03.05.2019
Lisa Obermaier, Simon Thum
Mobile Application Development
Prof. Dr. Gudrun Socher

Introduction to Swift

1. Motivations for Swift

> "Software development never stands still. But if there's one gravitational force that holds sway over the entire profession, it's that abstraction increases over time. Progress is not linear—there have been periods of stagnation, great leaps forward, and plenty of setbacks —but the long-term trend is undeniable."[1]

Since Mac OS X, Apple has relied on Objective-C as its general-purpose programming language, against which APIs and Frameworks are written; this is rooted in OS X's history as quasi-successor to NextStep.

Apple explored the possibility of creating a new programming language internally. Motivation can only be speculated on, but an unwillingness to lose technological high ground to more modern languages – especially those not requiring manual memory management –, and to stay in control of the whole development toolchain on its platforms, can be considered likely candidates.

Development was spearheaded by Chris Lattner of LLVM fame, and the result of this project was Swift 1.0, presented to the public for the first time at WWDC 2014.

Swift is described by Apple as safe, fast, and, as an end goal, as "the best available language for uses ranging from systems programming, to mobile and desktop apps, scaling up to cloud services."[2]

As of 2019, the current version of Swift 5.0 has been released.

---

[1] https://arstechnica.com/gadgets/2014/10/os-x-10-10/21/

[2] https://swift.org/about/

2. Technical background

2.1. Swift at a glance and beyond its basics

Swift is an imperative and functional programming language, follows a block-structure and is object- and protocol-oriented. It contains static typing as well as type inference and provides dot-notations and UTF-8 encoding. Most of these ideas are borrowed from other programming languages, such as Objective-C, Rust, Haskell, Ruby, Python, C#, CLU and many others.[3]

With a closer look, there come even more useful constructs with this programming language. Such as Generics, Extensions, Closures and Optionals. Swift uses ARC for Memory Management – Automatic Reference Counting. It also has proper Error Handling and Assertions, which are useful for Debugging. In the following, we will look at a simple example for every one of these constructs.

2.2 Error Handling

Swift uses the keywords try, throw(s) and catch for error handling. A function, that could throw an error needs throws in the signature and every following call of this function needs a try-catch. If the try does not work, the error is properly handled as implemented.

```
do {
    try makeASandwich()
    eatASandwich()
} catch SandwichError.outOfCleanDishes {
    washDishes()
} catch SandwichError.missingIngredients(let ingredients) {
    buyGroceries(ingredients)
}
```

In our example we are trying to call the function makeASandwich(), which fails, if there are not enough clean dishes and we call the error washDishes(), or if we are missing some ingredients, which calls the error buyGroceries() with the certain missing ingredient.

[3] https://en.wikipedia.org/wiki/Swift_(programming_language)

2.3 Debugging: Assertions

An assertion can be called from the standard library with assert(_:_:file:line:). This function gets an expression with a true/false evaluation and a message in case of false.

```
let age = -3
assert(age >= 0, "A person's age can't be less than zero.")
// This assertion fails because -3 is not >= 0.
```

In our example, if age >= 0 the assertion is true and the code execution would continue. But our age is negative and the assertion fails. This leads to the termination of our application.

2.4 Generics

Whenever the code doesn't need to be very specific and precise, it can be useful to write Generics instead of specified types.[4]

besonders bei Beispielen mit mehr als 2 Bildern brauchte ich einen Rahmen um den Text einfacher zu machen

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

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

In our example we want to swap strings in the first function and doubles in the second function. It is clearly visible that we are rewriting the same code just to change the type String into Doubles. Instead we can use the generic <T> to suggest every variable of type T has to be of the same type, but in general can be any possible type. In the second picture you can see how the two functions can be combined with an optional type. [5]

[4] 2016 Book Practical Swift, page 101f.

[5] https://docs.swift.org/swift-book/LanguageGuide/Generics.html

2.5 Extensions

With Extensions it is possible to add new functionality to an existing class, structure, enumeration, or protocol type, while you don't need to have access to the original source code. This is called retroactive modelling. But note, that extensions can not override existing functionality.[6]

```
protocol StringConvertible {
    func toString() -> String
}


extension String: StringConvertible {
    func toString() -> String {
        return self
    }
}



var thisMustHaveAToString: StringConvertible

/* ... */

print(thisMustHaveAToString.toString())
```

In our example we have an extension of the type *StringConvertible* and the variable *thisMustHaAToString* is of the same type. The extension allows the variable to use the toString() without implementing it itself.

2.6 Closures

Closures are self-contained blocks of functionality and those blocks can be subroutines, functions, procedures or methods. So a closure is a block, whose code refers to variables outside of the closure block. Most commonly closures are used, when you don't know when you want to perform a certain block.[7]

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

[6] https://docs.swift.org/swift-book/LanguageGuide/Extensions.html

[7] 2018 Book Learn Computer Science With Swift, page 216

```swift
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

```swift
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

```swift
reversedNames = names.sorted(by: >)
```

In our example, we use the method *sorted(by:)* from the Swift standard library, that works based on a sorting closure, which we provide in our function *backward*. The second picture shows an inline closure, where the parameters and return type are written inside the curly braces. An even shorter version in displayed in the third example, where we don't even need the String types, because we are calling the method on an array of Strings and no other type is possible. The last bit of code shows the shortest possible version, because Swift's String-type defines a string-specific implementation exactly as the needed type in sorted(by:). So all you need is the greater than.[8]

2.7 Optionals

Optionals are used, when a value could not be assigned. An optional integer would be displayed as „Int?" while the question mark indicated the Optional type.[9] This construct gives the possibility to avoid unintentional calculations or programming around possible missing values. Instead a missing value of our „Int?" type would be nil.[10]

```swift
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?", or "optional Int"

if convertedNumber != nil {
    print("convertedNumber has an integer value of \(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

In our example we convert our number to a possible int as you can see in the second line. The variable has the type of „Int?" and is printed correctly, because we were able to

[8] https://docs.swift.org/swift-book/LanguageGuide/Closures.html

[9] 2018 Book Swift 4 for absolute Beginners, page 33

[10] 2018 Learn Computer Science, page 201ff.

convert the string *possibleNumber* into the int *convertedNumber*. Getting the value of an optional is done via unwrapping with the ! operator.

2.8 Property Observers

Object properties can be observed with *willSet* and *didSet* methods, which will be called right before and after a change of this instantiated property occurs at runtime.

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
```

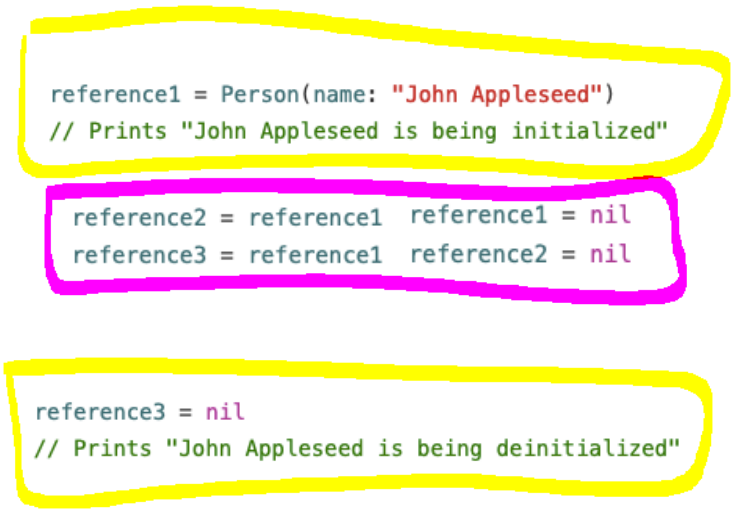
In our example ...

2.9 Memory Management: ARC

In Most cases you don't need to think about memory management, because ARC frees memory as soon as instances are no longer needed. ARC tracks how many properties, constants or variables are referring to an instance of a class and not deallocate the instance as long as those references still exist.

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}

var reference1: Person?
var reference2: Person?
var reference3: Person?
```



In our example, we have a class called *Person* and we can set the *name* and print a message. It also has a deinitializer with its own message. Next we define the variables of the optional type Person? that are automatically initialized with nil. If we create a new *Person* instance and assign it to reference1, we get the initialization printline. This reference1 can also be assigned to other references, as displayed next. If we assign nil to two of the references, the reference3 is still attached to the assigned Person, although it's not the originally created reference. Only if we set reference3 also to a nil value, ARC deallocates the *Person* instance.[11]

3. iOS App Development with Swift

3.1 Playgrounds

While Swift features a REPL (Read Evaluate Print Loop) mode thanks to its connection to the LLVM, it is of limited use for developing mobile applications. However, a conceptually similar feature named *Playgrounds* might merit a mention.

In its most basic form, Playgrounds are Swift source code files running in a permanent debug mode, allowing stepping and viewing of all variables at all times. It is however possible to programmatically create any Cocoa Touch widget without having to run code on a simulator. This can be a helpful feature for prototyping certain Views without the overhead of the complete project.

Playgrounds can be published with rich text annotations and packaged assets to create training material.

[11] https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

3.2 XCode

iOS app development is generally done in Apple's IDE, *XCode*.

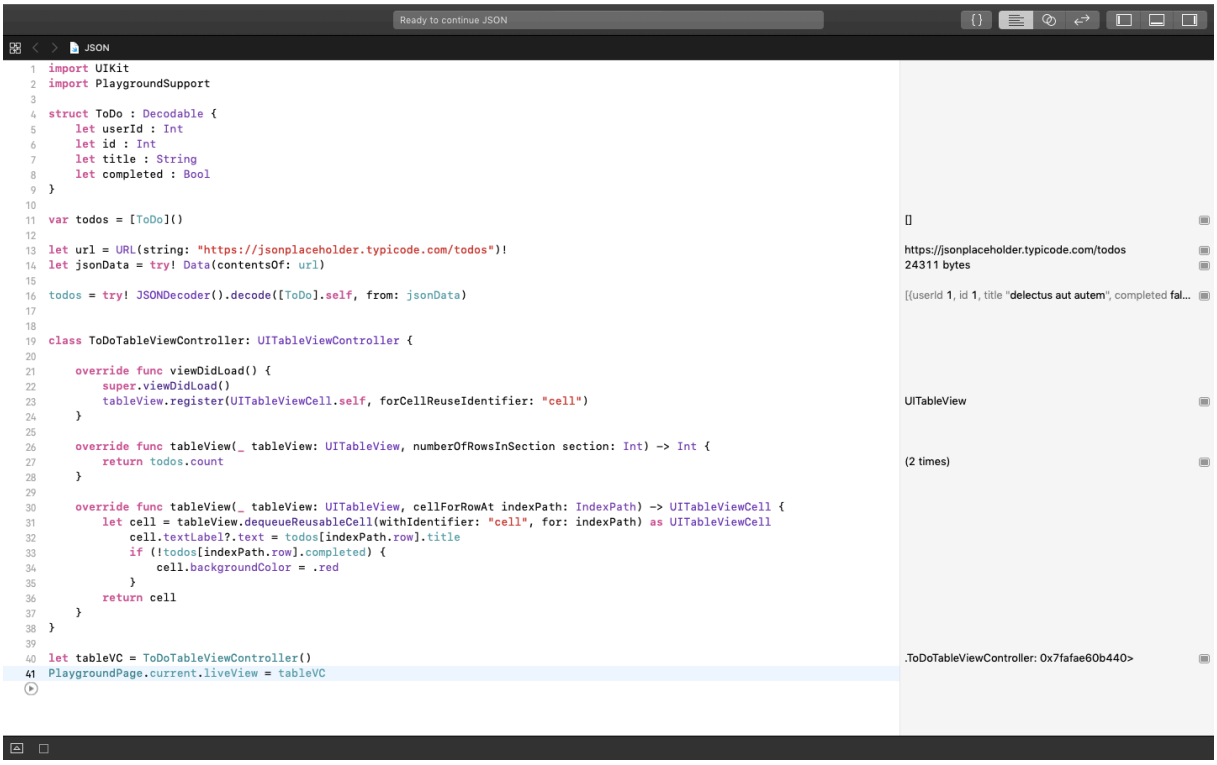
User Interfaces can be programmatically built or created by dragging and dropping widgets and modifying their parameters. ViewController swift files get hooks into the GUI by dragging widgets into the code and linking them as either callers of methods, or as values to be read.

Application logic code is generally separated from the view controller into own files, as per MVC paradigm.

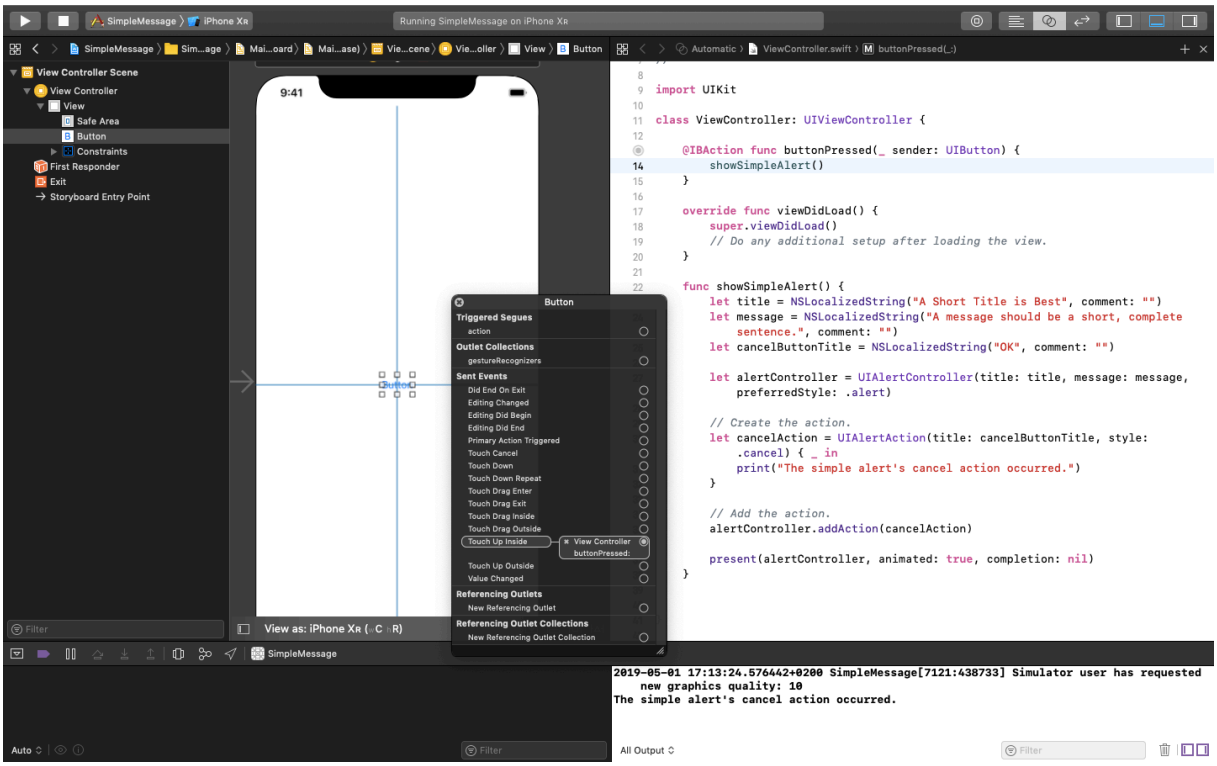
Applications can then be compiled and run on proper iOS hardware, or just in an included simulator.

4. Examples

Playground: Extracting data from a JSON file and filling a UITableView with it.



Application: Showing a system message box.



5. Summary

How far Swift has come in its original mission statement of being the best available language for virtually all use cases remains to be seen.

der Satzteil stimmt nicht ganz ...

However, when developing mobile applications for iOS without consideration for other platforms, it's safety and versatility makes Swift are far more worthwhile option than Objective-C. Its interface compatibility with Objective-C makes it easy to integrate new Swift classes into existing Objective-C projects.[12] As of Swift 5.0, ABI (application binary interface) stability has achieved, allowing for reuse of written libraries without recompilation. A side effect of ABI stability is a reduced likelihood of code-breaking Swift syntax changes, making now an opportune time to start development in Swift.

[12] https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis

6. Further Reading

https://arstechnica.com/gadgets/2014/10/os-x-10-10/22/

OS X 10.10 Yosemite: The Ars Technica Review, offering an in-depth view about the technical aspects of Swift, in particular the intermediate SIL and IR forms created by the Swift compiler and LLVM.