Xcode & Swift: Introduction

Dr.-Ing. Thomas Springer
M.Sc. Martin Weißbach
You need

- Apple Computer
- Xcode 8
- (iOS device)

Hands on course to learn how to program iOS

- 1/1/0 means 45min lecture, 45min seminar
- introducing concepts and theory
- life coding
Modules and Exams

• Modules
  • DSE-14-13, DSE-14-E14
  • INF-B-510, INF-B-520, INF-B-530, INF-B-540
  • INF-BAS4, INF-BI-1, INF-PM-FPA, INF-VERT4
  • Erasmus, … (make sure that you can use the credits for your course of study)

• Credits
  • 2 SWS → 3 credits

• Exam
  • Bachelor: oral – 25min
  • Master & Diplom: module exam
  • Others: check module definition
• regarding lecture, iOS, etc.
  • martin.weissbach1@tu-dresden.de

• exam dates, credibility for your course of study, bureaucracy
  • thomas.springer@tu-dresden.de
Xcode & Swift: Introduction
Xcode Navigator Panel

- Project Navigator
- Symbol Navigator
- Find Navigator
- Issue Navigator
- Test Navigator
- Debug Navigator
- Breakpoint Navigator
- Report Navigator

Utilities Panel

- File Inspector
- Quick Help Inspector

Debug Area

- Left: Variables View
- Right: Console
Xcode: Playground

• Just-In-Time evaluation of code
• chronologic depiction variables, executed command incl. loops etc.

• Useful for:
  • algorithm development (each step performed is easily visible)
  • test of Tests
  • gaining familiarity with new APIs
Xcode: Playground

Left: Playground Code Editor

Right: Life Evaluation of Source Code
Swift: Syntax & Control Structures
• var helloWorld = "Hello World"

• var helloWorld: String

• var red, green, blue: Double

• var red = 0.0, green = 0.0, blue = 1.0

• helloWorld = "Welcome to the new World!"

Variable Declaration
Constant Declaration

- let helloWorld = "Hello World!"
- let red = 0.0, green = 0.0, blue = 1.0

- helloWorld = "Welcome to the new World!"  
  compile-time error
• String
  • representation of sequences of characters
    • `var string = “I am a string”`

• Dictionary<K, V>
  • key-value store
    • `var dictionary = [“lang”: “Swift”, “version”: “2.0”]`

• Array<T>
  • ordered list of objects
    • `var array = [“Swift”, “Objective-C”]`

• Set<T>
  • unordered list of objects
    • `var set = [“Swift”, “Objective-C”]`
if <condition> {  
  <statements>
} else {
  <alternatives>
}

switch <considered_value> {
  case <value_0>:
    <response to value_0>
  case <value_1>, <value_2>:
    <response to value_1 and value_2>
  default:
    <otherwise>
}
for, for-in, while, repeat-while

Swift 3

for <initialization>; <condition>; <increment> {
    <statements>
}

for element in arrayOfElements {
    <statements>
}

for (key, value) in dictionaryOfElements {
    <statements>
}

while <condition> {
    repeat {
        <statements>
    }
} while <condition>
Since Swift 3 the closed range operator (...) is used:

```swift
for index in 1...10 { 
  <statements>
}
```

```swift
(1...10).forEach { 
  print($0) 
  <statements>
}
```
Functions / Methods

• Functions
  • self-contained pieces of code
  • each function has
    • a name, used to identify and “call” the function
    • a type, that consists of the function’s parameter types and return type
  • Methods
    • is function, that is only within the scope, e.g. classes, structures or enumerations
• func fSig(p1: String, p2: Int) -> (ret1: Int, ret2: String)
  • fSig(“A String”, 0) returns (ret1: 0, ret2: “A String”)

• func noParam() -> Int
  • noParam() returns 5

• func noReturnV(param: String) -> ()
  func noReturnV(param: String)
  • noReturnV(“Another String”) returns Void which is ()
• `func ex1(argumentLabel parameterName: Any)`
  • `ex1(argumentLabel: 5)`

• `func pow(basis: Int, exponent: Int) -> Int`
  • `pow(basis: 5, exponent: 3) returns 125`

• `func raiseBase(_ base: Int, by exp: Int) -> Int`
  • `raiseBase(5, by: 3) returns 125`
• used, when a function needs to modify variables outside of the function’s scope
• only variables can be passed as inout parameters
• constants and literals are not allowed, because they cannot be modified

```swift
func swapInts(inout a: Int, inout b: Int){
    let tmpA = a
    a = b
    b = tmpA
}
var a = 5
var b = 10
swapInts(&a,&b)
```
• functions bound to types and instances of types, respectively

• `func exampleMethod() -> String`
  • for instances of both value and reference types

• `class func exampleMethod() -> String`
  • type method for reference types

• `static func exampleMethod() -> String`
  • type method for value types
• similar to other languages’ lambdas

• capture and store references of any constant or variable from the context in which Closures are defined

• global and nested functions are special cases of closures
• global functions
  • have a name and do not capture anything

• nested functions
  • have a name and capture variables from their enclosing function

• closure expressions
  • unnamed and capture values from their surrounding context
Syntax of Closures

General Declaration Syntax:

```swift
func example(source: Int, closure: (parameters) -> returnType) -> Int
```

Definition of function expecting a closure as 2\textsuperscript{nd} parameter:

```swift
func own_filter<T: SequenceType>(source: T, element: (T.Generator.Element) -> Bool) -> [T.Generator.Element]
```

General Expression Syntax:

```swift
{ (parameters) -> returnType in
  // statements here
}
```

Example Implementation:

```swift
{ (array: T, (element: Int) -> Bool in
  return element == 42
}
```
Classes, Structures, Enumerations

Syntax & Semantics
• Reference Type

• Single Inheritance

• Memory Management rules apply

class ExampleClass: ExamplesSuperClass {
    var property_1: String
    let property_2 = 10.4

    func method_1() -> Bool {
        <statements>
    }
}
• Value Type

• No inheritance or Memory Management rules apply

• Best suited for data encapsulation

```swift
struct ExampleStructure {
    var width: Int
    var height: Int

    func plane() -> Int {
        return width * height
    }
}
```
Enumerations

- Value Type
- group related values
- No inheritance or Memory Management rules apply

```swift
enum Directions {
    case North
    case South
    case East
    case West

    func oppositeDirection() -> Directions {
        <statements>
    }
}
```
Behavior with `let` and `var`
Reference and Value Types behave differently
Reference Types with `let`

```swift
class Car {
    let ps: Int
    var numberPlate: String
}

let car = Car(ps: 310, numberPlate: "DD-XX-1234")
car = Car(ps: 120, numberPlate: "M-XX-1234")
car.ps = 410
car.numberPlate = 30
car.numberPlate = "DD-XY-9876"
```
Reference Types with `var`

class Car {
    let ps: Int
    var numberPlate: String
}

var car = Car(ps: 310, numberPlate: "DD-XX-1234")
car = Car(ps: 120, numberPlate: "M-XX-1234")
car.ps = 410

var car = Car(ps: 310, numberPlate: "DD-XX-1234")
car = Car(ps: 120, numberPlate: "M-XX-1234")
car.ps = 410

car.numberPlate = 30

car.numberPlate = "DD-XY-9876"
Value Types with let

```swift
struct Painting {
    let artist: String
    var price: Float
}

let painting = Painting(artist: "Aito", price: 5.99)
painting = Painting(artist: "Ashisu", price: 3.99)
painting.artist = "Fuwa"
painting.price = 5
painting.price = 7.99
```

✔
✗
✗
✗
✔
✗
✗
✗
Value Types with `var`

```swift
struct Painting {
    let artist: String
    var price: Float
}

var painting = Painting(artist: "Aito", price: 5.99)
painting = Painting(artist: "Ashisu", price: 3.99)
painting.artist = "Fuwa"
painting.price = 5
painting.price = 7.99

✔
✗
✔
✔
✔
```
## Summary

<table>
<thead>
<tr>
<th></th>
<th>let</th>
<th>var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Type</td>
<td>Value Type</td>
<td>Reference Type</td>
</tr>
<tr>
<td>Reassignments</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Modify</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutable Properties</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Immutable Properties</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Protocols
• declaration of methods and properties without providing an implementation
• can be adopted by a class, structure or enumeration
• types adopting a protocol are responsible for providing an implementation for the declared requirements of the protocol

```swift
protocol ExampleProtocol {
    // Protocol definition
}

class ClassType: ExampleProtocol {
    // Class definition
}
```