



BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences



MAD3

Swift Funktionen

Prof. Dr. Dragan Macos



- Codefragmente mit einer bestimmten Aufgabe.
- Definition und Aufruf

Der Name der Funktion

Der interne Name des
ersten Parameters

Typ des ersten
Parameters

Der Rückgabetyt

```
func sayHello(personName: String) -> String {  
    let greeting = "Hello, " + personName + "!"  
    return greeting  
}
```

```
println(sayHello("Anna"))
```

Aufruf

- Alle Parameter einer Funktion sind Konstanten. Sie können nicht geändert werden.
- Wenn wir diese ändern wollen, müssen wir das explizit sagen.
- ... Kommt noch



```
func halfOpenRangeLength(start: Int, end: Int) -> Int {  
    return end - start  
}
```

Mehrere Parameter

Keine Parameter

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

Kein Rückgabewert

```
func sayGoodbye(personName: String) {  
    println("Goodbye, \(personName)!")  
}
```

was ist das??



```
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)  
}
```

was ist das??



```
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)  
}
```

Funktion liefert mehrere
Werte. In ein Tupel
eingepackt.

Lokaler
Parametername . Wird
im Funktionsrumpf
verwendet.

```
func someFunction(parameterName: Int) {  
    // function body goes here, and can use parameterName  
    // to refer to the argument value for that parameter  
}
```

Lokaler
Parametername . Wird
im Funktionsrumpf
verwendet.



```
func someFunction(parameterName: Int) {  
    // function body goes here, and can use parameterName  
    // to refer to the argument value for that parameter  
}
```

```
func someFunction(externalParameterName localParameterName:  
    Int) {  
    // function body goes here, and can use  
    localParameterName  
    // to refer to the argument value for that parameter  
}
```

Externer
Parametername . Wird
beim Aufruf
verwendet.



```
func join(s1: String, s2: String, joiner: String) -> String
{
    return s1 + joiner + s2
}
```

Parameter in der Funktionsdefinition ohne externe Namen.

```
join("hello", "world", ", ")
// returns "hello, world"
```

Aufruf

```
func join(string s1: String, toString s2: String, withJoiner
    joiner: String)
-> String {
    return s1 + joiner + s2
}
```

Funktionsdefinition mit externen Namen.

```
join(string: "hello", toString: "world", withJoiner: ", ")
// returns "hello, world"
```

Aufruf

„#“ bedeutet „lokaler Name = externer Name“ 

```
func containsCharacter(#string: String, #characterToFind:
    Character) -> Bool {
    for character in string {
        if character == characterToFind {
            return true
        }
    }
    return false
}
```

```
let containsAVee = containsCharacter(string: "aardvark",
    characterToFind: "v")
// containsAVee equals true, because "aardvark" contains a
    "v"
```



```
func join(string s1: String, toString s2: String,  
  withJoiner joiner: String = " ") -> String {  
  return s1 + joiner + s2  
}
```

Initialwert

Initialwert
überschrieben.

```
join(string: "hello", toString: "world", withJoiner: "-")  
// returns "hello-world"
```

```
join(string: "hello", toString: "world")  
// returns "hello world"
```

Parameter mit Initialwert
muss im Aufruf nicht
vorkommen



```
func join(s1: String, s2: String, joiner: String = " ") ->  
    String {  
    return s1 + joiner + s2  
}  
join("hello", "world", joiner: "-")  
// returns "hello-world"
```

Wenn der Initialwert definiert ist, dann muss im Aufruf ein externer Name angegeben werden.
Wenn kein externer Name des Parameters definiert.
→ Externer Name = lokaler Name.
Vermeidung des Mechanismus: Durch „_“



- Definiert für jeden Typ. Maximal ein „variadic Parameter“ zugelassen
- Variadic Parameter muss am Ende der Parameterliste sein.
- Beispiel:

Double...

- Null oder mehrere Werte
- In der Funktion: Konstanter Array namens `numbers` mit dem Typen `Double[]`

```
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, 19)  
// returns 10.0, which is the arithmetic mean of these three  
// numbers
```



Er kann als „Variabler Parameter“ verwendet werden.

Der Parameter wird kopiert.
Die Kopie ist „änderbar“

```
func alignRight(var string: String, count: Int, pad:
Character) -> String {
    let amountToPad = count - countElements(string)
    for _ in 1...amountToPad {
        string = pad + string
    }
    return string
}

let originalString = "hello"
let paddedString = alignRight(originalString, 10, "-")
// paddedString is equal to "-----hello"
// originalString is still equal to "hello"
```



- Er hat einen Wert, den wir in die Funktion übernehmen, ändern und aus der Funktion ausgeben. Geändert.
- Keine variadic Parameter, keine var und kein let.

```
func swapTwoInts(inout a: Int, inout b: Int) {
```

```
    let temporaryA = a
```

```
    a = b
```

```
    b = temporaryA
```

```
}
```

```
var someInt = 3
```

```
var anotherInt = 107
```

```
swapTwoInts(&someInt, &anotherInt)
```

```
println("someInt is now \(someInt), and anotherInt is now \  
        (anotherInt)")
```

```
// prints "someInt is now 107, and anotherInt is now 3"
```

inout Parameter

Muss sein. Hiermit sagen wir „ein inout Parameter wird hier verwendet“



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

Zwei kleine
Funktionen

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

Typ der beiden Funktionen

```
(Int, Int) -> Int
```

```
func printHelloWorld() {  
    println("hello, world")  
}
```

Typ von *printHelloWorld*

```
() -> ()
```



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

Zwei kleine
Funktionen

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

Typ der beiden Funktionen

```
(Int, Int) -> Int
```

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

Variable

Ist eine „(Int, Int)->Int“-
Funktion

... Und das ist ihr Wert...



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

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

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

```
println("Result: \"(mathFunction(2, 3))\"")  
// prints "Result: 5"
```

```
mathFunction = multiplyTwoInts  
println("Result: \"(mathFunction(2, 3))\"")  
// prints "Result: 6"
```



```
func addTwoInts(a: Int, b: Int) -> Int {  
    return a + b  
}  
  
func multiplyTwoInts(a: Int, b: Int) -> Int {  
    return a * b  
}  
  
(Int, Int) -> Int  
  
var mathFunction: (Int, Int) -> Int = addTwoInts  
  
let anotherMathFunction = addTwoInts  
// anotherMathFunction is inferred to be of type (Int, Int)  
    -> Int
```



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

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

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

Funktionen als Liefertypen



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

```
}
```

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

```
}
```

`(Int) -> Int`

Type der beiden
Funktionen.

Die Funktion liefert..

.. Eine „(Int)→Int“ -
Funktion

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

Klar, oder???



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

dasselbe wie

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {  
    if backwards  
    {  
        return stepBackward  
    }  
    else  
    {  
        return stepForward  
    }  
}
```



```
func stepForward(input: Int) -> Int {  
    return input + 1  
}  
  
func stepBackward(input: Int) -> Int {  
    return input - 1  
}  
  
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {  
    if backwards  
    {  
        return stepBackward  
    }  
    else  
    {  
        return stepForward  
    }  
}
```



```
func stepForward(input: Int) -> Int {  
    return input + 1  
}  
  
func stepBackward(input: Int) -> Int {  
    return input - 1  
}  
  
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {  
    return backwards ? stepBackward : stepForward  
}  
  
var currentValue = 3  
let moveNearerToZero = chooseStepFunction(currentValue > 0)  
// moveNearerToZero now refers to the stepBackward()  
    function
```



```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {  
    return backwards ? stepBackward : stepForward  
}  
  
var currentValue = 3  
let moveNearerToZero = chooseStepFunction(currentValue > 0)  
// moveNearerToZero now refers to the stepBackward()  
function
```

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




```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {  
    func stepForward(input: Int) -> Int { return input + 1 }  
    func stepBackward(input: Int) -> Int { return input - 1 }  
    return backwards ? stepBackward : stepForward  
}
```

Funktion in Funktion.



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

Lesezeichen TL

- Ein Stück aufrufbarer Funktionalität.
- Wie Blocks in C und Objective-C
- Lambda Ausdrücke in anderen Programmiersprachen.
- Closures beinhalten die Werte aller Konstanten und Variablen, die in Closures vorkommen (capture, wir werden sagen „kapern“).
- Globale Funktionen sind Closures. Sie müssen nichts kapern
- Eingebettete Funktionen sind Closures. Sie kapern die Werte der Funktionen, in denen sie beinhaltet sind.
- Closures sind auch Closure-Ausdrücke in der `lightweight-Syntax`. Sie müssen die Werte Ihres Kontextes kapern.
- Kapern ☺





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

- `sorted` ist eine Swift-Standardfunktion. Sie hat zwei Parameter:
 - Einen Array von Werten mit bekanntem Typen
 - Ein Closure. Das zeigt in welcher Reihenfolge zwei Elemente sein sollen
 - Zweistellig: zwei Array-Elemente `(String, String) -> Bool`
 - Returnwert: `Bool`
 - `true`: Erster Parameter soll im sortierten Array vor dem zweiten Parameter erscheinen. Sonst `false`.

```
func backwards(s1: String, s2: String) -> Bool {  
    return s1 > s2  
}  
  
var reversed = sort(names, backwards)  
// reversed is equal to ["Ewa", "Daniella", "Chris",  
    "Barry", "Alex"]
```

backwards als Funktion

Jetzt ist noch die Frage, wie
backwards aussieht als
Closure-Ausdruck??



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

Beispiel

```
{ (s1: String, s2: String) -> Bool in  
  return s1 > s2  
}
```



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

Ein Stück Code,
direkt im
Funktionsaufruf.

Wieso ist das
gut?



Closure-Rumpf
fängt nach „in“
an.
Hier nur eine
Zeile.

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

Deshalb ist diese Achreibweise
auch nicht schlecht

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





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

Die Typen und Klammern können wir weglassen. Die kann die Typinferenz rauskriegen. Nur die Parameternamen sollen bleiben.

Der Rückgabebetyp auch.

Das kann auch weg.

```
reversed = sorted(names, { s1, s2 in return s1 > s2 } )
```




```
reversed = sorted(names, { s1, s2 in return s1 > s2 } )
```

Nur ein Ausdruck



```
reversed = sorted(names, { s1, s2 in s1 > s2 } )
```



```
reversed = sorted(names, { s1, s2 in s1 > s2 } )
```



```
reversed = sorted(names, { $0 > $1 } )
```

in Swift für jede
inline-Closure
vorhanden



- eine Funktion ist auch ein Closure
- „>“ ist eine Operator-Funktion, die auch für Strings definiert ist.

```
reversed = sorted(names, >)
```

würde auch gehen.



- Wenn eine Funktion als letzten Parameter ein Closure hat
- .. ann kann das Closure nach dem Funktionsaufruf angegeben werden. Layout-Sache

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

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

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

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

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



```
reversed = sorted(names) { $0 > $1 }
```





- klassische Funktion für Collections/Bäume...
- `map` braucht eine *Datenstruktur* und eine *Funktion*
„Wende die *Funktion* auf jedes Element der *Datenstruktur* an“
- Beispiel in Swift: der Typ *Array* hat eine *map*-Methode
- `map` von `Array` ist eine einstellige Funktion. Sie braucht ein Closure.
- `map` liefert einen neuen `Array`.
- Closure wird auf jedes Element des ursprünglichen `Arrays` angewendet.



numbers

[16, 58, 510]

?

numbers.map { *meinClosureRumpf* }



["OneSix", "FiveEight", "FiveOneZero"]

Alleine probieren?



```
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: Int) -> String in  
  
  }  
  
  return output  
}
```




```
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: Int) -> String in  
  var output = ""  
  var numberHlp = number  
  while numberHlp > 0 {  
    output = digitNames[numberHlp % 10]! + output  
    numberHlp /= 10  
  }  
  return output  
}
```

Das stört.
Wie kann man das
schicker machen?
Hier: schick = ohne
Hilfsvariable



```
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 {  
  (var number) -> String in  
  var output = ""  
  while number > 0 {  
    output = digitNames[number % 10]! + output  
    number /= 10  
  }  
  return output  
}
```

Ein paar Fragen...



```
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 {  
  (var number) -> String in  
  var output = ""  
  while number > 0 {  
    output = digitNames[number % 10]! + output  
    number /= 10  
  }  
  return output  
}
```

Wieso??

Ein paar Fragen...



```
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 {  
  (var number) -> String in  
  var output = ""  
  while number > 0 {  
    output = digitNames[number % 10]! + output  
    number /= 10  
  }  
  return output  
}
```

Wie würde es
ohne „var“
aussehen??





Lesezeichen

- „Capture“, auf Deutsch ... kapern 😊
- Mit „kapern“ meinen wir: „Einen Wert ändern.“
- Ein interessanter Fall: Den Wert einer Variablen ändern, die wir nicht definiert haben. Böse.
- .. und nun ein Beispiel. Wir sind Programmierer, keine Schriftsteller. Aber.... Philosophen sind wir irgendwie schon...





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

Was ist das??

Capture (=Kapern)



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

```
func incrementor() -> Int {  
    runningTotal += amount  
    return runningTotal  
}
```

Es wird eine schon definierte Variable der umgebenden Funktion gekapert. *incrementor* hat ihre Adresse (Referenz). Durch dieses Kapern wird die Variable „runningTotal“ nach dem Aufruf der Funktion „makeIncrementor“ nicht gelöscht. Die Referenz wird mit der Funktion „incrementor“ gespeichert.

Derselbe Wert ist auch nach dem nächsten Wert vorhanden.

Es wird der Parameter der Funktion „makeIncrementor“ gekapert. Das ist eigentlich die Kopie des Parameters (so funktionieren die Funktionsaufrufe von Swift). Die eigentliche Variable wird nicht angefasst.



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

- Was und wie wird gekapert, entscheidet Swift. Referenz, Wert, Kopie.....
- Die Speicherverwaltung erledigt Swift auch von alleine. Wir müssen nicht sagen, wann welche Speicherstellen freigegeben werden sollen. Das Speichermanagement ist vollautomatisch.



Beispiel

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

```
let incrementByTen = makeIncrementor(forIncrement: 10)  
incrementByTen()  
// returns a value of 10  
incrementByTen()  
// returns a value of 20  
incrementByTen()  
// returns a value of 30
```



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

```
incrementByTen()
```

```
// returns a value of 10
```

```
incrementByTen()
```

```
// returns a value of 20
```

```
incrementByTen()
```

```
// returns a value of 30
```

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

```
let incrementBySeven = makeIncrementor(forIncrement: 7)
```

```
incrementBySeven()
```

```
// returns a value of 7
```

```
incrementByTen()
```

```
// returns a value of 40
```

Anmerkung: Es kann passieren, dass durch kapern von Klasseninstanzen Zyklen entstehen. Swift führt Referenzlisten und lässt solche Situationen nicht zu. Diese führen zu Speicherleaks. Darüber später...



- Eine Closure-Instanz wird nicht kopiert.
- Zuweisungen, Funktionsaufrufe... bekommen immer die Referenz einer Closure-Instanz

```
let alsoIncrementByTen = incrementByTen  
alsoIncrementByTen()  
// returns a value of 50
```

Die meisten Sourcecode-Beispiele und die Sprachdefinition der Sprache Swift wurden aus:

Apple Inc. „The Swift Programming Language.“ iBooks. <https://itun.es/de/jEUH0.I>

genommen.

Eventuelle andere Quellen bzw. eigene Beispiele werden an den entsprechenden Stellen direkt angegeben bzw. gekennzeichnet.

