Swift

Language Guide

Kristmann Lukas
Rumpfhuber Clemens

Inhaltsverzeichnis

1. Funktionen (1)

1.1. Grundlagen
1.2. Funktionen ohne Parameter
1.3. Funktionen mit Parametern
1.4. Funktionen ohne Rückgabewert
1.5. Rückgabewert ignorieren
1.6. Funktionen mit mehren Rückgabewerten
1.7. Optionale Tupeltyp-Rückgabe
1.8. Implizierte Rückgabe
1.9. Argument- und Parameternamen
1.10. Argumentlabel

1. Funktionen (2)

1.11. Argumentbezeichnungen weglassen
1.12. Standardparameterwerte
1.13. Variierende Parameter
1.14. In-Out Parameter
1.15. Funktionstypen
1.16. Funktionstypen als Parametertypen
1.17. Funktionstypen als Rückgabetypen
1.18. Verschachtelte Funktionen

1.1. Grundlagen

- Abgeschlossener Codeblock
- Handelt eine bestimmte Aufgabe ab
- Besitzt Namen zur Identifizierung
- Optional: Parameter und/oder Rückgabewert
- Können geschachtelt werden

1.2. Funktionen ohne Parameter

- Funktion ohne Parameter mit Rückgabewert (String)
- Ausgabe der Methode auf Konsole

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Prints "hello, world"
```

1.3. Funktionen mit Parametern

- Funktion mit mehreren Parametern (String, Boolean) und Rückgabewert (String)
- Ausgabe der Methode auf Konsole

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

1.4. Funktionen ohne Rückgabewert

- Funktion mit Parameter (String) ohne Rückgabewert

```
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
// Prints "Hello, Dave!"
```

- Streng genommen:
  Dennoch Rückgabewert, obwohl keiner definiert vom Typ *Void* bzw. *()*

1.5. Rückgabewert ignorieren

- Der Rückgabewert kann ignoriert werden

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting(string: "hello, world")
// prints "hello, world" but doesn't return a value
```

1.6. Funktionen mit mehren Rückgabewerten

- Es können mehrere Werte mit dem Tupeltyp als Rückgabetyp zurückzugeben werden

```
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)
}
```

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```

1.7. Optionale Tupeltyp-Rückgabe

- Der Rückgabewert hat das potenzial, „keinen Wert“ zu haben.

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    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)
}
```

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
// Prints "min is -6 and max is 109"
```

1.8. Implizierte Rückgabe

- Bei einem einzelnen Ausdruck, kann auf das *return* verzichtet werden.

```swift
func greeting(for person: String) -> String {
    "Hello, " + person + "!"
}
print(greeting(for: "Dave"))
// Prints "Hello, Dave!"

func anotherGreeting(for person: String) -> String {
    return "Hello, " + person + "!"
}
print(anotherGreeting(for: "Dave"))
// Prints "Hello, Dave!"
```

1.9. Argument- und Parameternamen

- Jeder Funktionsparameter hat eine Argumentbezeichnung und einen Parameternamen.
- Die Argumentbezeichnung wird beim Aufrufen der Funktion verwendet

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

1.10. Argumentlabel

- Optional kann ein Label für die Argumente abgeschrieben werden.

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

1.11. Argumentbezeichnungen weglassen

- Wenn keine Argumentbezeichnung gewünscht ist, kann man einen Unterstrich (_) anstelle einer expliziten Argumentbezeichnung für diesen Parameter anschreiben.
- Achtung! Wenn ein Parameter ein Argumentlabel hat, muss das Argument beim Aufrufen der Funktion gekennzeichnet werden.

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(1, secondParameterName: 2)
```

1.12. Standardparameterwerte

- Es können Standardwerte für einzelne Parameter definiert werden.

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
    // If you omit the second argument when calling this function, then
    // the value of parameterWithDefault is 12 inside the function body.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) //
  parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault is 12
```

1.13. Variierende Parameter

- Ein variierender Parameter akzeptiert null oder mehr Werte eines angegebenen Typs.
- Die an einen Parameter übergebenen Werte werden als Array des entsprechenden Typs zur Verfügung gestellt.

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

1.14. In-Out Parameter

- Funktionsparameter sind standardmäßig Konstanten.
- Der Versuch, den Wert eines Funktionsparameters zu ändern, führt zu einem Kompilierzeitfehler.
- Wenn eine Funktion den Parameterwert ändert soll, definiert man diesen Parameter als einen In-Out-Parameter.
- Achtung! In-Out-Parameter dürfen keine Standardwerte haben und variierende Parameter können auch nicht als *inout* markiert werden.

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

1.15. Funktionstypen

- Es können „Shortcuts“ von Methoden erstellt werden.

```swift
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

```swift
var mathFunction: (Int, Int) -> Int = addTwoInts
```

```swift
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```

1.16. Funktionstypen als Parametertypen

- Funktionstypen als Parametertyp können für eine andere Funktion verwenden.
- So können einige Aspekte der Implementierung einer Funktion dem Aufrufer der Funktion überlassen, wenn die Funktion aufgerufen wird.

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8"
```

1.17. Funktionstypen als Rückgabetypen (1)

- Es können Funktionstypen als Rückgabetypen einer anderen Funktion verwenden werden. Dies ist direkt nach dem Return-Pfeil (->) der zurückkehrenden Funktion einen vollständigen Funktionstyp zuschreiben.

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

1.17. Funktionstypen als Rückgabetypen (2)

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

1.18. Verschachtelte Funktionen

- Es können Funktionen in Funktionen geschrieben werden.

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

2. Verschlüsse

2.1. Grundlagen
2.2. sorted-Methode
2.3. Syntax von Closures
2.4. Ableiten des Typs aus dem Kontext
2.5. Kurze Argumentnamen
2.6. Operatormethode
2.7. Trailing Closures
2.8. Werteerfassung
2.9. Autoclosure

2.1. Grundlagen

- Vergleichbar mit Lambdas aus anderen Programmiersprachen
- Eine sauberer, klarer Stil mit Optimierungen, die eine kurze, übersichtliche Syntax in gängigen Szenarien fördern:
  - Ableiten von Parameter- und Rückgabewerttypen aus dem Kontext
  - Implizite Rückgaben von Einzelausdrucksschließungen
  - Kurze Argumentnamen
  - Nachgestellte Closure-Syntax

2.2. sorted-Methode

- Die Standardbibliothek bietet eine Methode, die ein Array, basierend auf der Rückgabe einer bereitgestellten Funktion, sortiert.

```
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"]
```

2.3. Syntax von Closures

- Die Syntax einer Closures hat die folgende allgemeine Form:

```
{ (parameters) -> return type in
    statements
}
```

- Beispiel anhand des letzten Beispiels:

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

2.4. Ableiten des Typs aus dem Kontext

- Da der Sortierabschluss als Argument an eine Methode übergeben wird, kann Swift die Typen seiner Parameter und den Typ des zurückgegebenen Werts ableiten.

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

2.5. Kurze Argumentnamen

- Swift stellt Inline-Closures automatisch Argumentnamen in Kurzform bereit, die verwendet werden können, um auf die Werte der Closure-Argumente mit den Namen *$0*, *$1*, *$2* usw. zu verweisen.

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

2.6. Operatormethode

- Es geht noch kürzer. Die Variablen können bei der Operatormethode weggelassen werden.

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

2.7. Trailing Closures (1)

- Der Abschlussausdruck kann auch am Ende der Funktion geschrieben werden.

```swift
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

```swift
reversedNames = names.sorted() { $0 > $1 }
```

```swift
reversedNames = names.sorted { $0 > $1 }
```

2.7. Trailing Closures (2)

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings is inferred to be of type [String]
// its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

2.8. Werteerfassung (1)

- Closures können Konstanten aus dem umgegeben Kontext erfassen
- Sowie auf Konstanten und Variablen verweisen und diese verändern
- Umsetzung: verschachtelte Funktionen

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

2.8. Werteerfassung (2)

```
let incrementByTen = makeIncrementer(forIncrement: 10)

incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30

let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
// returns a value of 7

incrementByTen()
// returns a value of 40
```

2.9. Autoclosure

- Autoclosure verzögern die Ausführen
- Diese werden erst ausgeführt, wenn sie aufgerufen werden

```swift
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// Prints "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// Prints "5"

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// Prints "4"
```

3. Indizes

3.1. Grundlagen
3.2. Indexsyntax
3.3. Indexoptionen

3.1. Grundlagen

- Klassen, Strukturen und Aufzählungen können Indizes definieren
- Verwendung: Zugriff auf Werte nach Index, ohne separate Methoden

3.2. Indexsyntax (1)

- Syntax

```
subscript(index: Int) -> Int {
    get {
        // Return an appropriate subscript value here.
    }
    set(newValue) {
        // Perform a suitable setting action here.
    }
}
```

- Schreibgeschützt:

```
subscript(index: Int) -> Int {
    // Return an appropriate subscript value here.
}
```

3.2. Indexsyntax (2)

- Beispiel: Instanz von *TimesTable* stellt 3er-Tabelle dar

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")
// Prints "six times three is 18"
```

3.3. Indexoptionen (1)

- Indizes können eine beliebige Anzahl von Eingabeparametern annehmen
- Können auch einen Wert eines beliebigen Typs zurückgeben
- Klasse oder Struktur kann unendlich viele Indeximplementierungen bereitstellen

3.3. Indexoptionen (2)

- Beispiel: Matrix-Struktur

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

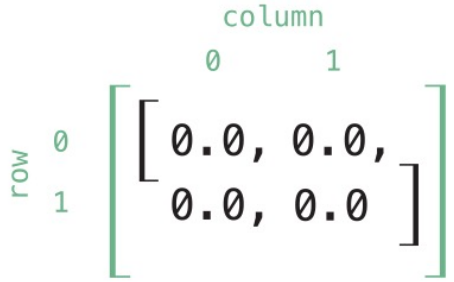
```
grid = [0.0, 0.0, 0.0, 0.0]

          column
            0     1
row  0  [ [0.0, 0.0,
     1    0.0, 0.0 ] ]
```

3.3. Indexoptionen (3)

- Beispiel: Matrix-Struktur

```
var matrix = Matrix(rows: 2, columns: 2)
```




```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

Optional Chaining

- 4.1. Grundlagen
- 4.2. Forced Unwrapping
- 4.3. Optional Chaining

4.1. Grundlagen

- Verfahren zum Abfragen und Aufrufen von einem optionalen Element
- Überprüft ob ein Element nil ist
- Abfragen können verkettet werden
- ! = Forced Unwrapping
  … wenn nil -> Runtime Error
- ? = Optional Chaining
  … wenn nil -> gibt nil zurück
- Typ des Ergebnisses ist immer der erwartete Typ

4.2. Forced Unwrapping

- Klassen erstellen

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

- Person-Instanz erstellen

```
let john = Person()
```

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

4.3. Optional Chaining (1)

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

4.3. Optional Chaining (2)

```
john.residence = Residence()
```

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "John's residence has 1 room(s)."
```

4.3. Optional Chaining (3)

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

Type Casting

- 5.1. Grundlagen
- 5.2. Checking Type
- 5.3. Downcasting
- 5.4. Type Casting for Any

5.1. Grundlagen

- Überprüft Typ oder behandelt eine Instanz anders
- Operatoren:
  - is
  - as
  - as!
  - as?

5.2. Checking Type (1)

- Klassen erstellen

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

- Subklassen erstellen

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

5.2. Checking Type (2)

- Array definieren

```
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick
  Astley")
]
// the type of "library" is inferred to be [MediaItem]
```

5.2. Checking Type (3)

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

print("Media library contains \(movieCount) movies and
  \(songCount) songs")
// Prints "Media library contains 2 movies and 3 songs"
```

5.3. Downcasting

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}

// Movie: Casablanca, dir. Michael Curtiz
// Song: Blue Suede Shoes, by Elvis Presley
// Movie: Citizen Kane, dir. Orson Welles
// Song: The One And Only, by Chesney Hawkes
// Song: Never Gonna Give You Up, by Rick Astley
```

5.4. Type Casting for Any and AnyObject (1)

- Any: repräsentiert eine Instanz von allen Klassen (inkl. Funktionstypen)
- AnyObject: repräsentiert eine Instanz von allen Klassen

5.4. Type Casting for Any and AnyObject (2)

- Array definieren

```
var things: [Any] = []

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan
  Reitman"))
things.append({ (name: String) -> String in "Hello, \(name)" })
```

5.4. Type Casting for Any and AnyObject (3)

```
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Michael"))
    default:
        print("something else")
    }
}
```

```
// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called Ghostbusters, dir. Ivan Reitman
// Hello, Michael
```

Quellennachweis

- https://docs.swift.org/swift-book/LanguageGuide/Functions.html
- https://docs.swift.org/swift-book/LanguageGuide/Closures.html
- https://docs.swift.org/swift-book/LanguageGuide/Subscripts.html
- https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html
- https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html

Vielen Dank für eure Aufmerksamkeit!