

# Programmieren fürs iOS

## 1. Swift (Crash Course) + Xcode



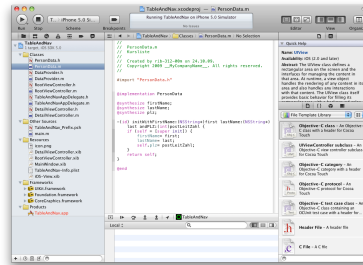
# Inhalt

- Swift (Crash Course)
  - Variablen, Konstanten & Typinferenz
  - Datentypen inkl. Optionals & Tupel
  - Properties
  - Kontrollfluss (Bedingung, Schleifen, Auswahl)
  - Funktionen, inkl. benannte Parameter & Default Werte
  - Klassen: Instanziierung & -Initialisierung
- Xcode

# "iOS Programmierung"

## Tools

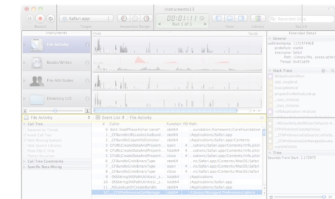
Xcode 14 mit Swift 5.7 😊



Xcode



iOS Simulator



Instruments

## Frameworks

Swift Standard Lib

SwiftUI /  
UIKit

## Languages & Runtime

Swift: `myObj.doItWith(aParam)`

# ...zu diesem "Crash Course"

- Rel. viel Stoff in diesem Foliensatz
  - Idee: "Kick-Start: Swift Essentials", damit sie nachher selber loslegen können...
  - Nicht alle Code-Bsp. in Xcode / live Demos
    - Die meisten Dinge werden später im Modul auftauchen
      - Ggf. selber nochmals anschauen & ausprobieren...
- Denken sie mit, stellen sie Fragen bei Unklarheiten, usw.!
- So profitieren sie am meisten 😊




A new programming language for iOS and OS X.

- Rel. neue Sprache für macOS und iOS
    - Interoperabel mit Objective-C
    - Unterstützt Cocoa & Cocoa Touch
    - Version 1.0: September 2014
      - d.h.: junge Sprache!
    - OS seit 2.2 (Apache License 2.0) 😊
- Im Modul: Swift 5.7
- Inbegriffen bei Xcode 14 [beta]

Quellen:

<https://developer.apple.com/swift/resources/>

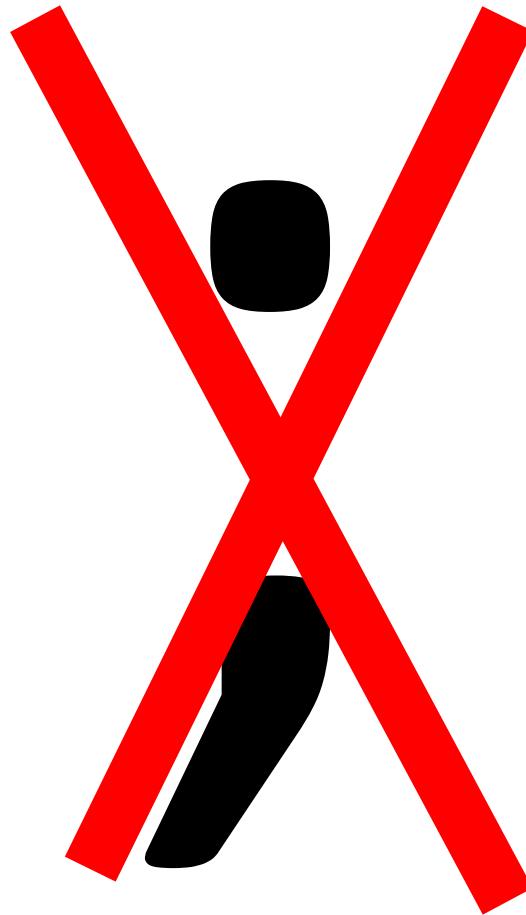
[http://en.wikipedia.org/wiki/Swift\\_%28programming\\_language%29](http://en.wikipedia.org/wiki/Swift_%28programming_language%29)

	<h1>Swift</h1> <p>Logo</p>
<b>Paradigm</b>	Multi-paradigm: protocol-oriented, object-oriented, functional, imperative, block structured, declarative, concurrent
<b>Designed by</b>	Chris Lattner, Doug Gregor, John McCall, Ted Kremenek, Joe Groff, and Apple Inc. <sup>[1]</sup>
<b>Developer</b>	Apple Inc. and open-source contributors
<b>First appeared</b>	June 2, 2014; 8 years ago <sup>[2]</sup>
<b>Stable release</b>	5.7 <sup>[3]</sup> / 12 September 2022; 1 day ago
<b>Preview release</b>	5.7 branch (5.8 and 6 coming next)
<b>Typing discipline</b>	Static, strong, inferred

# Motivation für Swift

- Apple WWDC 2014: "Objective C ohne C"
  - Sicherer als Objective C
    - Keine Pointer-Typen, Optionals
    - Starke Typisierung, Generics
  - Expressiver als Objective C
    - Ausdrucksstärkerer, kompakterer Code
  - Weg von der Objective C-Syntax
    - Kein `[obj doIt:arg]` und `@class` usw. mehr
- "Moderne", zeitgemässe Sprache

# Swift: Semikolon ist optional am Ende von Zeilen 😊



# Gute Apple Doku



Swift

The powerful programming language  
that is also easy to learn.

- Einstieg: <https://developer.apple.com/swift/>
  - Gute offizielle Quelle: "Language Guide" unter <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>



Swift

THE SWIFT  
PROGRAMMING  
LANGUAGE  
SWIFT 5.1

WELCOME TO SWIFT

LANGUAGE GUIDE

**The Basics**

Basic Operators

Strings and Characters

Collection Types

## The Basics

Swift is a new programming language for iOS, macOS development. Nonetheless, many parts of Swift will be developing in C and Objective-C.

Swift provides its own versions of all fundamental C and C++ types: `Int` for integers, `Double` and `Float` for floating-point values, `String` for textual data. Swift also provides powerful value types, collection types, `Array`, `Set`, and `Dictionary`, as described in this guide.

Like C, Swift uses variables to store and refer to values by an identifying name. Swift also

ON THIS PAGE ×

Constants and Variables  
Comments  
Semicolons  
Integers  
Floating-Point Numbers  
Type Safety and Type Inference  
Numeric Literals  
Numeric Type Conversion  
Type Aliases  
Booleans  
Tuples  
Optionals  
Error Handling  
Assertions and Preconditions





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

# Variablen, Konstanten & Typinferenz

# Variablen-Deklaration & 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)

# Konstanten-Deklaration: 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

# Var. & Konst.: Swift vs. Java

- Swift: `<name> [ : <Typ> ]`
- Java (anders rum): `Typ <name>`
- Swift: **jede** Variable (Konst.) mit `var` (`let`) deklariert
  - Folgende zwei Code-Zeilen kompilieren z.B. nicht:

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

- Bemerkungen:
  - Java kennt keine zu `var` & `let` analoge Schlüsselworte
  - Java kennt keine derartige Typinferenz (ausser im Kontext von Lambda-Ausdrücken, siehe z.B. PCP-Modul ;-)





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

# Datentypen

# Primitivtypen: Int, Double, Bool

- Swift kennt direkt **keine (native) primitiven Datentypen** (analog zu Java oder C) wie `int`, `float` oder `double`
- Primitive Datentypen sind in Swift in entsprechenden Strukturen (struct) verpackt:  
`Int`, `Double`, `Float`
  - Hinweis: Strukturen werden in Swift (wie in C) by value übergeben (d.h. kopiert)

# Bsp.: Int, Double, Bool

- 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
```

# z.B.: Struct Int

Structure

## Int

A signed integer value type.

- Beinhaltet
  - Initializers
  - Instanz- und Typ-Methoden
  - adoptierte Protokolle
    - (<https://developer.apple.com/documentation/swift/int>)
- Hinweise
  - Initializers sind "eine Art" Konstruktoren
  - Structs (und Enums) können Methoden haben
  - Protokoll (Swift/ObjC) = Interface (Java)



# Arrays

- In Swift typisiert: Generics
  - Wie in Java
- Beispiel-Typ: String-Array
  - Lange Version: `Array<String>`
  - Kompakte Version: `[String]`
    - Äquivalent, von Apple (und im Modul!) bevorzugt
- Anwendungsbeispiel inkl. Array-Literal:

```
let names : [String] = ["Anna", "Alex"]
```

  - Typ-Angabe hier natürlich nicht nötig (Typinferenz!)

# Dictionary

- Enthalten Schlüssel-Werte-Paare
  - Wie `Map<K, V>` in Java
- Beispiel-Typ: Dictionary mit String & Int
  - Lange Version: `Dictionary<String, Int>`
  - Kompakte Version: `[String: Int]`
    - Äquivalent, von Apple (und im Modul!) bevorzugt
- Anwendungsbeispiel inkl. Dictionary-Literal:

```
let ages : [String: Int] = ["Ruedi": 21, "Anna": 23]
```

  - Typ-Angabe hier natürlich auch nicht nötig (Typinferenz!)

# Bemerkung zu Collection-Types

- Veränderbarkeit (Mutability): per Default gegeben (Grösse & Inhalt)
  - Unveränderbar falls als konstant (let) deklariert
- Neben Arrays & Dictionarys gibt's auch noch Sets (Ungeordnete Menge ohne Duplikate)
  - Inkl. Mengen-Operationen wie `intersect`, `subtract` oder `union` 😊

# The Swift Standard Library

The Swift standard library defines a base layer of functionality for writing Swift programs, including:

- Fundamental data types such as `Int`, `Double`, and `String`
- Common data structures such as `Array`, `Dictionary`, and `Set`
- Functions and methods such as `print(_:separator:terminator:)`, `sorted()`, and `abs(_:)`
- Protocols that describe abstractions such as `Collection` and `Equatable`.
- Protocols used to customize operations that are available to all types, such as `CustomDebugStringConvertible` and `CustomReflectable`.
- Protocols used to provide implementations that would otherwise require boilerplate code, such as `OptionSet`.

<https://developer.apple.com/reference/swift>

# Bem. zur Standard Library

- Definiert u.a. Basis-Typen wie `Int` und `String` und Collection-Typen wie `Arrays` und `Dictionaries`
- Die Swift Standard Library ersetzt viele Klassen aus dem Foundation Framework von Objective-C
  - Automatisches "Bridging", z.B. von `Array` nach `NSArray`, bzw. Cast möglich mittels `as`




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

# Optionals

# Datentyp Optional: <DatenTyp>?

- Optional-Typen heisst: es kann sein, dass kein Wert vorhanden ist
- Für Optional-Typen gilt also immer:
  - Es gibt einen Wert und er ist x
  - Es gibt keinen Wert (aber ein "Optional-Objekt")
- **Wichtig: Nicht-optionale Datentypen können nicht nil sein!**
  - d.h. Klassen, structs und enums werden als (weitgehend) gleichwertig betrachtet und müssen immer Wert != nil haben
  - z.B.: 

```
let myInstance : MyClass = nil
```

 Nil cannot initialize specified type 'MyClass'

# Bsp: Deklaration Int-Optional

- "Konstruktor" `Int(String)` liefert `Int?` zurück, d.h. einen Optional vom Typ `Int`

- Sinnvoll: Argument könnte ja keine Zahl sein...

- Kann also nicht einem `Int` zugewiesen werden:

```
let convNumb : Int = Int("1234")
```

• Value of optional type 'Int?' not unwrapped;

- Zuweisung möglich an Var vom Typ "Int-Optional":

```
let convNumb : Int? = Int("1234")
```





# Optionals & Unwrapping

- Ausgabe von `let convNumb : Int? = Int("1234")` ?  
`print(convNumb)`
  - Antwort: `optional(1234)`
- Q: Zugriff auf Optional-Werte?
- A: Forced-Unwrapping (Operator: **!**)
  - Bsp.: `let convNumb : Int? = Int("1234")`  
`let number : Int = convNumb!`



# "Sicheres" Auspacken

- Besser zuerst auf Wert testen beim Zugriff auf Wert von Optional:

```
let convNumb : Int? = Int("1234")
if convNumb != nil {
    let result = convNumb! + 2
    print(result)
}
```

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

# Optional Binding: if let

```
if let constantName = someOptional {  
    statements  
}
```

- Temporäre Bindung von einem Optional an eine Variable bzw. Konstante

– z.B.: 

```
let convertedNumber : Int? = Int("1234")  
if let number = convertedNumber {  
    print("The number is \(number)")  
} else {  
    print("No number available")  
}
```

# Implicitly Unwrapped Optionals: <Typ>!

- Sind Optionals, die immer einen Wert haben (sollten), Laufzeitfehler falls nicht
  - Können ohne "unwrapping" verwendet werden
  - Zweck: "Bequemlichkeit" (Objective-C & "alte" APIs), siehe später (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
```

# Hinweis zum Ausrufezeichen

- Achtung: ! hat im Kontext von Optionals also zwei Bedeutungen:

1. "Forced Unwrap"-Operator von einem Optional:

- Bsp.: `let value = optional!`

2. Typendeklaration von einem "implicitly unwrapped" Optional:

- Bsp.: `var text : String! = nil`

# Alternative zu unsicherem Auspacken: Optional Chaining

- 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 Methoden- oder Property-Aufruf auf `nil` keinen Fehler à la `NullPointerException` produziert, mit Optionals und Optional Chaining kann nun dieses Verhalten auch in Swift erreicht werden

# Beispiel für Optional Chaining: ?.

```
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 {  
    print("John's residence has \(roomCount) room(s).")  
} else {  
    print("Unable to retrieve the number of rooms.")  
}
```

# Nil Coalescing Operator: ??

- Entpackt einen Optional-Typ oder liefert einen Default-Wert zurück, falls der Optional keinen Wert hat
  - Coalescing (EN) = Verschmelzen, Vereinigen
  - Äquivalenter Code:
    - `a ?? b`
    - `a != nil ? a! : b`
  - Analog zum "Elvis Operator" (?:) von Groovy





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

# Datentyp Tupel

# 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  
let inc = number + 7  
var (_, justTheFlag, _) = testTuple
```

Index-Zugriff auf Tupel-Werte

Tuple Decomposition

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

# Tupel: Elemente-Namen & als Rückgabewert von Funktionen

- Tupel-Elemente können benannt sein:

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

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

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



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

# Properties



# Properties: Zugriff mit Dot-Syntax

- (Instanz-)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 folgende 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 Property: Variable (`var`) oder Konstante (`let`) ist Teil einer Klasse oder Struktur (`struct`)
  - Benötigt keine spezielle Syntax, kein Schlüsselwort
  - Entspricht Java-Instanzvariablen
  - 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

# Bsp.: Stored Property & Lazy

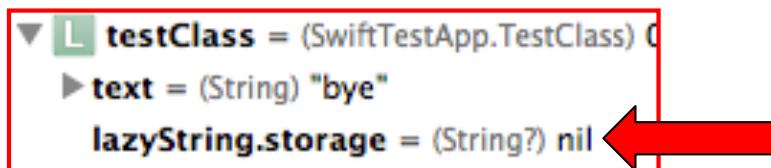
- Deklaration in TestClass.swift:

```
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  
print(testClass.lazyStr) // first access -> create
```

```
var testClass = TestClass()  
testClass.text = "bye"  
println(testClass.lazyString) Thread 1: breakpoint 2.1
```



```
testClass = (SwiftTestApp.TestClass) 0  
  text = (String) "bye"  
  lazyString.storage = (String?) nil
```

# 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 {  
        get {  
            return message + " 2" // getter implementation  
        }  
        set {  
            message = newValue + " 1" // setter implementation  
        }  
    }  
}
```

Achtung: Wir setzen  
message und nicht text!

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

- Verwendung vom computed Property:

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

# Property Observers

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

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

# Type Properties (static)

- Die bisher gesehenen (Instanz-)Properties gehören jeweils zu einer Instanz (Analog zu Instanzvariablen, z.B. in Java)
- Es gibt auch "Klassen"-Properties, diese heissen bei Swift **"Type Properties"**

- Schlüsselwort `static` für "normale" Properties
  - (oder `class` für in Subklassen überschreibbare "computed properties")
- Allg. Namen "Type-Properties" da auch für enums und structs!
- Bsp.:

```
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        get {return 42 }
    }
    class var overrideableComputedTypeProperty: Int {
        get {return 42 }
    }
}
```

# 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 { ... }`



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

# Kontrollfluss

# Bedingungen ohne Klammer

- In Swift gibt's analog zu den meisten anderen imperativen Programmiersprachen u.a. folgende Sprach-Konstrukte zur Steuerung des Kontrollflusses: if, for, while, switch-case
- Generell sind für die Bedingungen keine runden Klammern notwendig, wie z.B. bei Java

– Legaler Swift-Code:

```
if 4 < a {  
    print("smaller")  
}
```

# for-in & Range-Operator

- Beispiel: 

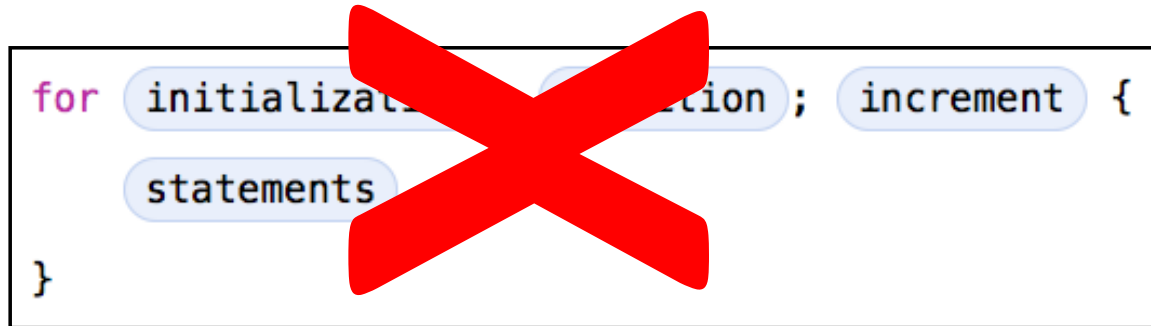
```
for i in 1...3 {  
    print("i = \(i)")  
}
```
- Bemerkungen zu obigem Code
  - Verwendet den "range operator" . . .
    - Praktisches Sprach-Konstrukt 😊
  - Ausgabe: 

```
i = 1  
i = 2  
i = 3
```



# KEIN "traditionelles" for

- "Old school", C-Style for-Schleifen gibt's in Swift nicht (mehr)



```
for (initialization; condition; increment) {  
    statements  
}
```

# for-in & Iteration über Arrays

- Code-Beispiel:

```
let names : [String] = ["Anna", "Alex", "Peter"]
for name in names {
    print("Hello, \(name)!")
}
```

- Ausgabe: Hello, Anna!  
Hello, Alex!  
Hello, Peter!

# for-in mit Tupeln

- Code-Beispiel mit einem Dictionary:

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
```

- Beispiel mit `enumerated()` auf Array:

– Hinweis: so gibt's "gratis" index-Variable

```
let names = ["Anna", "Alex", "Luana", "Peter"]
for (index, name) in names.enumerated() {
    print("\(index+1). \(name)")
}
```

# while & repeat-while

- Keine Überraschungen:

– while: `while` `condition` {  
    `statements`  
}

– repeat-while: `repeat` {  
    `statements`  
} `while` `condition`

# switch-case

```
switch some value to consider {  
    case value 1 :  
        respond to value 1  
    case value 2 ,  
        value 3 :  
        respond to value 2 or 3  
    default :  
        otherwise, do something else  
}
```

- Bemerkungen
  - Pro case mehrere Werte möglich
  - Kein impliziertes "Fall-through" (d.h, break i.a. nicht notwendig)

# Control Transfer Statements

- `continue`: nächster Schleifendurchgang
  - auch inkl. Labels (bei verschachtelten Schleifen!)
- `break`: Abbruch bei Schleife oder switch-case
  - auch inkl. Labels (bei Verschachtelung!)
- `fallthrough`: "Durchfallen" von zum nächsten case (erreicht Verhalten von Standard-C-switch-case)
- `return`: Funktionen (siehe später)
- `throw`: Error Handling (siehe später)



<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 Arg & Rückgabe

- Methode: Impl. in Klasse `SwiftTest`

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

- Aufruf aus Klasse `SwiftTest`

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

– Hinweis: Syntax Funktionsaufruf analog zu Java

# Beispielfunktion inkl. Aufruf

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

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

- Aufruf aus Klasse `SwiftTest`

```
var result = sayHelloTo(name: "Swift")  
var testSwift = SwiftTest()  
var value = testSwift.sayHelloTo(name: "Swift")
```

# Funktion mit mehreren Rückgabewerten 😊

- Mit Tupels 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(array: [22, 4, 6, 42, 17, 1, 34])  
print("min is \(bounds.min) and max is \(bounds.max)")

# Param: 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 value: 'a' is a 'let' constant

- Falls ein Parameter doch veränderbar sein soll: mit lokaler `var` überschreiben

- Illustrationsbeispiel:

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

# Funktionen: Argument Labels & Parameter Names

- Argumente von Funktionen können zwei verschiedene Bezeichnungen haben
  - "extern" (beim Aufruf): Argument Label
  - "intern" (im Funktions-Body): Parameter Name
- Beispiel:

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

- Funktionsaufruf: `someFunc(extArgumentLabel: 77)`
- Im Body der Funktion ist der übergebene Wert 77 dann in der Konstanten **localParamName** verfügbar

# Aufruf mit Argument Label

- Aufruf mit Argument-Label ist zwingend, sonst gibt's Kompilierfehler

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

## – Bsp. Anwendung (Funktionsaufruf):

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

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

# Bem. zu Argument & Parameter

- Wenn "Argument Label" und "Parameter Name" gleich sind, reicht einmalige Angabe
  - Siehe z.B. vorhin bei `sayHelloTo(name: String)`, da heissen Argument & Parameter `name`
- Falls Argument Label unerwünscht: `_` (underscore)!

```
func add(_ a: Int, _ b: Int) -> Int {  
    return a + b  
}  
// ...  
let c = add(1, 2)           // ok without arg labels
```



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

# Bem. zu Default-Werten

- 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.



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

# Klassen: Instanziierung & Initialisierung

# Klassen: Dekl. und Instanziierung

```
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`: Intialisierungsmethode, an sich normale Methode
    - ähnlich zu Java-Konstruktor, jedoch ohne spezielle Syntax

# Und noch mehr...

- Neben dem gesehenen bietet / unterstützt Swift noch viel mehr Sprachkonzepte und –konstrukte, z.B.:
  - Closures ("Funktionstyp", siehe später)
  - Zugriffskontrolle: private, internal, public
    - ähnlich wie in Java, "internal = Projekt"
  - Funktionale Programmierung (analog zu Stream@Java8): filter, map & reduce
  - Geschachtelte Funktionen

# ...und viel mehr!

- Subscripts für Klassen, Enums & Structs
  - Ermöglicht Index-Zugriff auf beliebige Objekte
- 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
- ...



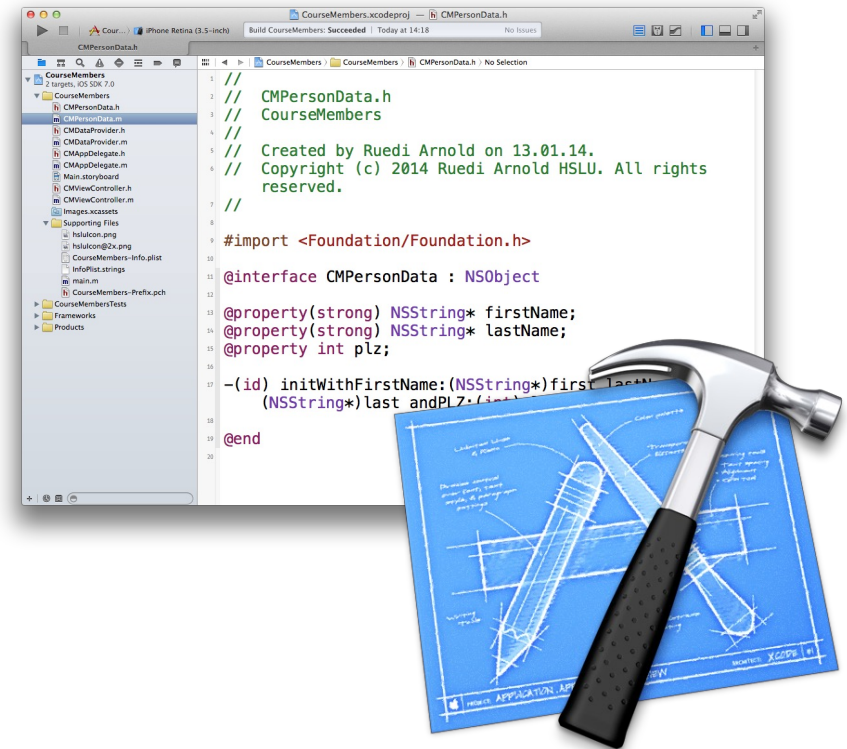


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

# Xcode

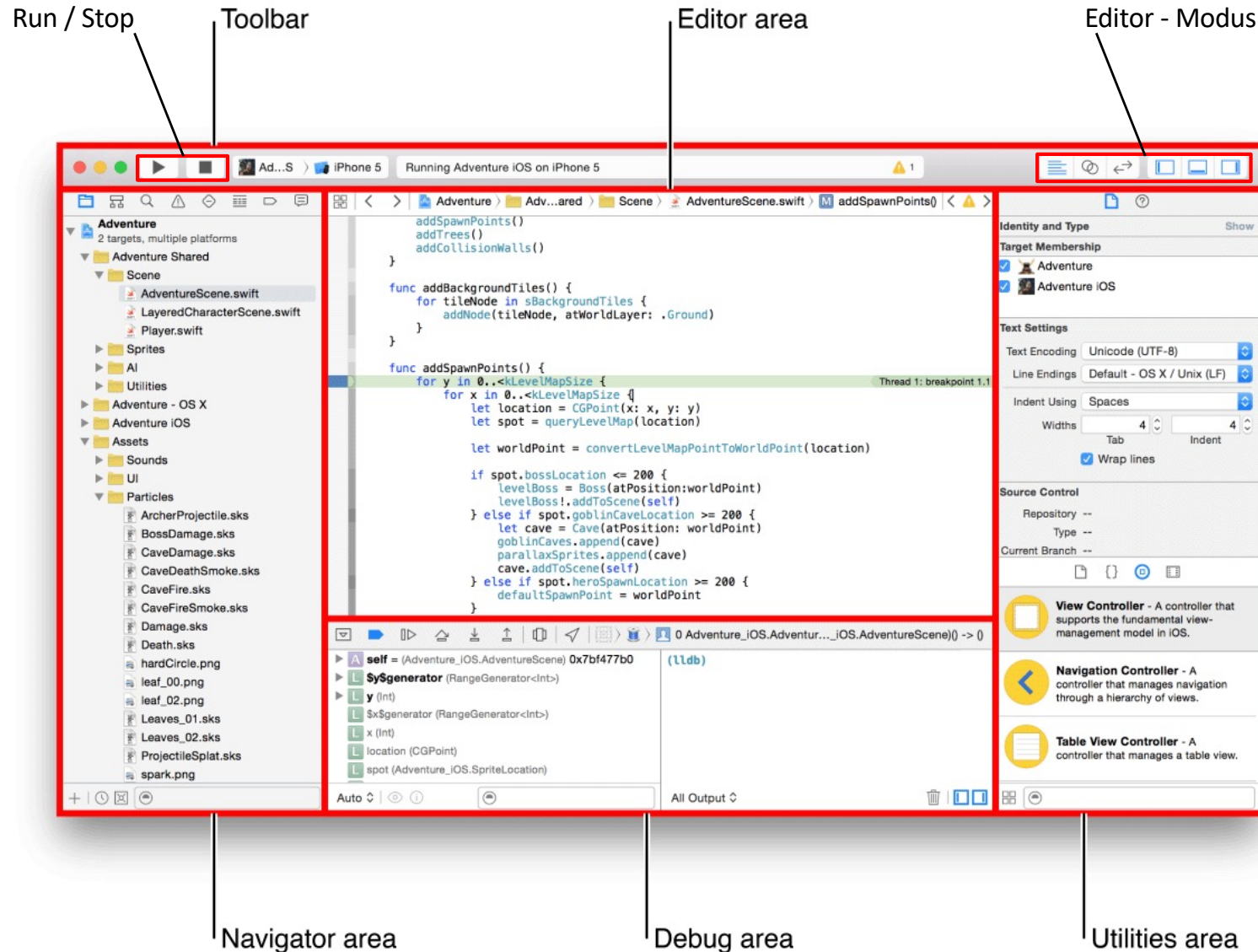
# Xcode IDE

- Code- und Daten-Editor
- Compiler
- iOS-Simulator
- Interface Builder
  - GUI Erstellung
- Debugger
- Instruments
  - Analyse Speicherverbrauch, Funktionsaufrufe, usw.
- Projekt- und Dateiverwaltung
- Versionsverwaltung (Git, SVN)





# Xcode



# Xcode Shortcuts

- esc Autocompletion
- cmd-r Run
- cmd-s Save
- cmd-b Build
- ⌘-Maus-[Code] Springen zur Deklaration
- alt-Maus-[Code] Zugriff Apple Doku

responder <UIApplicationDelegate>

Window \*

Description The UIApplicationDelegate protocol declares methods that are implemented by the delegate of the singleton UIApplication object. These methods provide you with information about key events in an app's execution such as when it finished launching, when it is about to be terminated, when memory is low, and when important changes occur. Implementing these methods gives you

# Apple Resources

- The Swift Programming Language
  - <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>
  - Gute Einführung in die Sprache, sehr nützliche Quelle (vor allem der "Language Guide"!)
- Xcode 14 + Swift 5.7
  - <https://developer.apple.com/download/>
  - oder (bald) über den macOS App Store
- Swift Standard Library Reference
  - [https://developer.apple.com/documentation/swift/swift\\_standard\\_library](https://developer.apple.com/documentation/swift/swift_standard_library)

# Aufgabe 1: iOS-Kursliste

- 1. iPhone-App in Swift! 😊
  - Eigene Klassen & Methoden
  - Selber UIKit-Objekte instanziiieren
  - Properties
  - Eigene init-Methode
  - if, for, switch, ...

