Hochschule für angewandte Wissenschaften München
Fakultät für Informatik und Mathematik
SoSe 2019
Mobile Application Development
Prof. Dr. Gudrun Socher

An Introduction to Swift

in Regards to Mobile Application Development

Lisa Obermaier & Simon Thum

Table of Contents

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[2], and the result of this project was Swift 1.0, presented to the public for the first time at WWDC 2014[3].

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."[4]

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

---

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

[2] http://nondot.org/sabre/Resume.html

[3] https://developer.apple.com/videos/play/wwdc2014/101/

[4] 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.[5]

With a closer look, even more useful constructs with this programming language become apparent, such as Generics, Extensions, Closures, Optionals or Property Observers. 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.[6]

```
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 resolution method *buyGroceries()* with the certain missing ingredient.

---

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

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

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.[7]

```
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.[8]

```
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
}
```

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 *double*. 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. [9]

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

[7] https://developer.apple.com/documentation/swift/1541112-assert

[8] 2016 Book Practical Swift, page 101f.

[9] 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.[10]

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


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



var thisMustHaveAToString: StringConvertible

/* ... */

print(thisMustHaveAToString.toString())
```

In our example we declare a *string* extension called *StringConvertible* and the variable *thisMustHaveAToString* is of the same type. The protocol of the extension guarantees that any variable of that type will have a *toString()* method.

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

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 the closure block. Most commonly closures are used, when you don't know when you want to perform a certain block.[11]

```
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"]
```

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.

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

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.

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

The last bit of code shows the shortest possible version, because Swift's string-type defines its string-specific implementation exactly as the type needed in *sorted(by:)*. So all you need is the greater than.[12]

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

---

[11] 2018 Book Learn Computer Science With Swift, page 216

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

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.[13] 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.[14]

```
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 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
```

13 2018 Book Swift 4 for absolute Beginners, page 33

14 2018 Book Learn Computer Science With Swift, page 201ff.

In our example we see a variable called totalSteps. The willSet and didSet methods are hooked with this variable, which you can see if you look closer at the curly-braced, right after the variable declaration. So these willSet and didSet work exclusively for this variable and willSet is called right before the variable is set and didSet is called right after the variable is set.[15]

In this case, print methods will be called every time before and after the field *totalSteps* of the *StepCounter* class is changed, no matter what induced the change. This can be used for sanity checks and data history, for 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")
    }
}
```

In our example, we have a class called *Person* and we can set the *name* and print a message. It also has a de-initialiser with its own message.

Next we define the variables of the optional type Person? that are set to *nil* before initialisation.

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

[15] https://docs.swift.org/swift-book/LanguageGuide/Properties.html

If we create a new *Person* instance and assign it to *reference1*, we get the expected initialisation *printline*.

```
reference1 = Person(name: "John Appleseed")
// Prints "John Appleseed is being initialized"
```

This *reference1* can also be assigned to other references, as displayed next. Now three variables retain a reference to the *Person* instance.

```
reference2 = reference1
reference3 = reference1
```

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.

```
reference1 = nil
reference2 = nil
```

Only if we set *reference3* to a *nil* value as well, ARC deallocates the *Person* instance automatically, because all three retained references have been released.[16]

```
reference3 = nil
// Prints "John Appleseed is being deinitialized"
```

---

16 https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

3. iOS App Development With Swift: An Overview

3.1 Playgrounds

While Swift features a *REPL* (Read Evaluate Print Loop) mode thanks to its connection to the LLVM[17], 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.[18]

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 and set.

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

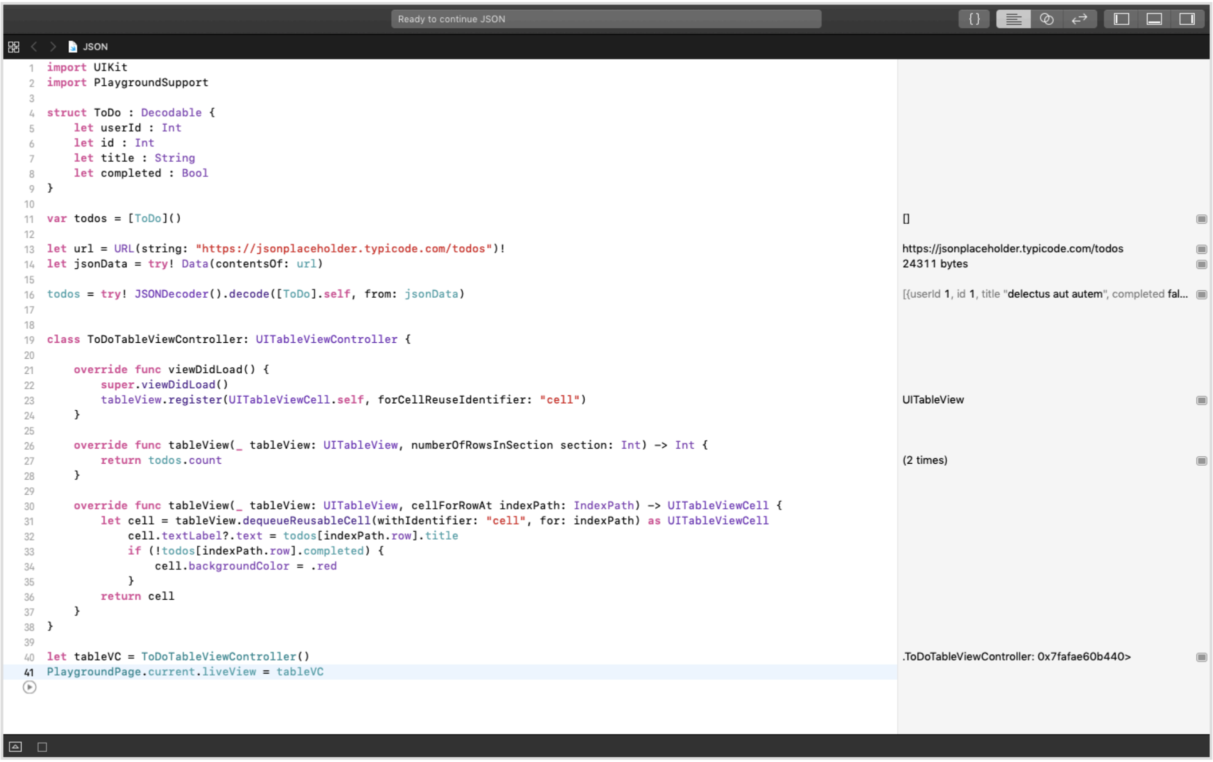
Applications can then be compiled and run on proper iOS hardware, or in an included simulator.

[17] https://swift.org/lldb/

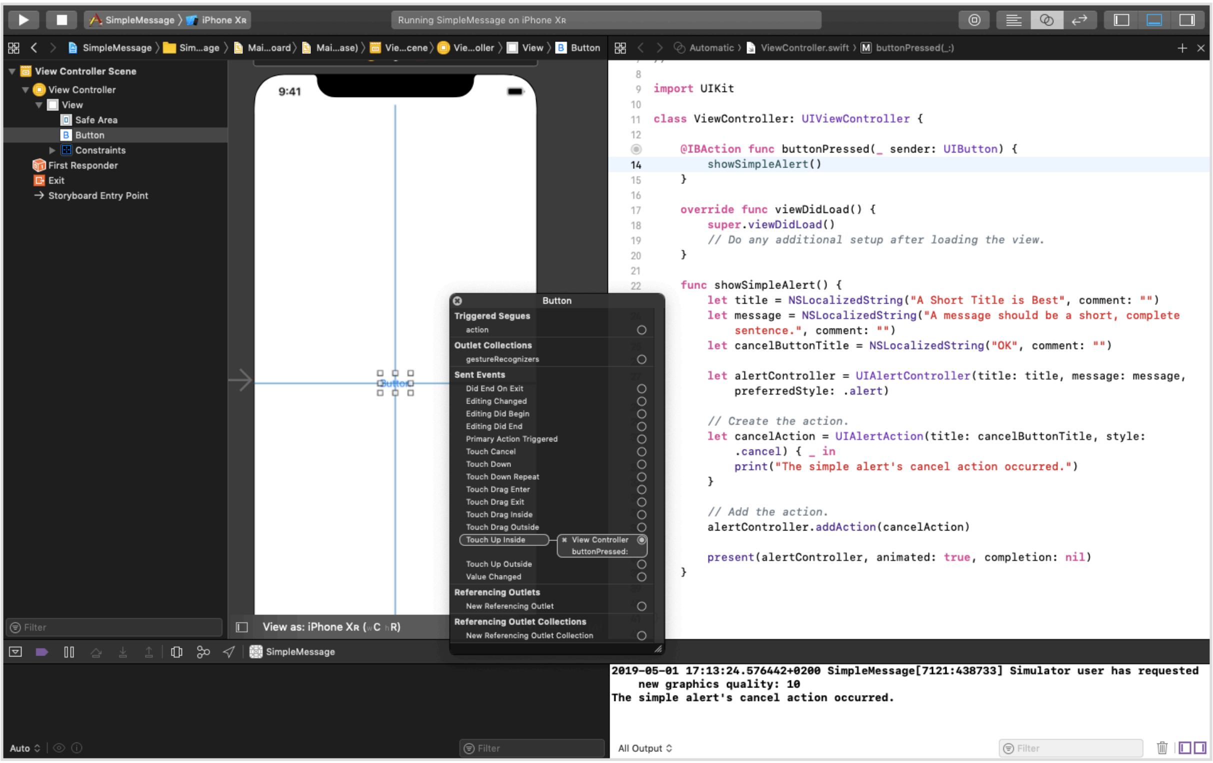
[18] https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_markup_formatting_ref/index.html

4. Examples

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



Mobile 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.

However, when developing mobile applications for iOS without consideration for other platforms, its safety and versatility make Swift a 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.[19] 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, should you be so inclined.

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

6. References

- Downey, Eric: **Practical Swift**. 1st edition. Berkeley: Apress, 2016.
- Feiler, Jesse: **Exploring Swift Playgrounds** - The Fastest and Most Effective Way to Learn to Code and to Teach Others to Use Your Code. 1st edition. Berkeley: Apress, 2017.
- Feiler, Jesse: **Learn Computer Science With Swift**. 1st edition. Berkeley: Apress, 2018.
- Kaczmarek, Stefan; Lees, Brad; Bennett, Gary: **Swift 4 for Absolute Beginners**. 4th edition. Berkeley: Apress, 2018.

- Apple, Inc.: **About Swift**. https://swift.org/about/ *most recently accessed 30 May 2019.*
- Apple, Inc.: **assert(_:_:file:line:)**. https://developer.apple.com/documentation/swift/1541112-assert *most recently accessed 30 May 2019.*
- Apple, Inc.: **Automated Reference Counting**. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html *most recently accessed 30 May 2019.*
- Apple, Inc.: **Closures**. https://docs.swift.org/swift-book/LanguageGuide/Closures.html *most recently accessed 30 May 2019.*
- Apple, Inc.: **Error Handling**. https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html *most recently accessed 30 May 2019.*
- Apple, Inc.: **Extensions**. https://docs.swift.org/swift-book/LanguageGuide/Extensions.html *most recently accessed 30 May 2019.*
- Apple, Inc.: **Generics**. https://docs.swift.org/swift-book/LanguageGuide/Generics.html *most recently accessed 30 May 2019.*
- Apple, Inc.: **Imported C and Objective-C APIs**. https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis *most recently accessed 30 May 2019.*
- Apple, Inc.: **Markup Overview**. https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_markup_formatting_ref/index.html *most recently accessed 30 May 2019.*
- Apple, Inc.: **Properties**. https://docs.swift.org/swift-book/LanguageGuide/Properties.html *most recently accessed 30 May 2019.*
- Apple, Inc.: **REPL and Debugger**. https://swift.org/lldb/ *most recently accessed 30 May 2019.*
- Apple, Inc.: **WWDC 2014 Keynote**. https://developer.apple.com/videos/play/wwdc2014/101/ *most recently accessed 30 May 2019.*
- Lattner, Chris: **Chris Lattner's Resume**. http://nondot.org/sabre/Resume.html *most recently accessed 30 May 2019.*
- Siracusa, John: **OS X 10.10 Yosemite: The Ars Technica Review**. https://arstechnica.com/gadgets/2014/10/os-x-10-10/ *most recently accessed 30 May 2019.*

7. Further Reading

https://swift.org

Swift.org, the official project website, featuring all relevant information.

https://github.com/apple/swift-evolution

The Swift Programming Language Evolution repository, showcasing a roadmap for future feature development. Open to community proposals via pull requests.

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.