Praktikum – iOS-Entwicklung

Wintersemester 2018/2019

Prof. Dr. Linnhoff-Popien

Markus Friedrich, Christoph Roch

Swift

Einführung

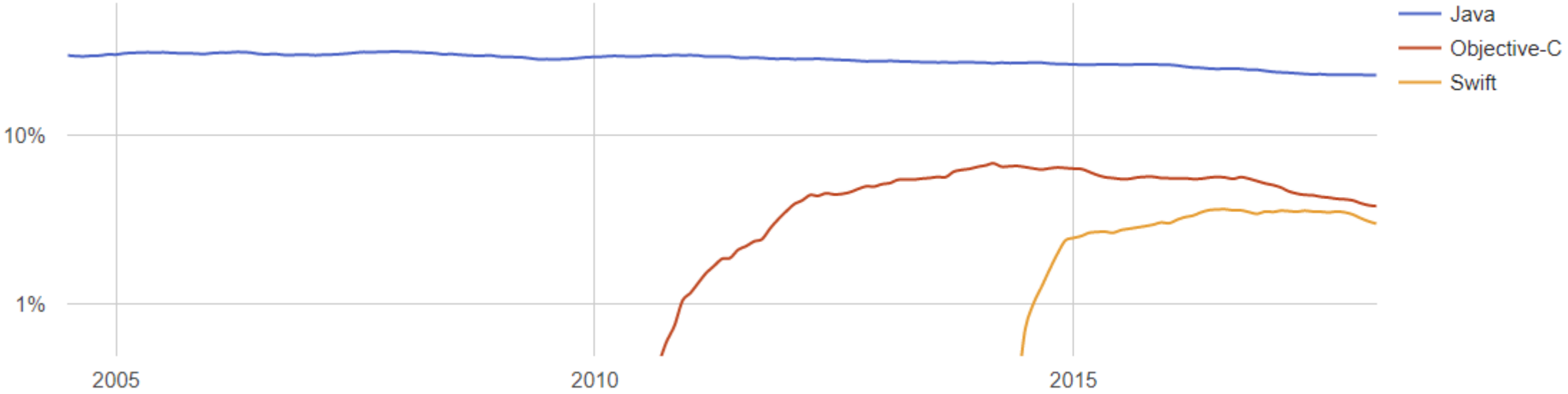
Objective-C vs Swift

Objective C

- Objekt-orientierte Erweiterung von C.
- Bedeutendste Programmiersprache für iOS (vor Swift).

Swift

- Moderne Sprache mit state-of-the-art Konstrukten.
- Vorgestellt auf der WWDC 14.



Objective-C vs Swift

Objective C

✓ Erleichtert u.U. die Arbeit mit low-level Code

✓ Integration von C, C++

✓ Wird von iOS Versionen <= 7 unterstützt

- Wenig elegante Syntax

- Abnehmende Popularität

- Performance (Dynamic Runtime) (https://www.infoq.com/articles/swift-objc-runtime-programming)

Swift

✓ Moderne Syntax

✓ Open Source (https://github.com/apple/swift)

✓ Performance (Static Runtime)

- Junge Sprache

Wir verwenden Swift.

Swift

Type-Inference

Closures

Functional Programming Patterns

Structs & Classes

ARC

Swift

https://github.com/apple/swift

Optionals

Tuples

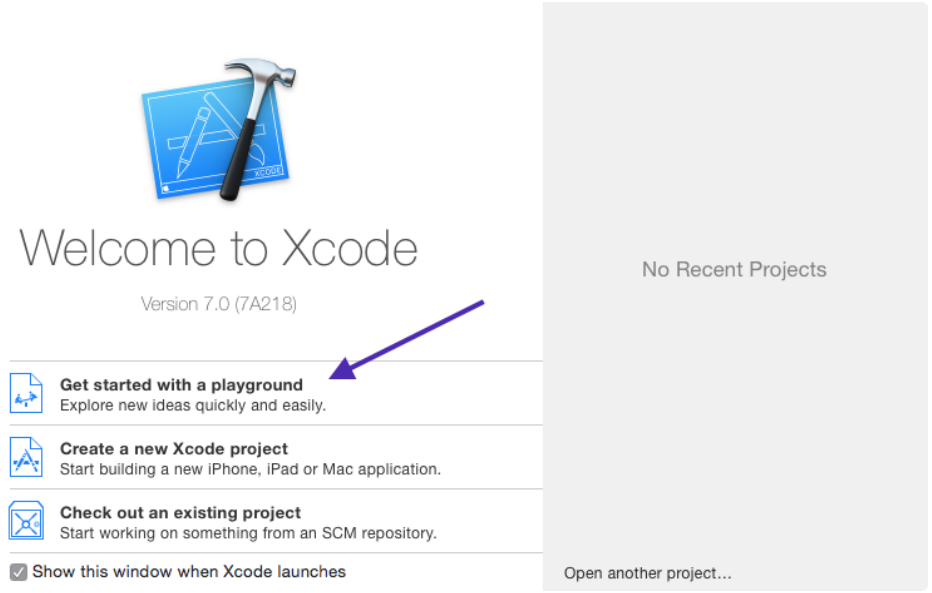
Protocol-oriented Programming

Swift

- Swift ist eine kompilierte Sprache und damit potentiell für performancekritische Anwendungen geeignet.

- Tool, um Swift kennenzulernen: **Playground**

- Betrachtete Sprachelemente:
  - Variablen, Konstanten und Collection Types
  - Verzweigungen und Schleifen
  - Funktionen
  - Strukturen, Klassen und Enumerationen
  - Optionals
  - Protokolle
  - Fehlerbehandlung



Variablen und Konstanten

- Variablentypen können implizit hergeleitet werden:

```
var a = 3
a = 5
a = 1.4 //Fehler
```

- Explizite Festlegung ist ebenfalls möglich:

```
var b: Double = 3.0
var c = 3, d = 8.4, e = false
```

- Konstanten sind Variablen vorzuziehen, falls ein Wert nur einmal zugewiesen wird:

```
let f = 5
f = 3 //Fehler
```

Basistypen:
[U]Int[8|16|32]
Float, Double
Bool
String

Funktionen

Verschiedene Funktionstypen:

- Globale Funktionen
- Closures (Anonyme Funktionen, oder „Lambdas“)
- Methoden
- Computed Properties

Funktionen - Template

```
func funktionsname (parametername1 : Typ1, parametername2 : Typ2) -> Ergebnistyp {
    //Code
    return ergebnis
}
```

- Beispiel:

```
func add (s1: Int, s2: Int) -> Int {
    return s1 + s2
}
let s = add(s1: 5, s2: 10)
```

- Für Funktionen, die nichts returnieren, kann der Ergebnistyp weggelassen werden.

Einschub: Wert vs Referenztyp

- Werttypen
  - Strukturen
  - Enumerationen
  - Basistypen (int, bool, double)
  - String
  - Array, Dictionary, Set, …
- Referenztypen
  - Klassen
  - (Funktionen und Closures)

Sind Werttypen als Funktionsparameter nicht langsam?
Kopiert wird erst bei Änderung (copy-on-write).

Funktionen - Parameter

In Swift gibt es unterschiedliche Parametertypen:

```
func f(p: Int) {…}                                  //Standard          => f(p: 5)
func f(_ p: Int) {…}                                //Unbenannt         => f(5)
func f(externP internP: Int {print(internP)} //Intern/Extern Name => f(externP:5)
```

Veränderlichkeit von Parametern:

```
func f(p: inout Int) {p = 6}

var a=5
f(p: &a)
print(a) //Ergebnis: 6
f(p: 5)  //Fehler (Variable muss übergeben werden)
```

=> Auf diese Weise können auch Referenztypen neue Objekte zugewiesen werden.

Funktionen – Parameter

- Optionale Parameter:

```
func add (s1: Int, s2: Int = 5) -> Int {
    return s1 + s2
}
let x = add(s1: 5, s2: 10) //Ergebnis: 15
let y = add(s1: 5)         //Ergebnis: 10
```

- Variable Anzahl von Parameter:

```
func add (_ ss: Int...) -> Int {
    var res = 0
    for s in ss {
        res = res + s
    }
    return res
}
let x = add(1,2,3,4,5) //Ergebnis: 15
```

Verzweigungen – If

- **if-else-else if**

```
var a = Int(arc4random_uniform(123))
if a < 3 { /*Do stuff*/
}
else if a <= 100 { /*Do stuff*/
}
else { /*Do stuff*/
}
```

Bool'sche Operatoren:
- And: &&
- Or: ||
- Not: !

- **guard**: Code wird nur ausgeführt, wenn alle Bedingungen erfüllt sind:

```
func foo() {
  guard let a = maybeBaby() else {return}
  print(a)
}
```

Verzweigungen – Switch-Case

Gegenüber If vorzuziehen, wenn sich viele unterschiedliche, klare Fälle definieren lassen.

```
let carBrand = „Mercedes“

switch carBrand {
    case „Mercedes“:
        print(„rich guy!“)
    case „Fiat“:
        print(„poor guy!“)
    default:
        print(„Car brand is invalid.“)
}
```

```
let value = 533

switch value {
    case 0…10:
        print(„small number!“)
    case 10…500:
        print(„big number!“)
    default:
        print(„Strange number!“)
}
```

Schleifen

- `for-in:`

```
for i in 1…1_000_000 {print(i)}
for i in (1…1_000_000).reversed() {print(i)}
for i in start..<end {print(i)}
for i in [1,2,3] {print(i)}
```

- `While (-let)`

```
var i = 0
while i < 50 {
  print(i)
  i += 1
}
```

```
var it = [1,2,3].makeIterator()
While let i = it.next() {
  print(i)
}
```

Sprünge
- `break`
- `continue`

- `repeat-while:` Bedingung wird erst am Ende geprüft.

Optionals

- Normale Variablen haben nie einen unitialisierten Wert (vgl. Java *null*).
- Optionals können nicht belegte Variablen beschreiben (nicht belegt: *nil*).
- Auch Werttypen können nicht belegt sein (!= Java).
- 2 Arten der Deklaration:

```
var op1: Int? //Gewöhnlicher Optional
var op2: Int! //Implicitly Unwrapped Optional
```

- Gewöhnliche Optionals:

```
let i1: Int = op1 //Fehler
let i2: Int = op1! //OK, expliziter unwrap
```

- Implicitly Unwrapped Optional:

```
let i3: Int = op2 //OK, impliziter unwrap
```

Immer gilt: Unwrapping ist beim auslesen, nicht aber beim Zuweisen notwendig.

Optionals

- Tests

```
let i1 = Int(„123“)
if i1 != nil { print(i1) } //nil Test
if let i2 = Int(„123“) { print(i2) } //if-let Test
```

- Optional Chaining

```
if let a = opt1, let b = a.do(), let c = b.do()
{ /*Do sth. with c*/ } //If-let-Kaskade
//Shorter:
if let a = opt1?.do()?.do() {/*Do sth. with a*/ }
```

- Nil Coalescing

```
let n1 = Int(„123“) ?? 0 // n1 = 123
let n2 = Int(„abc“) ?? 0 // n2 = 0
```

Eigene Datentypen erstellen

Drei Möglichkeiten, eigene Datentypen zu erstellen:

- Struktur
- Klasse
- Enumeration

Klassen

```
class CircleClass {

    var r: Double

    init(_ r: Double) {
        self.r = r
    }

    func getArea() -> Double {
        return r*r*3.14;
    }
}
```

- **Referenztyp**
- Vererbung => TypeCasting
- Ändern von Eigenschaften
- `Deinit() =>` Wird aufgerufen, wenn Instanz gelöscht wird

Strukturen

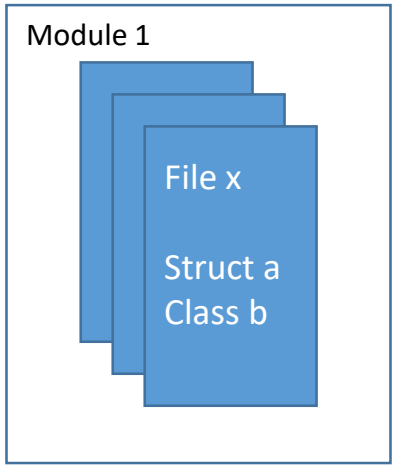
```
struct CircleStruct {

    var r: Double

    init(_ r: Double) {
        self.r = r
    }

    func getArea() -> Double {
        return r*r*3.14;
    }
}
```

- **Werttyp**
- Keine Vererbung => Kein TypeCasting
- Kein Ändern von Eigenschaften (nur mit `mutating`)
- Kein `Deinit()`

Einschub: Modularisierung

Ein **Modul** beinhaltet mehrere Klassen, Strukturen, … und kann per `import` in anderen Modulen zur Verfügung gestellt werden.

Module bestehen aus mehreren **Quelltext-Dateien**.



Zugriffsmodifizierer

- **Zugriffsmodifizierer** können auf Datentypebene (Struktur, Klasse, Enumeration) und auf Datentypelementebene benutzt werden (Methode, …).

Modifizierer	Beschreibung
open	Zugriff und Vererbung, Modulübergreifend
public	Zugriff, Modulübergreifend
internal	Zugriff und Vererbung, in eigenem Modul
fileprivate	Zugriff und Vererbung, in eigener Datei
private	Zugriff, eigene Klasse

- **Default:** internal

Strukturen vs Klassen

Warum Strukturen? Per-Value => Klarer bei multi-threaded Algorithmen.

Warum Klassen? Modellierung von komplexen, hierarchischen Datenstrukturen (=> Vererbung)

Und interessant in der Benutzung:

Klasse:

```
let c1 = CircleClass(5.0)
c1.getArea()
c1.radius = 1.0 //Funktioniert
let c2 = c1 //Referenzkopie
```

Struktur:

```
let c1 = CircleStruct(5.0)
c1.getArea()
c1.radius = 1.0 //Fehler
let c2 = c1 //Tiefe Kopie
```

Enumerationen

```
enum CarVendor {
  case fiat
  case mercedes
  case volkswagen

  init() {
    self = .fiat
  }
}

var vendor = CarVendor.mercedes
```

- **Werttyp**
- Keine Vererbung => Kein TypeCasting
- Keine Eigenschaften
- Kein `Deinit()`
- **Associated Values:** Jedem Enumerationswert können n Datentypen zugeordnet werden:

```
enum CarVendorEx {
  case fiat(String)
  case mercedes(Int)
  case volkswagen
}

var vendor =
CarVendorEx.mercedes(5)
```

```
switch vendor {
  case let .fiat(str):
    print(str)
  case let .mercedes(n):
    print(n)
  default:
    print(„invalid“)
}
```

Protokolle

- Schnittstellendefinition (Java, C#: `interface`, C++: `class` with pure virtual methods only):
  - Eigenschaften
  - Methoden
  - Init-Funktionen

```
protocol Movable {
  func move()
}

protocol Bike : Movable {
  func getWheels() -> [Wheel]
}
```

```
class SuperBike : Bike {
  func move() {
    print(„Move!“)
  }
  func getWheels() -> [Wheel] {
    return [Wheel(0), Wheel(1)]
  }
}
```

Protokolle

- Protokolle können als Variablen- und Parametertypen angegeben werden:

```
var x: P1 & P2 & P3
func foo(param: P1)
```

- Optionale Schnittstellenelemente:

```
@objc protocol Movable {
  func move()
  @objc optional func superMove() -> Int
}

var m : Movable = Bike()
m.superMove?() // Returniert nil
```

Optionale Methoden und Eigenschaften liefern immer ein Optional zurück.

Strings

- Unicode String Implementierung mit copy-on-write Optimierung:

```
let s1 = “Fiat“
let s2 = “““
  Mehrzeiliger, \
  Langer
  String
  “““
// \ bewirkt, dass kein Zeilenumbruch stattfindet
```

- Vergleich und Änderung:

```
let s1 = “Fiat1“
let s2 = “Fiat“
print(s1 == s2) // false
s2 += “1“
print(s1 == s2) // true
```

Strings

- Elementzugriff ähnlich zu Arrays:

```
let s1 = “Fiat 500“
let firstSpace = s1.index(of: " ") ?? s1.endIndex
let s2 = s1[..<firstSpace] //s2 == “Fiat”
//Schleife über Character-Elemente:
for c in s2 {print(c)}
```

- Count und Unicode:

```
let capitalA = "A"
print(capitalA.count)
// Prints "1"
print(capitalA.unicodeScalars.count)
// Prints "1"
print(capitalA.utf8.count)
// Prints "1"
```

```
let flag = "🇵🇷"
print(flag.count)
// Prints "1"
print(flag.unicodeScalars.count)
// Prints "2"
print(flag.utf8.count)
// Prints "8"
```

Collection Types - Tupel

- Geordnete Aufzählung fixer Größe von Daten unterschiedlichen Typs.
- Use-Case: Mehrere Rückgabewerte einer Funktion.

```
let tupel1 = (1, “test“, true)
let (x,y,z) = tupel1
let a = tupel1.1
```

- Tupel-Elemente können mit Namen versehen werden:

```
let tupel2 = (nr:1, str:“test“)
tupel2.nr = 2
```

- Tupel-Typen und Namen können auch bei der Deklaration festgelegt werden:

```
let tupel3 = (nr:Int, str:String)
tupel3.str = 3 //Fehler: Falscher Typ
```

Collection Types - Arrays

- Geordnete Menge von Daten fixen Typs.
- Als Variable deklariert => Elemente können hinzugefügt und entfernt werden. Als Konstante => Fixe Größe.

```
var array1 = [1,2,3]
array1.append(4)
array1+=[5,6,7]
array1.remove(at: 2)
array1.count
array1.isEmpty
```

- Elemente auslesen:

```
var i = array1[0]
var j = array1.first //j ist ein Optional!
```

Collection Types - Arrays

- **Array Slices**

```
var a = [1,2,3,4,5]
var s1 = a[2…4]  // s1 = [3,4,5]
var s2 = a[3…]   // s2 = [4,5]
var s3 = a[..<3] // s3 = [1,2,3]
```

- Slices sind vom Typ ArraySlice<T> (Generic) und damit nur Zeiger auf Start und Ende im Ausgangs-Array.
- Erst bei einer Wertzuweisung (z.B. `s3[0] = 5`) findet eine Kopie statt.
- Neues Array aus einem Slice (nicht immer nötig): `var a2 = Array(s2)`

- Viele Operationen auf Arrays benutzen **anonyme Funktionen** (Closures), die erst später behandelt werden (z.B. `sort`, `filter`, `map`,…).

Collection Types - Dictionaries

- Assoziativer Container (Key-Value).
- Initialisieren/hinzufügen:

```
var dict = [“1“:1, “2“:2, “3“:3]
dict[“1“] = 0
```

- Entfernen: `dict[“1“] = nil`
- Elementzugriff:

```
if let i = dict[“1“] {print(i)}
```

- Alternativ: Nutzung von `!` oder `??`
- Alternativ:

```
var i = dict[“1“, default:0]
print(i)
```

Key muss das Protokoll `Hashable` implementieren (also die Methode `hashValue`).

Collection Types - Dictionaries

- Über alle Schlüssel iterieren:

```
for key in dict.keys {print(key)}
```

- Über alle Werte iterieren:

```
for value in dict.values {print(value)}
```

- Über Schlüssel und Werte gleichzeitig iterieren:

```
for (key, value) in dict {
  print(„Key: \(key), Value: \(value)“)
}
```

Collection Types - Sets

- Ungeordnete Menge von Daten fixen Typs **ohne Doppelte Elemente**.
- Wichtige Methoden: `insert, remove, contains:`

```
var s = Set<String>()
s.insert(„1“)
s.insert(„2“)
s.insert(„1“) //Wird ignoriert
if s.contains(„2“) {
  print(„Set contains 2!“)
}
s.remove(„3“) //Returns nil
```

- Warum Sets? Immer dann, wenn man Mengenoperationen (`union`, `intersection`, `subtracting, isSubset, isSuperset, isDisjoint`) benötigt:

```
var s2 = s.union([„1“, „2“, „3“])
```

Collection Types - OptionSets

- Ungeordnete Menge von Daten fixen Typs **ohne Doppelte Elemente** für kombinierbare Optionen (Flags => C++ Enumeration mit Zweierpotenzen)
- Implementiert als Protokoll

```
struct CarTraits: OptionSet {
  let rawValue: Int
  init(rawValue: Int){
    self.rawValue = rawValue
  }
  static let DriverAssistant = CarTraits(rawValue: 1)
  static let GPS             = CarTraits(rawValue: 2)
  static let AirConditioning = CarTraits(rawValue: 4)
}

let car : CarTraits = [.GPS, .AirConditioning]
let isStandardCar = car.contains(.GPS) && car.contains(.AirConditioning)
```

Fehlerbehandlung – Try-Catch

- **Try-Catch** Syntax ermöglicht das Abfangen von Ausnahmen ausgelöst in mit **throws** deklarierten Funktionen.

```
enum CustomErrors : Error {
  case invalidVal (v : Int)
}

func foo(_ n:Int) throws -> Int {
  if n < 0 { throw CustomErrors.invalidVal(v:n)
}

  return n * n;
}
```

```
do {
  let x = try foo(3)
  let y = try foo(-3)
}
catch CustomErrors.invalidVal(let v){
  print(„Invalid Value: \(v).“)
}
catch {
  print(„Default error.“)
}
```

- Try ist immer notwendig, wenn mit throws deklarierte Methoden/Funktionen aufgerufen werden.
- Funktioniert nicht bei Überläufen oder Out-Of-Bounds Zugriffe (dort immer vorher Parametercheck).

Fehlerbehandlung – Try-Catch

- **`try?`** und **`try!`**

```
var a = try? foo(-5) // a ist von Typ Int? und a == nil
var b = try! foo(-5) // Programmende
```

- **`Defer`** (Java, C++ `finally`)

```
do {
  let tcpStream = open(„localhost“,8080)
  defer {
    tcpStream.close()
  }

  try tcpStream.send(„hello world!“)
} catch { print(„Fehler!“) }
```