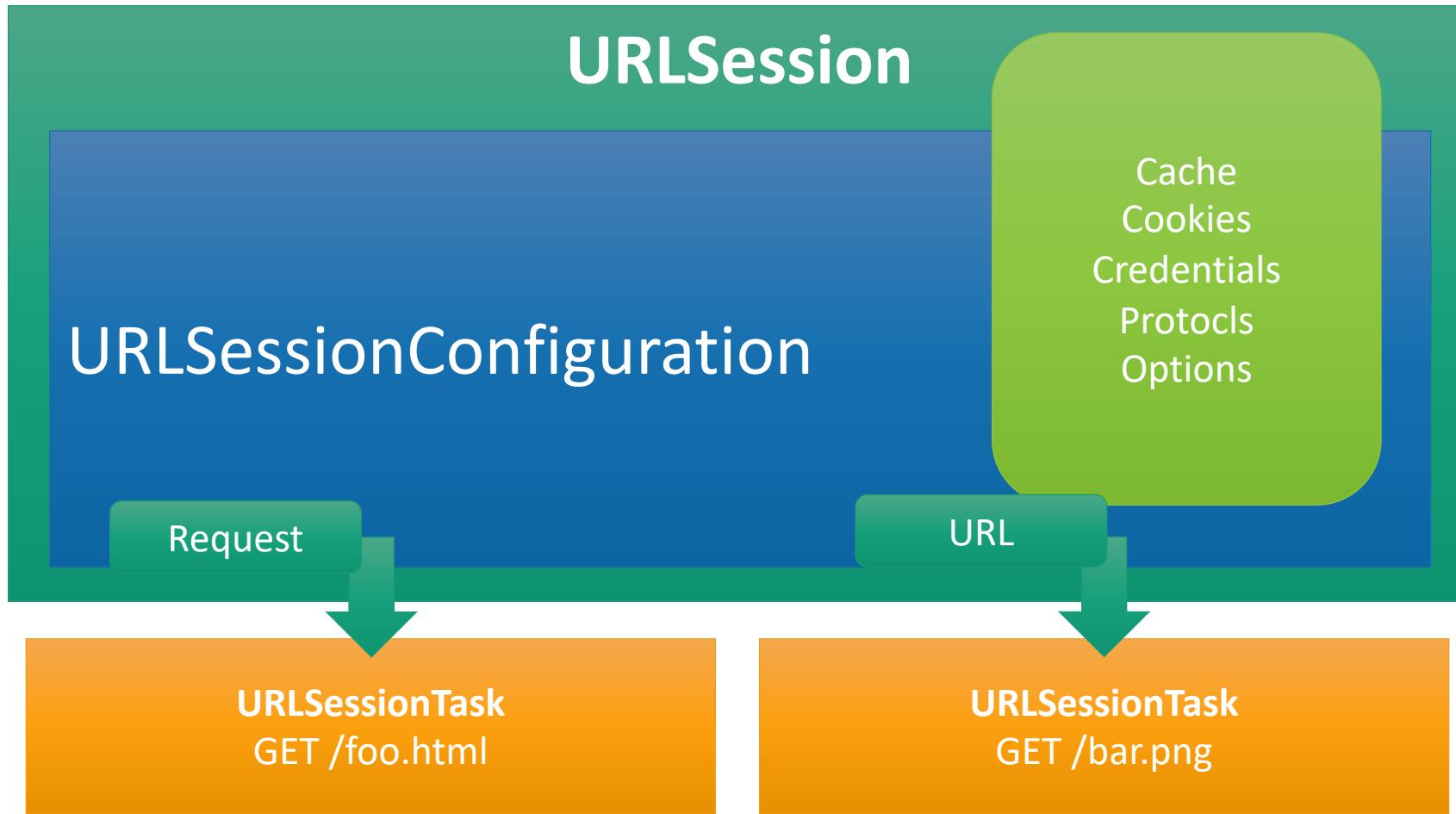
```
func loadData() async throws -> Data {  
    let (data, _) = try await URLSession.shared.data(from: myUrl)  
    return data  
}
```

# Kommunikation mittels URLs

- URL = Uniform Resource Locator, z.B.:
  - `http://www.hslu.ch`
  - `ftp://user:password@my.ftp.server.org`
  - `file://private/var/containers/Bundle/Application/Networking.app/file.txt`
- Entsprechende Klassen und Structs in Swift
  - `URL`
  - `URLSession`
  - `URLRequest`
  - `URLResponse`
  - ...

```
let myUrl = URL(string: "https://www.hslu.ch")!  
  
var request = URLRequest(url: myUrl)  
request.httpMethod = "POST"  
request.addValue("application/json", forHTTPHeaderField: "Accept")
```

# URLSession – Überblick



# JSON-Parsing mit Codable

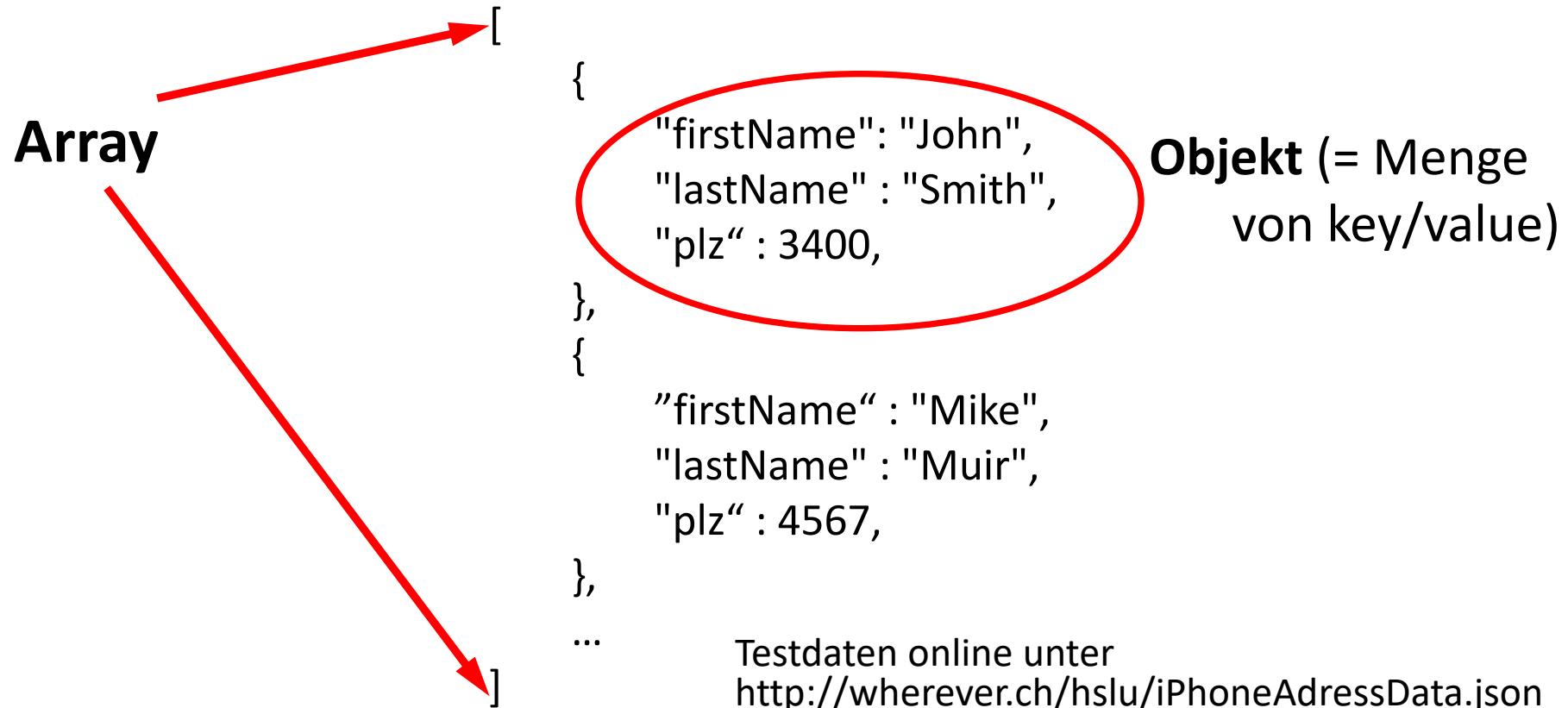
- Das Codable-Protokoll:
  1. Codable-Typ deklarieren
  2. Mit JSONDecoder direkt Data-Objekt darauf abbilden lassen

```
struct Person: Codable {  
    var firstName: String  
    var lastName: String  
}  
  
let model = try JSONDecoder().decode([Person].self, from: data)
```

Typ “Array<Person>”

# JSON

- JavaScript Object Notation
  - [http://de.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](http://de.wikipedia.org/wiki/JavaScript_Object_Notation)



# App Transport Security

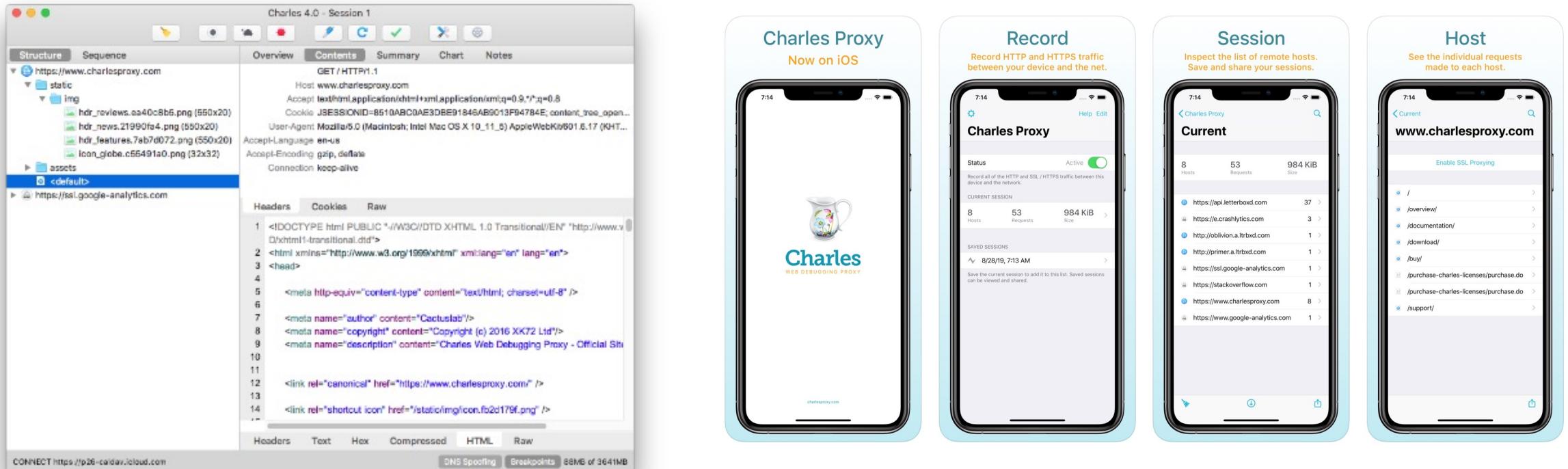
- Seit iOS 9 ist **https mit TLS 1.2 und Forward Secrecy per Default Pflicht!**

```
2015-10-22 21:01:55.079 ComAndCons[69975:5987456] App  
Transport Security has blocked a cleartext HTTP (http://)  
resource load since it is insecure. Temporary exceptions can  
be configured via your app's Info.plist file.
```

- Falls "nur" http gewünscht: URL in Projekt-Info "genehmigen":

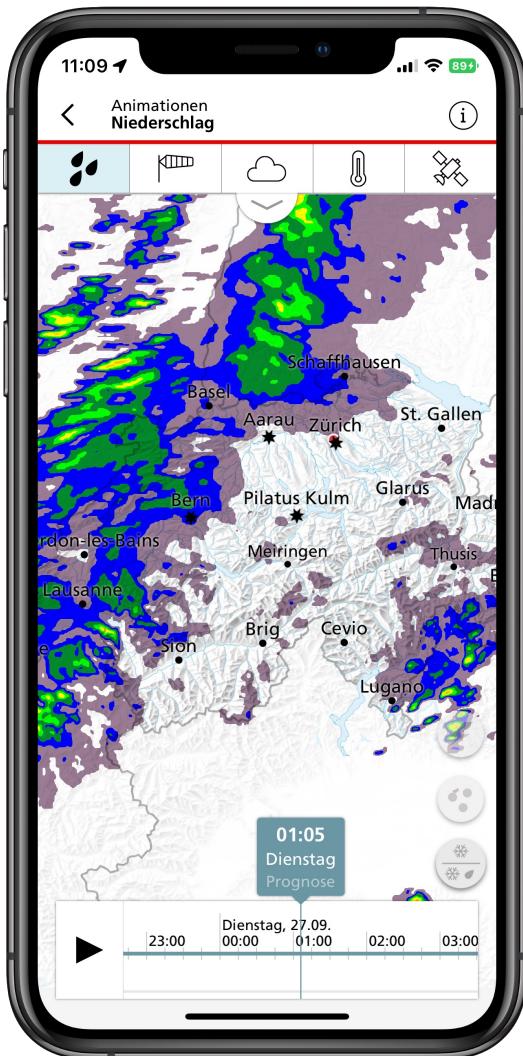
Info			
NSAppTransportSecurity		Dictionary	(2 items)
NSAllowsArbitraryLoads	Boolean	NO	
NSExceptionDomains		Dictionary	(1 item)
wherever.ch		Dictionary	(1 item)
NSEExceptionAllowsInsecureHTTPLoads	Boolean	YES	

# Proxy



- Mit Apps wie Charles oder Proxyman Network können Requests aufzeichnen oder bearbeitet werden
- Super zum Testen von Networking (oder um APIs von anderen Apps kennenzulernen)

# Networking – Case Study



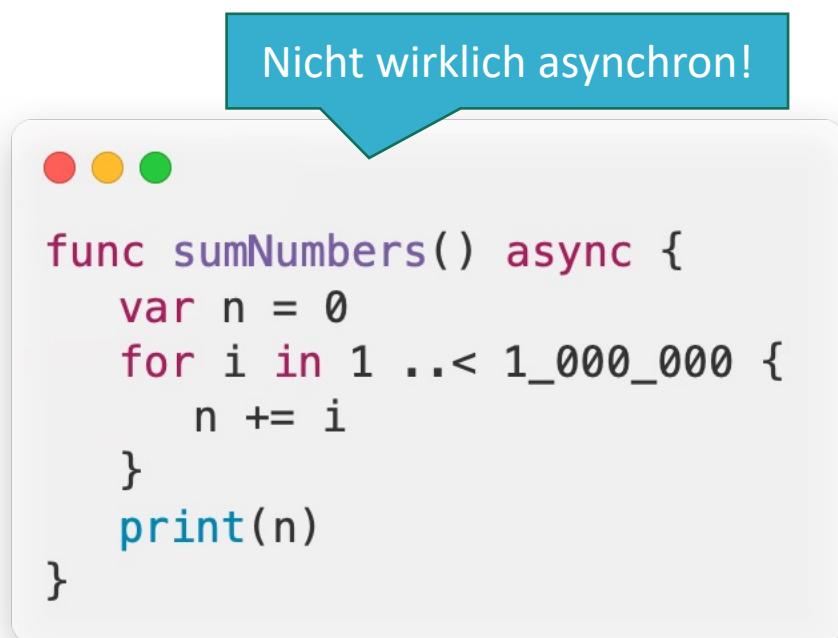
# Concurrency 2

Programmieren für iOS



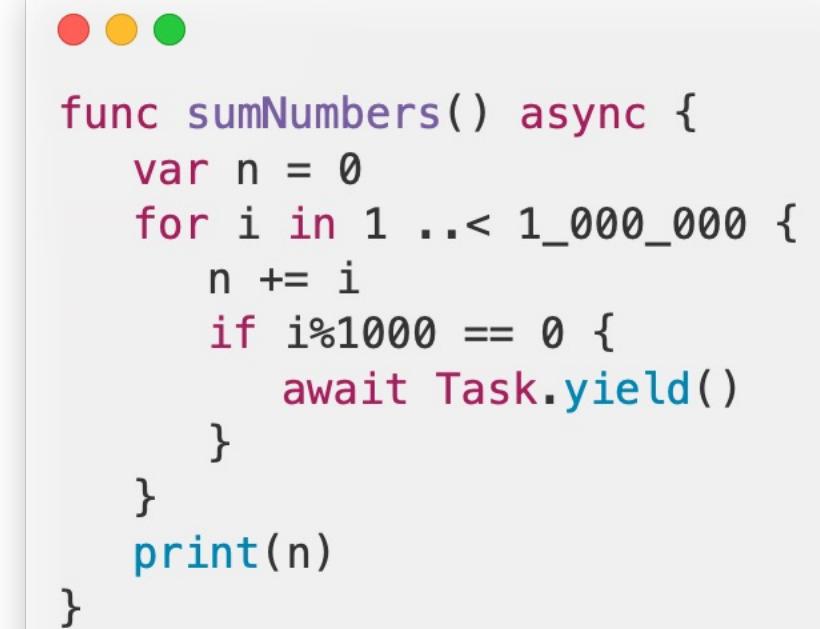
# Yield

- Mit `Task.yield()` kann explizit eine Code-Stelle als möglicher «Suspension Point» markiert werden.



Nicht wirklich asynchron!

```
func sumNumbers() async {
    var n = 0
    for i in 1 ..< 1_000_000 {
        n += i
    }
    print(n)
}
```



```
func sumNumbers() async {
    var n = 0
    for i in 1 ..< 1_000_000 {
        n += i
        if i%1000 == 0 {
            await Task.yield()
        }
    }
    print(n)
}
```

# Structured & Unstructured Concurrency

- Mit `async` und `await` wird implizit eine Hierarchie von Tasks und Subtasks definiert.
- Tasks können auch explizit mit dem Struct `Task` erstellt werden, dessen Initializer eine Closure erwartet.
- Solche Tasks müssen selbst verwaltet werden, z.B. beim Abbruch der Durchführung. Man spricht deshalb von **Unstructured Concurrency**.

```
● ○ ●  
let newPhoto = // ... some photo data ...  
let handle = Task {  
    return await add(newPhoto, toGalleryNamed: "Spring Adventures")  
}  
let result = await handle.value
```

# Cancellation

- Swift Concurrency setzt auf «**kooperativen**» Abbruch einer asynchronen Methode.
- Ein Subtask erhält lediglich die Information, dass die Task-Hierarchie abgebrochen wurde und muss selbst entscheiden, ob er die Ausführung beenden möchte.



```
func sumNumbers() async {  
    var n = 0  
    for i in 1 ..< 1_000_000 {  
        n += i  
        if i%1000 == 0 {  
            if Task.isCancelled {  
                return  
            }  
            await Task.yield()  
        }  
    }  
    print(n)  
}
```

# TaskGroup

- Swift bietet mit `withTaskGroup` eine API, um viele ähnliche Subtasks zu erstellen

```
let images = await withTaskGroup(of: UIImage.self, returning: [UIImage].self) { taskGroup in
    for title in titles {
        taskGroup.addTask {
            return await self.loadImage(title: title)
        }
    }

    var images: [UIImage] = []
    for await image in taskGroup {
        images.append(image)
    }
    return images
}
```

# TaskGroup

- Dynamische Anzahl an Child Tasks erstellen:

```
let images = await withTaskGroup(of: UIImage.self, returning: [UIImage].self) { taskGroup in
    for title in titles {
        taskGroup.addTask {
            return await self.loadImage(title: title)
        }
    }
}

var images: [UIImage] = []
for await image in taskGroup {
    images.append(image)
}
return images
}
```

Tasks erstellen

# TaskGroup

- Dynamische Anzahl an Child Tasks erstellen:

```
let images = await withTaskGroup(of: UIImage.self, returning: [UIImage].self) { taskGroup in
    for title in titles {
        taskGroup.addTask {
            return await self.loadImage(title: title)
        }
    }
}

var images: [UIImage] = []
for await image in taskGroup {
    images.append(image)
}
return images
}
```

Resultate aggregieren

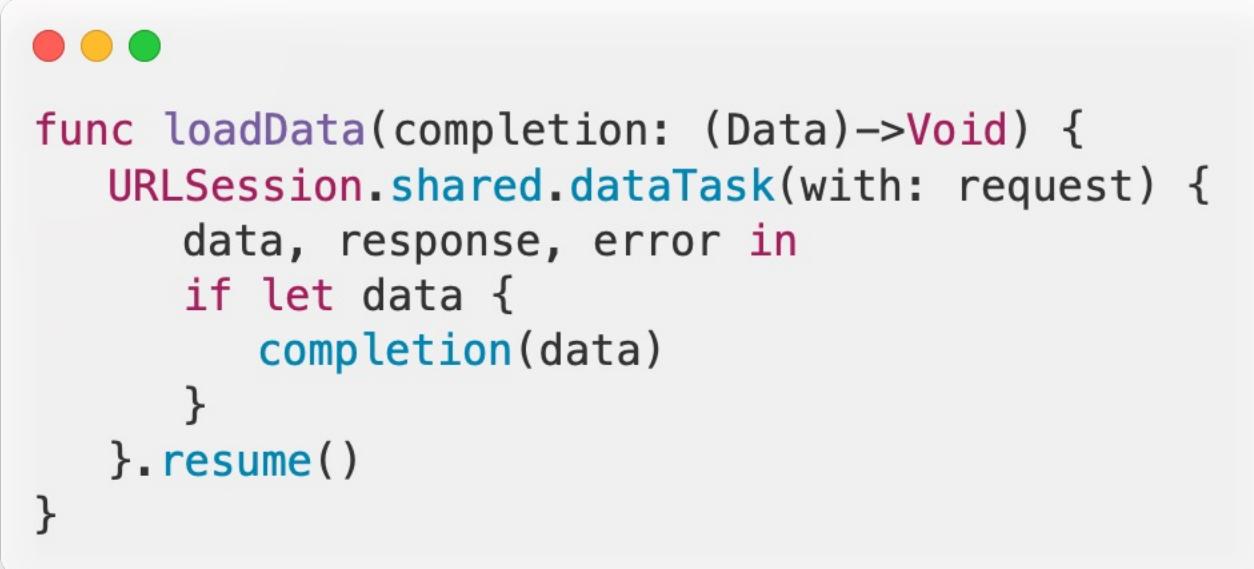
# TaskGroup, zweites Beispiel

- Falls ein Task Fehler werfen kann, muss withThrowingTaskGroup verwendet werden
- TaskGroup ohne Rückgabe möglich

```
await withThrowingTaskGroup(of: URLResponse.self, body: { taskGroup in
    for request in postRequests {
        taskGroup.addTask {
            let (_, response) = try await URLSession.shared.data(for: request)
            return response
        }
    }
})
```

# Ältere APIs

- Letzte Übung: Asynchroner Code ist mit Completion-Closures auch ohne `async/await` möglich.
- Das Ende einer Methode entspricht einem impliziten Yield, andere Tasks können ausgeführt werden, bevor die Completion-Closure läuft.
- Viele APIs wurden entwickelt, bevor Swift mit Concurrency erweitert wurde.



A screenshot of the Xcode code editor showing a completion closure example. The code uses URLSession.shared.dataTask to download data, then handles the result in a completion block. The code is as follows:

```
func loadData(completion: (Data) -> Void) {
    URLSession.shared.dataTask(with: request) {
        data, response, error in
        if let data = data {
            completion(data)
        }
    }.resume()
}
```

# Threads

- Threads bilden die Grundlage zur parallelen Ausführung von Code.
- Swift bietet eine API zum Erstellen oder Beenden von Threads.
- Es ist praktisch unmöglich zu entscheiden, ob weitere Threads die Performance erhöhen, da viele Faktoren wie CPU-Auslastung oder Gerät-Typ einen Einfluss haben.
- Stattdessen bietet iOS verschiedene «**Queues**», denen Code übergeben werden kann. Das System verwaltet eine «**Thread Pool**» mit mehreren Threads, welche die Queues abarbeiten.



```
Thread.detachNewThread {  
    print("Run on new Thread")  
}
```

# DispatchQueue

- Aus Übung: DispatchQueues sind Objekte, denen mit `sync` oder `async` Closures übergeben werden können.
- Queue: Das System garantiert, dass eine Closure nicht vor einer anderen startet, die später hinzugefügt wurde.



```
let myQueue = DispatchQueue(label: "ch.hslu.ios.MyQueue")
myQueue.async {
    print("Do some work here.")
}
print("The first block may or may not have run.")
myQueue.sync {
    print("Do some more work here.")
}
print("Both blocks have completed.")
```

**Do some work here.**  
**The first block may or may not have run.**  
**Do some more work here.**  
**Both blocks have completed.**

# DispatchQueue 2

- DispatchQueues sind standardmäßig seriell, mit dem Attribut `.concurrent` kann eine Queue erstellt werden, die mehrere Closures gleichzeitig auf mehreren Threads starten kann.
- Trotzdem gilt: Die Reihenfolge der Tasks bleibt erhalten.
- Das System verwaltet mehrere globale Queues, für einfache Nebenläufigkeit werden meistens diese verwendet.



```
let queue1 = DispatchQueue(label: "concurrent-queue", attributes: .concurrent)

let queue2 = DispatchQueue.global()
let queue3 = DispatchQueue.global(qos: .userInitiated)
```

# OperationQueue

- Zusätzliche API basierend auf DispatchQueues. Queues werden nicht mit Closures gefüllt, sondern mit Operation-Objekten.
- Dadurch bieten sich weitere Möglichkeiten wie Abhängigkeiten zwischen Operations, Prioritäten oder Cancellation

```
let queue = OperationQueue()
let op1 = MyCustomOperation()
let op2 = BlockOperation {
    print("op1 done")
}
op2.addDependency(op1)
queue.addOperation(op1)
queue.addOperation(op2)
```

# Race Conditions & Thread Safety

- Als **Race Condition** (Wettlaufsituation) wird der Umstand bezeichnet, dass das Ergebnis einer Operation vom zeitlichen Verhalten von Einzeloperationen abhängt.
- Diese treten insbesondere auf, wenn mehrere Threads den gleichen Speicher bearbeiten.
- Als **Thread Safety** (Threadsicherheit) wird die Eigenschaft einer Softwarekomponente bezeichnet, gemeinsamer Speicher nur so zu verändert, dass keine ungewünschte oder unerwartete Effekte auftreten.
- Swift ist im Allgemeinen **nicht** thread-safe.

# Race Condition Beispiel

Read «counter»

R = 0

Addition R + 1

R = 1

Write «counter»

Counter = 1

Read «counter»

R = 1

Addition R + 1

R = 2

Write «counter»

Counter = 2



```
var counter = 0
for _ in 0 ..< 2 {
    counter += 1
}
print(counter)
```

# Race Condition Beispiel

Read «counter»  
R = 0

Addition R + 1  
R = 1

Write «counter»  
Counter = 1

Read «counter»  
R = 0

Addition R + 1  
R = 1

Write «counter»  
Counter = 1



```
var counter = 0
DispatchQueue.concurrentPerform(iterations: 2) { _ in
    counter += 1
}
print(counter)
```

Mehrere Iteration vom  
DispatchQueues auf den  
Thread-Pool verteilen

# Race Condition Crash

- Race Conditions bei Datenstrukturen wie Arrays oder Maps führen oft nicht nur zu falschen Ergebnissen, sondern zum Crash der App



The screenshot shows a macOS application window with three colored window controls (red, yellow, green) at the top left. Inside the window, there is a code editor displaying Swift code. The code defines a mutable map and performs 1000 concurrent iterations on a dispatch queue. In each iteration, it sets the value of the map at index `i` to `i`. A red rectangular highlight covers the line of code `map[i] = i` and the error message that follows. The error message is: "Thread 12: EXC\_BAD\_ACCESS (code=1, address=0x800000000000...)".

```
var map: [Int: Int] = [:]
DispatchQueue.concurrentPerform(iterations: 1000) { i in
    map[i] = i  Thread 12: EXC_BAD_ACCESS (code=1, address=0x800000000000...
}
```

# MainQueue

- SwiftUI (und UIKit) sind ebenfalls nicht thread-safe. Um Inkonsistenzen zu verhindern, wird erwartet, dass das User Interface nur auf dem ersten Thread, der mit der App gestartet wurde, ausgeführt werden.
- Dieser wird als **Main Thread** bezeichnet. DispatchQueue und OperationQueue bieten beide eine globale Instanz, die alle Tasks auf diesem Thread ausführen.



```
DispatchQueue.main.sync {
    print(Thread.isMainThread) // true
}
OperationQueue.main.addOperation {
    print(Thread.isMainThread) // true
}
```

# Actors

- Um Thread Safety auch bei nebenläufiger und paralleler Ausführung zu ermöglichen, setzt Swift auf das **Aktorenmodell**.
  - Dieses wurde erstmals 1973 von Carl Hewitt beschrieben und wird auch in anderen Sprachen wie D oder Scala verwendet.
- In Swift werden Actors mit einem neuen vierten Typ erstellt und sind somit weder Klassen noch Structs.

enum

struct

class

actor

# Actor

- Actors sind wie Klassen **Referenztypen**, und unterstützen alle von Klassen und Structs Elemente wie Properties, Initializers, Methoden.
- Sie erlauben (noch?) keine Vererbung.
- Der Compiler garantiert, dass immer nur ein Task mit veränderbarem Speicher eines Actors interagiert.
- Aufrufe von aussen müssen deshalb mit **await** markiert werden und können somit aussetzen.



```
actor TemperatureLogger {  
    let label: String  
    var measurements: [Int]  
    private(set) var max: Int  
  
    init(label: String, measurement: Int) {  
        self.label = label  
        self.measurements = [measurement]  
        self.max = measurement  
    }  
}
```



```
let logger = TemperatureLogger(label: "Outdoors", measurement: 25)  
print(await logger.max)
```

# MainActor

- Ein **Task** wird in der Regel von einem Actor ausgeführt.
- Der spezielle globale Actor **MainActor** führt alle Tasks auf dem Main Thread aus. Somit können alle Tasks, die von diesem Actor ausgeführt werden, das User Interface ohne Risiko verändern.
- Mit der Annotation **@MainActor** kann eine Klasse, ein Struct oder einzelne Properties oder Methoden dem MainActor zugewiesen werden.



```
@MainActor  
class MyModel: ObservableObject {  
    @Published var title = "Foo"  
    @Published var counter = 0  
}  
  
MainActor.run {  
    someOtherModel.show = true  
}
```

# Tasks & SwiftUI

- SwiftUI bietet einen speziellen Modifier `.task`, der die Closure vor dem Erscheinen asynchron aufruft.
- Der Task wird auch abgebrochen, wenn die View frühzeitig entfernt wird.

```
● ○ ●  
class Loader: ObservableObject {  
    @Published var title = ""  
    func load() async {  
        // ...  
    }  
}  
  
struct TestView: View {  
    @StateObject var loader = Loader()  
    var body: some View {  
        Text(loader.title)  
        .task {  
            await loader.load()  
        }  
    }  
}
```

# Tasks & SwiftUI 2

- Mit dem Task-Initializer können asynchrone Tasks z.B. nach Interaktionen ausgeführt werden.
- Ein Task wird normalerweise von dem Actor ausgeführt, von dem er auch erstellt wurde
  - Im ersten Beispiel: MainActor
  - Tasks werden somit normalerweise nebenläufig, aber nur auf dem Main Thread ausgeführt
- Mit `Task.detached` wird ein Task unabhängig vom aktuellen Actor erstellt, somit können Subtasks auch parallel ausgeführt werden

```
struct TestView: View {  
    @StateObject var loader = Loader()  
    var body: some View {  
        VStack {  
            Button("Reload") {  
                Task {  
                    await loader.load()  
                }  
            }  
            Button("Reload") {  
                Task.detached {  
                    await loader.load()  
                }  
            }  
        }  
    }  
}
```

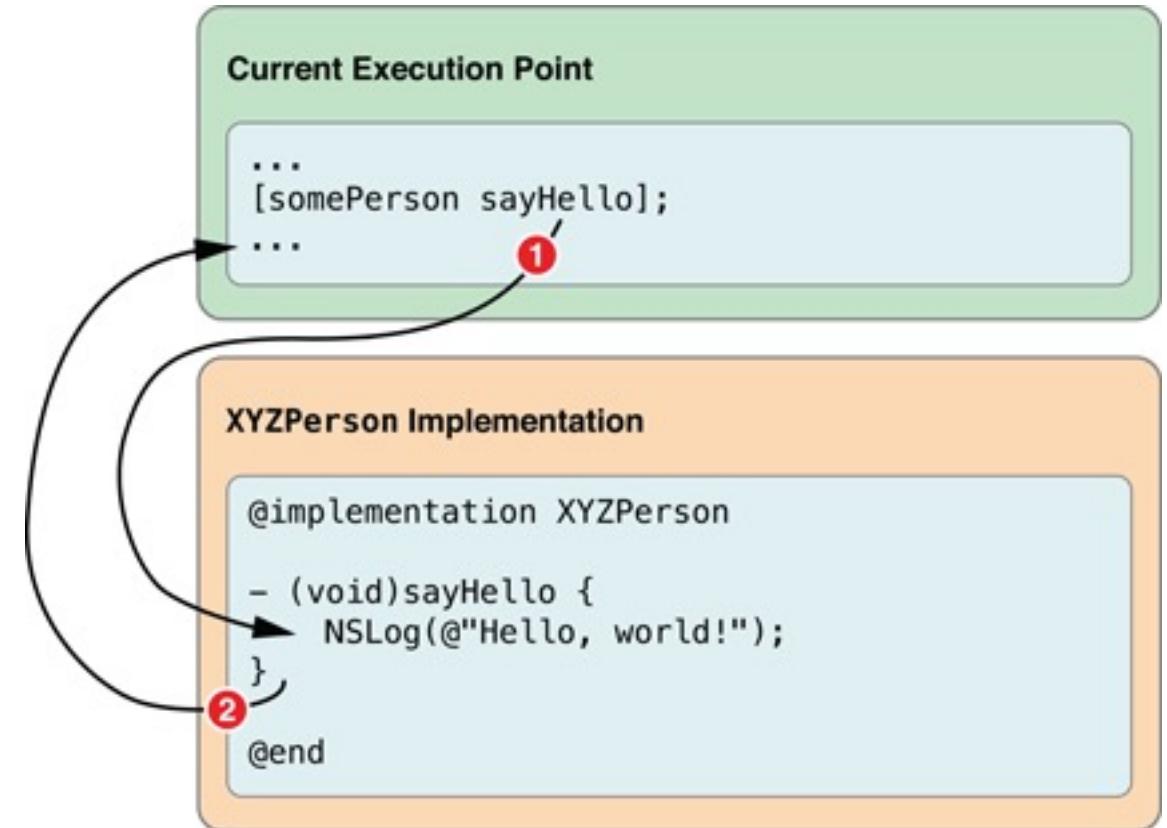
# Wofür eigene Actors einsetzen?

- Wie bei der Unterscheidung von Klassen und Structs, gibt es auch zum Einsatz vom Actors keine allgemeingültige Formel
- In Betracht zu ziehen sind Actors dann, wenn Objekte von mehreren Threads referenziert und bearbeitet werden sollen

```
public actor TrackerManager: ObservableObject {  
    static let shared = TrackerManager()  
  
    @MainActor @Published  
    private(set) var syncError: Error?  
  
    @MainActor  
    private func setSyncError(_ error: Error?) {  
        self.syncError = error  
    }  
  
    var trackers: [Trackers]  
  
    func syncTracker(_ tracker: Tracker) async {  
        // ...  
    }  
}
```

# Message Passing

- Vor langer Zeit (Objective-C): Jeder Methodenaufruf eine "Nachricht", die an ein anderes Objekt geschickt wird
- Actor Model: Nachrichten werden verschickt («Message Passing») und abgearbeitet, es gibt keinen Shared State



<https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html>

# Sendable

- Nur Objekte, die **sendable** sind, können von einem Actor zu einem anderen übermittelt werden.
- In der Regel trifft das auf alle Structs zu, die aus einfachen Typen kombiniert wurden und auf Klassen, die nur let-Properties enthalten.
- Für komplexe, dynamische Inhalte muss das Sendable-Protokoll erfüllt werden. (schauen wir nicht genauer an.)

# Distributed Actors

- Die Swift-Community arbeitet zur Zeit an einer Erweiterung, mit der Actors auf einem Cluster eingesetzt werden können, um Arbeit auf mehreren Servern korrekt verteilen zu können.
- Mit dem Proposal wird, ein neues Keyword eingeführt, voraussichtlich **distributed actor**.

```
●●●  
distributed actor Worker {  
    var data: SomeData  
    distributed func work(item: String) -> WorkItem.Result {  
        // ...  
    }  
}  
  
let first = ActorSystem("FirstNode") { settings in  
    settings.cluster.enable(host: "127.0.0.1", port: 7337)  
}  
let second = ActorSystem("SecondNode") { settings in  
    settings.cluster.enable(host: "127.0.0.1", port: 8228)  
}  
  
first.cluster.join(host: "127.0.0.1", port: 8228)
```

# Server-side Swift

- Swift ist eine allgemeine Programmiersprache und kann nicht nur für die App-Entwicklung eingesetzt werden.
- Vorteile von Swift als Server-Sprache
  - Kleiner Memory-Footprint
  - Schneller Startup
  - Gute und deterministische Performance
- SwiftNIO
  - Event-driven Low-Level Framework
  - <https://github.com/apple/swift-nio>
- Vapor / Perfect / Kitura
  - Opensource High-Level Frameworks für HTTP Server

# Vapor

- Verwendet SwiftNIO
- Bietet Ökosystem für Datenbank-Interaktionen, HTML-Output Verschlüsselungen und andere häufige Server-Aufgaben.
  - Kann natürlich (noch?) nicht mit Java/Spring oder anderen populären Server-Frameworks mithalten.
- Versucht stark, neuste Swift-Syntax zu ermöglichen, z.B. `async/await`.
  - Kitura setzt eher auf eine stabilere Entwicklung.



```
import Vapor

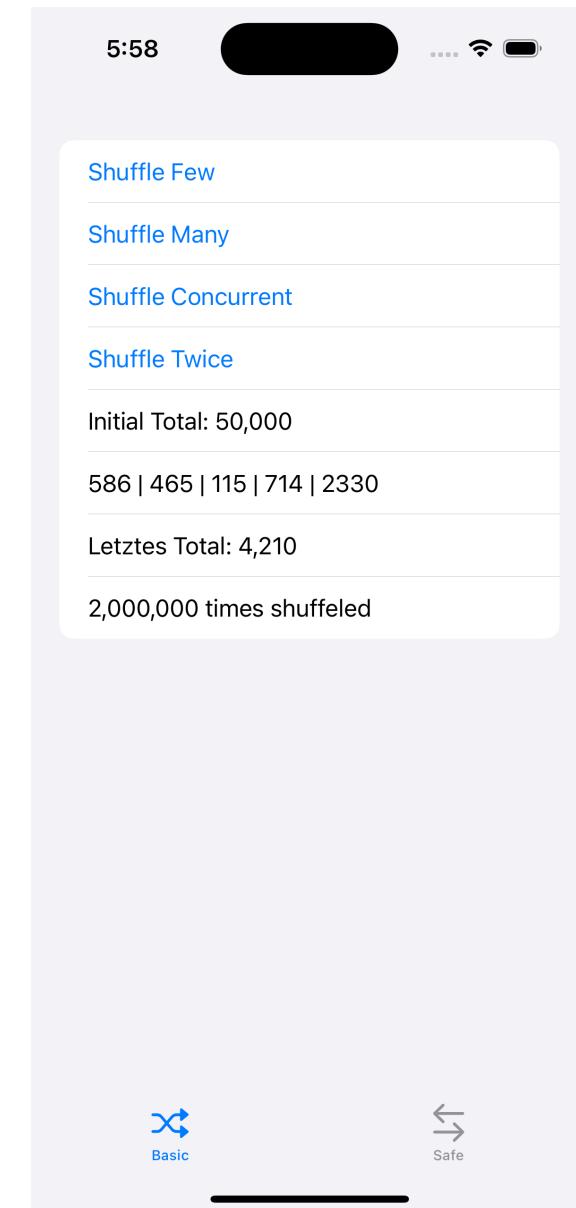
let app = try Application(.detect())
defer { app.shutdown() }

app.get("hello") { req in
    return "Hello, world."
}

try app.run()
```

# Ausblick: Übung 4

- Arbeiten mit Concurrency und Parallelism
- Achtung: Keine Schritt-für-Schritt Anleitung mehr



# Danke!

## Programmieren für iOS

