Luzern, 22. Januar 2015

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
Technik & Architektur

Die Programmiersprache Swift

- Eine Einführung für Java-Entwickler

Ruedi Arnold ruedi.arnold@hslu.ch

- Neue Sprache von Apple für OS X und iOS
  - Interoperabel mit Objective-C, unterstützt Cocoa & Cocoa Touch
  - Version 1.0 September 2014, ist also eine sehr junge Sprache...

Quellen:
https://developer.apple.com/swift/resources/
http://en.wikipedia.org/wiki/Swift_%28programming_language%29

Paradigm(s)	multi paradigm (object-oriented, functional, imperative, block structured)
Designed by	Chris Lattner and Apple Inc.
Developer	Apple Inc.
Appeared in	2014
Stable release	1.1
Typing discipline	static, strong, inferred
Influenced by	Objective-C, Rust, Haskell, Ruby, Python, Scala, C#, CLU,[1] D[2]

Motivation für Swift

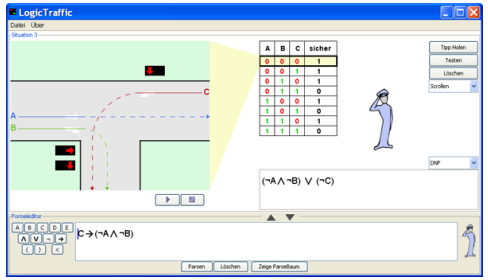
- Apple WWDC 2014: „Objective-C ohne C“
- Sicherer als Objective-C
  - Keine Pointer-Typen
  - Starke Typisierung, Generics
- Expressiver als Objective-C
  - D.h. Ausdrucksstärker, kompakterer Code möglich
- Weg von der Smalltalk-Syntax (Objective-C)
  - Kein `[obj doIt:arg]` und `@class` usw. mehr
  - „moderne“ Syntax (viel näher bei Java als Objective-C)

Ausgewählte Sprachkonstrukte von Swift

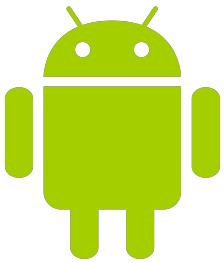
- In der Folge schauen wir ausgewählte Sprachkonstrukte/Syntax von Swift an
  - **Fokus: ...gibt‘s so nicht in Java (und/oder C)**
  - (Weniger im Fokus: IDE Xcode & iOS-Programmierung)
- Vorwarnung: Natürlich können wir nicht die ganze Sprache komplett besprechen, sondern nur kompakt einige relevante Aspekte
  - Swift ist eine sehr umfangreiche Sprache mit vielen Eigenheiten!..

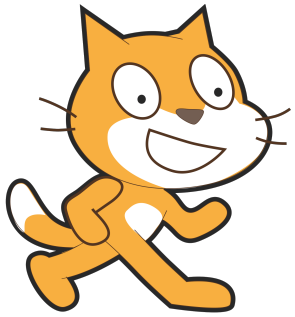
➔ Disclaimer: all Code run & tested with Xcode 6.1.1 ☺

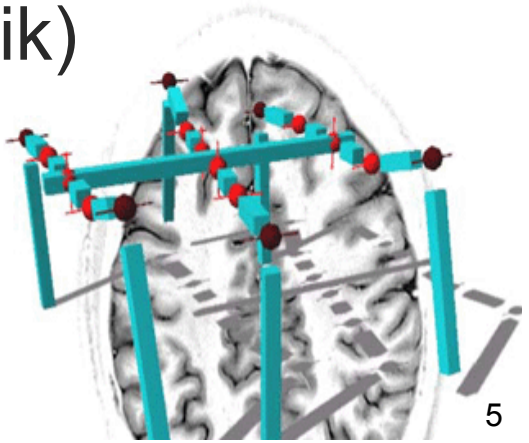
getShortCV("Ruedi Arnold")



- 2002 Dipl. Informatik Ing. ETH
- 2007 Dr. sc. ETH Zürich
- 2008-11 Ergon Informatik AG, Zürich
- 2009-11 EB Zürch (Kursleiter)
- 2011- Vorstand JUG.CH
- 2012- HSLU T&A (Dozent Informatik)

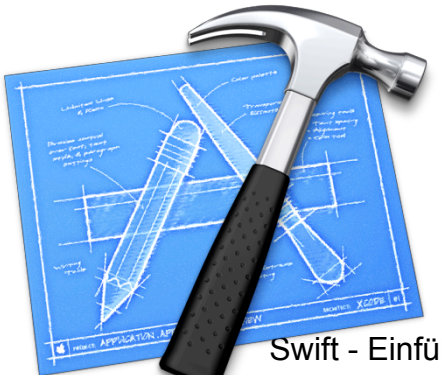


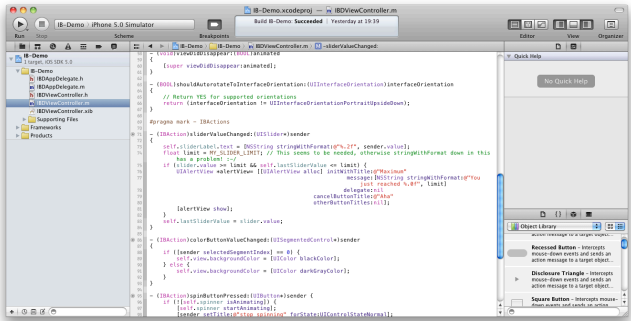




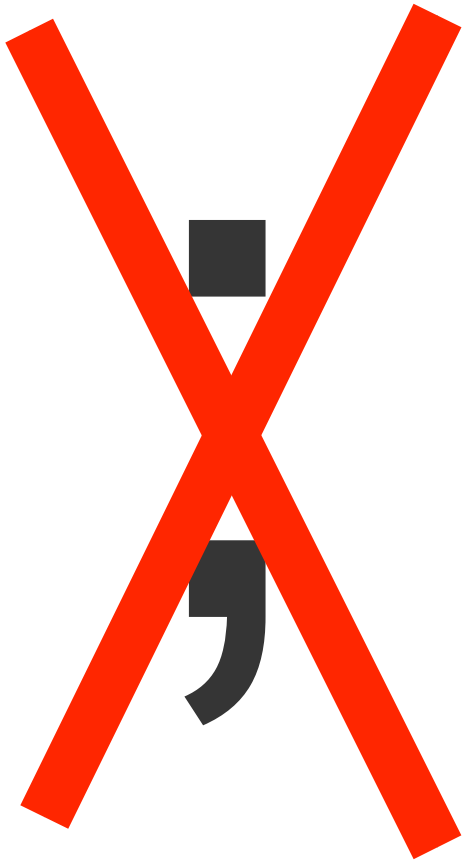
Werbeblock: iOS-Programmierkurs@HSLU ☺

- Weiterbildungskurs@HSLU
- 7 Abende à 4 Lektionen
- Inhalt: Native App-Entwicklung iOS
- Voraussetzung: Basics OOP
- Nächste Durchführung: September 2015
- Infos & Anmeldung
  https://www.hslu.ch/de-ch/technik-architektur/weiterbildung/fachkurse/app-entwicklung-fuer-ios/





Swift: Semikolon ist optional am Ende von Zeilen ☺



http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Variablen-Deklaration

Variablen-Deklaration: var & Typinferenz

- Schlüsselwort `var`
- Typangabe ist optional, wird ggf. abgeleitet vom zugewiesenen Wert

  ➔Typinferenz (type inference)
- Beispiele:

```
var myInt : Int = 42
var myOtherInt = 43
myInt = 1234
var myString : String = "Hello Swift"
var myDouble = 1.234
```

Hinweis: Semikolon ; generell nicht nötig am Ende von Zeilen (ausser bei mehreren Anweisungen auf einer Zeile)

Analog: Konstanten-Deklaration mittels let

- Schlüsselwort `let`
- Typangabe ist optional, wird ggf. abgeleitet vom zugewiesenen Wert: Typinferenz
- Beispiele:

```
let myConstInt : Int = 77
let myConstDouble = 66.6
let myFixString = "constant"
myFixString = "newValue"          // compile error!!
```

- Randbemerkung: Funktionsargumente sind per Default konstant (d.h. Typ let), [siehe später]

Deklaration Variablen & Konstanten: Swift vs. Java

- Swift: `<name> [: <Typ>]` vs. Java: `Typ <name>`
- In Swift **muss** jede Variable mit `var` deklariert werden
  - Folgende zwei Code-Zeilen kompilieren z.B. nicht:

```
text : String = "hallo";   // compile error!!
name = "Ruedi"             // compile error!!
```

- Bemerkungen:
  - Java kennt kein zu `var` analoges Schlüsselwort
  - Java keine keine derartige Typinferenz (ausser im Kontext von Lambda-Ausdrücken)

Primitive Datentypen: Int, Double, Bool

- Swift kennt direkt **keine (native) Primitivtypen** (analog zu Java oder C) wie `int`, `float` oder `double`
- Primitive Datentypen sind in Swift in entsprechenden C-Strukturen (struct) verpackt: `Int`, `Double`, `Float`
  - Hinweis: Strukturen werden in Swift (wie in C) by value übergeben (d.h. kopiert)
- Anwendungsbeispiel zur Illustration (inkl. Fehlermeldung aus der Entwicklungsumgebung Xcode):

```
var i : int = 1          ● Use of undeclared type 'int'
var j : Int = 2
var d : double = 3       ● Use of undeclared type 'double'
var e : Double = 4
var b : Bool = true
```

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Properties

Properties: Zugriff mit Dot-Syntax (x.property)

- Properties assoziieren Werte mit Instanzen einer Klasse, Struktur (`struct`) oder Enumeration (`enum`, nur computed Properties)
  - Analog zu Properties in Objective-C oder C#
- Swift unterscheidet folgend zwei Arten:
  - Stored Properties („Gespeicherte Werte“)
    - Unterart: Lazy stored Properties
  - Computed Properties („Berechnete Werte“)
- Zugriff über Dot-Syntax: `x.property`
  - Analog zu Zugriff auf Instanzvariable in Java

Stored Properties & Lazy Stored Properties

- Stored Property: Variable (`var`) oder Konstante (`let`) ist Teil einer Klasse oder Struktur (`struct`)
  - Benötigt keine spezielle Syntax, kein Schlüsselwort
  - Entspricht Java-Instanzvariabeln
  - Einfachster & „typischer“ Property-Typ:
- Unterart: Lazy Stored Properties, Schlüsselwort `lazy`
  - Wie Stored Property, aber Wert wird erst bei erstem Property-Zugriff ausgewertet und zugewiesen,
    - Praktisch wenn Property-Erstellung „teuer“ ist und Property evtl. gar nie gebraucht wird

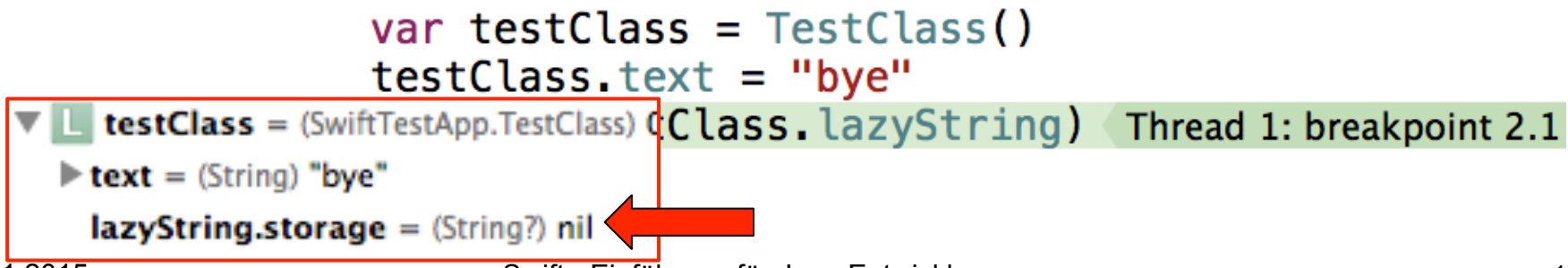
Beispiel: Stored Property & Lazy Stored Property

- Deklaration in TestClass.swift:

```
class TestClass {
    var text = "hi"                  // stored property
    lazy var lazyString = "test"  // lazy stored property
}
```

- Verwendung: Property-Zugriff mit dot-Syntax

```
var testClass = TestClass()          // create instance
testClass.text = "bye"               // set new value
println(testClass.lazyStr)           // first access -> create
```



Computed Properties: Werte „on the fly“ berechnen

- Getter- und Setter-Methoden werden explizit ausprogrammiert
  - Schlüsselworte `get` + `set`
    - Hinweis: Im `set`-Block darf nicht direkt dieses Computed Property gesetzt werden, da daraus ein nicht-terminierender rekursiver Setter-Aufruf resultieren würde! Compiler merkt‘s und warnt:

```
text = "recursive value"
⚠ Attempting to modify 'text' within its own setter
```

- setter-Block ist optional
  - Falls nicht vorhanden: Read-Only Computed Property

Beispiel: Computed Property

- Deklaration in TestClass.swift:

```
class TestClass {
    var message = "hi"                      // stored property (as seen)
    var text : String {                     // computed property
        get {
            return message + " 2"           // getter implementation
        }
        set {
            message = newValue + " 1"       // setter implementation
        }
    }
}
```

Achtung: Wir setzten `message` und nicht `text`!

`newValue` = Default Argumentname für den neu zu setzenden Wert (Apple nennt das „Shorthand Setter Declaration“)

- Verwendung von unserem computed Property:

```
var testClass = TestClass()
testClass.text = "bye"
println(testClass.text);                    // prints "bye 1 2"
```

Property Observers

- Properties können beobachtet werden
  - Schlüsselworte `willSet` und `didSet`

- Anwendungsbeispiel:

```
var value : Int = 7 {
        willSet {
                println("value will be set to \(newValue)")
        }
        didSet {
                println("value was set to \(value)")
        }
}
```

Bemerkungen zu Properties

- Properties können Zugriff auf Instanzvariablen kapseln, oder Konzept Instanzvariable auch „dynamisch“ erweitern (im Fall von computed property)
- Ähnliches Sprachkonstrukt gibt‘s auch in Objective-C
  - Schlüsselwort dort `@property`
    - Bsp. Deklaration: `@property strong NSString* name;`
  - Unterschied: Instanzvariable hinter einem Property ist bei Objective-C sichtbar, bei Swift nicht
- Ähnliches Konstrukt gibt‘s auch bei C#
  - ganz ähnliche Syntax für `get {...}` und `set {...}`
- Gemäss Doku gäb‘s auch „Type Properties“ (Schlüsselwort `class` oder `static`) : 

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Tupel-Typen

Tupel-Datentypen

- Tupel können beliebig viele Werte von beliebigen Datentypen enthalten
- Beispiel-Deklaration

```
let testTuple = (77, true, "Hi")
```

- Wert-Zugriff mittels Index oder „Tuple Decomposition“

```
var x = testTuple.0 + 1
var (number, flag, text) = testTuple
number++
var (_, justTheFlag, _) = testTuple
```

Index-Zugriff auf Tupel-Werte

Tuple Decomposition

Tuple Decomposition, bei der gewisse Tupel-Werte ignoriert werden (Schlüsselzeichen _)

Bemerkung: Tuple Decomposition: erinnert stark an das Matching von Prolog ☺

Tupel: benannte Elemente & Tupel als Rückgabewert

- Elemente von einem Tupel können benannt sein:

```
var anotherTuple = (id : 66, name : "Ruedi")
println("The id is \(anotherTuple.id)")
```

  - Zugriff wie auf Properties, d.h. mittels Dot-Syntax

- Tupel sind beispielsweise praktisch als Rückgabetyp von Funktionen: Funktionen können so beliebig viele Werte zurück liefern! (Siehe später)

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Optionals

Datentyp Optional: <DatenTyp>?

- Optional-Typen bedeuten, dass es sein kann, dass kein Wert vorhanden ist, analog zu Optionals in Java. Für Optional-Typen gilt also immer:
  - Es gibt einen Wert und er ist x
  - Es gibt keinen Wert (aber ein „Optional-Objekt“)
- (M)eine Motivation für Optionals in Swift:
  - Nicht-optionale Datentypen können nicht nil sein, z.B.:

```
var name : TestClass = nil
```

Type 'TestClass' does not conform to protocol 'NilLiteralConvertible'

    - Klassen, structs und enums werden als (weitgehend) gleichwertig betrachtet und müssen immer Wert != nil haben

Beispiel für Optional, inkl. Forced Unwrapping: !

```
let convertedNumber = "1234"
let convNumb : Int? = "4321".toInt()
```

- Methode `toInt()` der Klasse `String` liefert `Int?` zurück, d.h. einen Optional vom Typ Int
- Korrekter Zugriff auf Optional-Wert

```
var convNumb : Int? = "1234".toInt()
if convNumb != nil {
      var result = convNumb! + 2
}
```

  - „Forced Unwrapping“: <OptionalTyp>!
    - Holt Wert „heraus“, Schlüsselzeichen: !
    - Laufzeitfehler falls Optional keinen Wert hat

Implicitly Unwrapped Optionals: <Typ>!

- Sind Optionals, die immer einen Wert haben (sollten)...
  - Können ohne „unwrapping“ direkt verwendet werden
  - Zweck: „Bequemlichkeit“ (Objective-C & „alte“ APIs)
    - Viele API-Methoden haben Parameter, die nil sein können...
- Anwendungsbeispiel aus der Swift-Doku von Apple:

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation mark

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation mark
```

Optional Chaining vs. Forced Unwrapping

- Forced Unwrapping (!-Konstrukt) produziert Laufzeitfehler, falls Optional keinen Wert hat
- Alternative: Optional Chaining (Konstrukt „?.“)
  - Idee: Falls Wert von Optional ohne Wert abgefragt wird, kommt `nil` zurück und Anfragen (Property, Methode) darauf liefern wiederum `nil` zurück: „nettes“ Verhalten!
    - Wie „Safe Navigation Operator“ (?.) von Groovy
    - Hinweis: Das war bei Objective-C schon immer so, dass ein Methodenaufruf auf nil keinen Fehler à la NPE produziert, mit Optionals und Optional Chaining kann nun dieses Verhalten auch in Swift erreicht werden

Apple-Beispiel für Optional Chaining: ?.

- Vorbereitung: Zwei Klassen

```
class Person {
        var residence: Residence?  // note: optional type
}

class Residence {
        var numberOfRooms = 1
}
```

- Optional Chaining im Einsatz:

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
        println("John's residence has \(roomCount) room(s).")
} else {
        println("Unable to retrieve the number of rooms.")
}
```

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Spezielle Operatoren

Range Operatoren: ... und ..<

- In Swift können (numerische) Wertebereiche mit Bereichsoperatoren definiert werden
- Zwei Varianten:
  - **Closed Range Operator: `a...b`**
    - Alle Werte von a bis b, inkl. a und b
  - **Half-Open Range Operator: `a..<b`**
    - Alle Werte von a bis b, inkl. a und exklusiv b

Code-Beispiel Half-Open Range Operator: ..<

```
let names = ["Anna", "Alex", "Loana", "Peter"]
for i in 0..<names.count {
     println("Person \(i + 1) heisst \(names[i])")
}
```

- Range Operatoren können z.B. mit für Iterationen über Wertebereiche (Schleifen) verwendet werden
  - Half-Open: exkl. rechte Grenze, d.h. passend z.B. für 0-basierte Datenstrukturen wie Arrays

Nil Coalescing Operator: ?? („Elvis Operator“)

- Entpackt einen Optional-Typ oder liefert einen Default-Wert zurück, falls der Optional keinen Wert hat

`a ?? b` ist äquvalent zu `a != nil ? a! : b`

  - Coalescing (EN) = Verschmelzen, Vereinigen
  - Analog zum „Elvis Operator“ (?:) von Groovy

- Anwendungsbeispiel:

```
let defaultColorName = "red"
var userDefinedColorName: String? // defaults to nil var

//...

colorNameToUse = userDefinedColorName ?? defaultColorName
```

Custom Operators: Eigene Operatoren definieren

- In Swift können eigene Operatoren definiert werden
  - 3 Typen: Prefix, Infix, Postfix
    - Schlüsselwörter: `prefix`, `infix`, `postfix`
  - Präzedenz („Operatorvorrang“) und Assoziativiät (links oder rechts) kann festgelegt werden mittels Schlüsselwörter: `associativity`, `precedence`

- Wir schauen ein konkretes Beispiel an...

Beispiel: Eigener Operator

- Deklaration + Implementierung:

```
infix operator +* {associativity left precedence 140 }
func +* (a: Int, b: Int) -> Int {
        return (a + 1) * b;
}
```

  – Im „global scope“ (d.h. ausserhalb von Klassen-Impl.)

- Anwendung von unserem eigenen +*-Operator:

```
var result = 4 +* 5
println("result = \(result)")
```

  – Gibt aus: `result = 25`

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Funktionen

Funktionsdeklaration in Swift

- Muster der Grundsyntax:

```
func <name>(<arg>: <type>) -> <returnType> { ... }
```

  - Schlüsselwort `func`
  - Argumente nach Funktionsname in `()`
    - Mehrere Argumente durch Komma getrennt
  - Rückgabetyp nach `->`
    - Ist optional, d.h. kein `-> <returnType>` falls Funktion nichts zurückliefert

Funktion ohne Argument & ohne Rückgabe

- Methodenimplementierung in Klasse `SwiftTest`

```
func printHello() {
        println("Hello Swift")
}
```

- Aufruf aus Klasse `SwiftTest`

```
printHello()
var myInstance = SwiftTest()
testSwift.printHello()
```

  – Hinweis: Syntax Funktionsaufruf analog zu Java

Einfache Beispielfunktion inkl. Aufruf

- Ein Argument, ein Rückgabetyp (in Klasse `SwiftTest`)

```
func sayHello(personName: String) -> String {
      let greeting = "Hello, " + personName
      return greeting
}
```

- Aufruf aus Klasse `SwiftTest`

```
var result = sayHello("Swift")
var testSwift = SwiftTest()
var fromOtherInstance = testSwift.sayHello("Swift")
```

Funktion mit mehreren Rückgabewerten ☺

- Mit Hilfe von Tupels können Funktionen mehrere Werte zurück liefern:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
        var currentMin = array[0]
        var currentMax = array[0]
        for value in array[1..<array.count] {
                if value < currentMin {
                        currentMin = value
                } else if value > currentMax {
                        currentMax = value
                }
        }
        return (currentMin, currentMax)
}
```

- Bsp.:

```
let bounds = minMax([22, 4, 6, 42, 17, 1, 34])
println("min is \(bounds.min) and max is \(bounds.max)")
```

Parameter sind per Default konstant (let)

- „Praktische“ Eigenschaft von Swift: Parameter sind per Default konstant (`let`)
  - Guter Stil in Java?
  - Illustrationsbeispiel:

```
func doStuff(a: Int) {
    a = 7      Cannot assign to 'let' value 'a'
}
```

- Falls ein Parameter explizit veränderbar sein soll: mit `var` deklarieren
  - Illustrationsbeispiel:

```
func doStuff(var a: Int) {
    a = 7          // ok
}
```

Funktionsargument-Namen: Named Parameters

- Namen von Funktionsargumenten sind bisher nur „intern“ (d.h. im Funktionsblock) sichtbar, z.B.:

```
func sayHello(personName: String) -> String
```

  - `personName` ist nur im Funktionsblock sichtbar

- Namen von Funktionsargumenten können „extern“ (d.h. beim Funktionsaufruf) nicht verwendet werden. Falls dies möglich sein soll, muss externer Parametername bei Funktionsdeklaration gesetzt werden (= „benannter Parameter“). Beispiel:

```
func someFunc(extParamName localParamName: Int)
```

Funktionsaufruf mit benanntem Parameter

- Falls eine Funktion (extern) benannte Parameter hat, müssen diese beim Aufruf verwendet werden
  - Bsp. Deklaration:

```
func someFunc(extParamName localParamName: Int)
```

  - Bsp. Anwendung (Funktionsaufruf):

```
someFunc(extParamName: 7)      // ok
someFunc(7)                    // compile error
```

    - Hinweis: Die (bei der Deklaration festgelegte) relative Reihenfolge der benannten Parameter muss beim Funktionsaufruf beibehalten werden

Kurzform für benannte Parameter: #-Syntax

- Falls gelten soll Parametername extern = intern, kann vereinfachend nur ein Parametername angegeben werden mit # davor. Bespiel-Funktion:

```
func contains(#numList: [Int], value: Int) -> Bool {
        for cur in numList {
        if cur == value {
                return true
                }
        }
        return false
}
```

- Aufruf:

```
contains(numList: [22, 4, 6, 42, 17], value: 42)
```

Default Parameterwerte

- Parameter können Default-Werte haben ☺
  - Beispielfunktion:

```
func join(#s1: String, s2: String, joiner: String = " ") -> String {
        return s1 + joiner + s2
}
```

  - Funktionsaufrufe:

```
join(s1: "HSLU", s2: "T&A", joiner: "-")      // provide all 3 args
join(s1: "HSLU", s2: "T&A")                   // use default value
join(s1: "HSLU")                              // compile error
```

  - Parameter mit Default-Werten haben automatisch einen externen Namen (nämlich per internen)

Bemerkungen zu Default Parameterwerten

- Praktisches Konstrukt: Beim Aufruf müssen nur wirklich essentielle Parameter mitgegeben werden, für die andern können Default-Werte gesetzt werden
  - Methoden müssen nicht überladen werden und sich gegenseitig aufrufen

- Praktisch z.B. zur Objekt-Initialisierung
  - Behebt z.B. die Java-“Unschönheit“: viele ähnliche Konstruktoren mit je 1 Argument mehr u.ä.

inout-Parameter & call by reference

- Was wir kennen von Java: Parameter werde bei Methoden als Kopien mitgegeben: „call by value“
  - Auch bei Referenztypen: Übergeben wird eine Kopie von der Objektreferenz (und keine Kopie vom referenzierten Objekt!)
- inout-Parameter: Werte werden direkt übergeben, nicht als Kopie (d.h. call by reference)
  - Damit können Parameter-Werte ausserhalb eines Methoden-Bodys modifiziert werden (!)
    - ...magic?!

Beispiel: inout-Parameter & vertauschte Werte

- Deklaration von Methode mit inout-Parametern:

```
func swapStrings(inout a: String, inout b: String) {
      let temporaryA = a
      a = b
      b = temporaryA
}
```

- Verwendung von dieser Methode:

```
var x = "bye"
var y = "hello"
self.swapStrings(&x, b: &y)
println("\(x) - \(y)")
```

Hinweis: inout-Parameter müssen mit Präfix & übergeben werden (C-Syntax für Referenzierung: & liefert Speicheradresse der Variablen zurück)

  - Gibt aus: `hello - bye`
    - Achtung: d.h. also die Werte von x und y sind auch **ausserhalb** der Methode swapStrings vertauscht!

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Closures

Closure-Ausdrücke: analog zu Lambdas in Java

- Swift kennt Closure-Ausdrücke, eine Art „anonyme Funktionen“, die direkt inline im Code verwendet werden können (analog zu Lambda-Ausdrücken in Java)
- Der Typ von Closures entspricht einem äquivalenten Funktionstypen
  - Mit Closures-Ausdrücken können beliebig komplexe „Funktionen“ inline angegeben werden
- Syntax von Closure-Ausdrücken:

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

Beispiel: Closures-Ausdruck & Anwendung

```
var myClosure: (String) -> Bool = {
        (s: String) -> Bool in
                return countElements(s) <= 3
}
```

- Wie „normale“ Variable: `<name>: <Typ> = <value>`
  - „Code as Data“, „First class cititzens“ ☺

- Anwendung von unserem Closure-Ausdruck:

```
let text = "swift"
if myClosure(text) {
        println("'\(text)' is too short")
} else {
        println("'\(text)' is long enough")
}
// prints "'swift' is long enough"
```

Aufruf wie bei Funktion

Closure-Ausdrücke sind „capturing“

- Closures in Swift sind „capturing“ (wie Lambdas in Java), d.h. Variablen und Konstanten vom umgebenden Kontext sind bekannt und können verwendet werden
  - Unterschied zu Java: Werte müssen nicht konstant (resp. „effectively final“) sein wie in Java, sondern können in Swift modifiziert werden. Code-Bsp:

```
var c = 77
var myTrue: () -> Void = {
       () -> Void in c = 45          // ok
}
```

    - D.h. der Kontext von einer Closure muss in Swift erhalten bleiben, da Variablen schreibend verwendet werden können

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Klassen & Instanziierung

Klassen-Deklaration und -Instanziierung

- Beispielklasse:

```
class MyClass : MyBaseClass {
        var name = "Test"
        var id: Int

        init(id: Int) {
                self.id = id
        }
}
```

Stored Property mit initialem Wert

Stored Property ohne intialen Wert – muss in einer Initilizer-Methode gesetzt werden!

„Initilizer“-Methode, einfachste Form mit Schlüsselwort `init`

- Instanziierung eines MyClass-Objekts:

```
var myClass = MyClass(id: 7)
```

  - Ruft „automatisch“ die init-Methode von MyClass auf
    - init: an sich normale Methode, ähnlich zu Java-Konstruktor

Bemerkungen zur Intitialisierung

- Konzept mit Intitializier-Methoden wurde grossteils von Objective-C übernommen
- Initialisierung kann auch an andere Intitializier-Methode delegiert werden, Apple spricht dann von „Initializer Delegation“
- Intitializier-Methoden werden unterschieden in 2 Typen:
  - Designated Initializers
    - Initialisieren alle Propertys
  - Convenience Initializers
    - Müssen schlussendlich einen Designated Initlializer aufrufen

```
init( parameters ) {
    statements
}
```

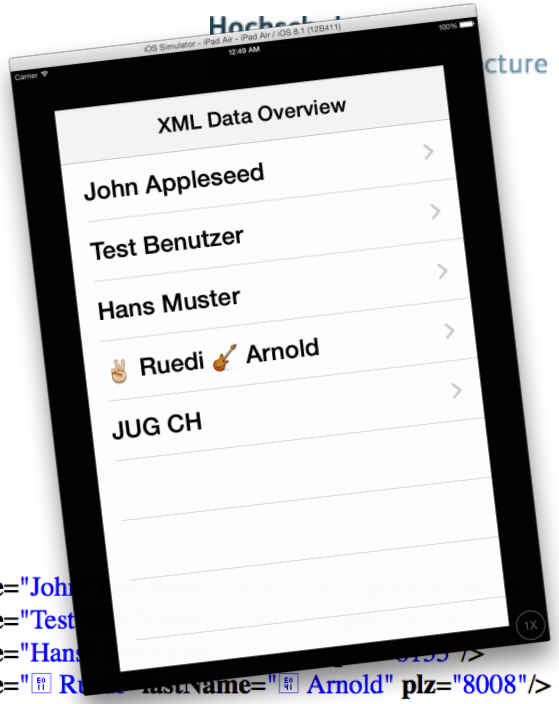
```
convenience init( parameters ) {
    statements
}
```

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Entwicklung iOS-App

-> Demo-App in Xcode

- Aufgabe: XML-Datei im Internet holen, parsen und darstellen
  - http://wherever.ch/hslu/iOSNameData.xml




```
<iOSNameData>
  <Entry firstName="John" lastName="Appleseed" plz="8135" />
  <Entry firstName="Test" lastName="Benutzer" plz="8135" />
  <Entry firstName="Hans" lastName="Muster" plz="8135" />
  <Entry firstName="Ruedi" lastName="Arnold" plz="8008" />
  <Entry firstName="JUG" lastName="CH" plz="8135" />
</iOSNameData>
```

```
// MARK: - XML parsing

func parser(parser: NSXMLParser!, didStartElement elementName:
    String!, namespaceURI: String!, qualifiedName qName:
    String!, attributes attributeDict: [NSObject : AnyObject]!)
    {
    if elementName == "Entry" {
        let firstName = attributeDict["firstName"] as String
        let lastName = attributeDict["lastName"] as String
        self.names.addObject(firstName + " " + lastName)
    }
}
```

```
@IBOutlet weak var myNameLabel : UILabel!
```

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetail" {
        if let indexPath = self.tableView.indexPathForSelectedRow() {
```

Fazit / Bemerkungen iOS-App-Entwicklung mit Swift

- „iOS-Basics“ bleiben (wie in Objective-C)
  - Wichtige Basiskonzepte: Delegates, ViewControllers, ...
  - Storyboards inkl. @IBOutlet und @IBAction
  - ARC: weak & unowned Variables
- Neues / auffälliges
  - Viele „forced unwrapped Optionals“ in API-Methoden
  - 1 Datei/Swift-Klasse vs. Header- & Impl.-Datei in ObjC
  - Neue Typen Any & AnyObject
  - ...

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

usw.

...und noch viel mehr! (I)

- Neben dem gesehenen bietet / unterstützt Swift noch viel mehr Sprachkonzepte und –konstrukte, z.B.:
  - Generics (analog zu Java)
    - z.B. struct Stack<T>, auch für generische Methoden usw.
  - Subscripts für Klassen, Enums & Structs
    - Ermöglicht Index-Zugriff auf beliebige Objekte
  - Zugriffskontrolle: private, internal, public
    - ähnlich wie in Java, „internal = Projekt“
  - Funktionale Programmierung (analog zu Stream@Java8)
    - filter, map & reduce für Arrays (exklusiv diese 3 Methoden!)

...und noch viel mehr! (II)

- Geschachtelte Funktionen
- Extensions
  - Erweitern Klassen dynamisch um neue Methoden (ähnlich zu Kategorien in Objective-C)
- Automatic Reference Counting (ARC)
  - Schlüsselwörter `weak` und `unowned`
  - Übernommen von Objective-C
- Casting, Schlüsselwort: `as`
- Protokolle (= Interface in Java)
  - Übernommen von Objective-C
- ...

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Fazit & Ende

Mein Fazit zu Swift

- Spannende Sprache, interessante Konzepte ☺
  - teilweise mittelfristig auch in Java? (z.B. Tupel, Named-Params?, Elvis-Operator, ...)
- iOS Zukunft: Objective-C vs. Swift ?
  - Aktuell Unsicherheit: Swift noch nicht ganz fertig (IDE, Compiler, usw.), Objective-C noch voll unterstützt

➔ Apple hat & nutzt die Macht!..

Persönliche Bemerkung: Ich unterrichte iOS seit iOS 3 (2009) – langweilig wird‘s da nicht... ;-)

https://vinnylanni.files.wordpress.com/2012/12/thats-all-folks.jpg

Das war der heutige Blick über den Java-Tellerrand...

Danke, dass ihr dabei gewesen seid! ☺

http://www.mcstudy.de/newsletter/archiv/images/bild_tellerrand.jpg

...Fragen?