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

Introduction to Swift

1. Motivations for Swift

Swift is a rather new programming language designed by Chris Lattner and developed by Apple. It had its first official release on September 9, 2014 with Swift 1.0. It combines features from Objective-C and features used by modern programming languages. It also bridges the gap between compiled languages, which are faster in execution, and interpreted languages, which are easier to learn. [1]

But why was Swift invendet, while Objective-C was the programming language for iOS?

[1] 2018 Swift 4 for absolute beginners, page 83

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.[2]

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. Generics

Whenever the code doesn't need to be very specific and precise, it can be useful to write Generics instead of specified types.[3]

```
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 cleary 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

[2] https://en.wikipedia.org/wiki/Swift_(programming_language)
[3] 2016 Book Practical Swift, page 101f.

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. [4]

2.3. 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 modeling. But note, that extensions can not override existing functionality.[5]

```
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.

2.4. 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.[6]

---

[4] https://docs.swift.org/swift-book/LanguageGuide/Generics.html
[5] https://docs.swift.org/swift-book/LanguageGuide/Extensions.html
[6] 2018 Book Learn Computer Science With Swift, page 216

```swift
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"]
```

```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 ist string-specific implementation exactly as the needed type in sorted(by:). So all you need is the greater than.[7]

2.5. 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 opional type.[8] 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.[9]

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

[7] https://docs.swift.org/swift-book/LanguageGuide/Closures.html
[8] 2018 Book Swift 4 for absolute Beginners, page 33
[9] 2018 Learn Computer Science, page 201ff.

```
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*.

2.6. Memory Management: ARC

In Most cases you don’t need tot hink about memory management, because ARC frees u 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?
```

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

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

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

```
reference3 = nil
// Prints "John Appleseed 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 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 tot wo 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.[10]

2.7. Error Handling

Swift uses the keywods *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. [11]

```
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.

2.8. 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.[12]

[10] https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html
[11] 2016 Practical Swift, page 7f.
[12] https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html

3. iOS App Development with Swift

4. Examples

5. Summary