



MAD3

Swift Funktionen

Prof. Dr. Dragan Macos



- Codefragmente mit einer bestimmten Aufgabe.
- Definition und Aufruf

Der Name der Funktion

Der Name des ersten Parameters

Typ des ersten Parameters

Der Rückgabetyt

```
1 func greet(name: String, day: String) -> String {  
2     return "Hello \ \(name), today is \ \(day)."  
3 }
```

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



```
1  func greet(person: String) {  
2      print("Hello, \ (person)!")  
3  }  
4  greet(person: "Dave")  
5  // prints "Hello, Dave!"
```

Label

Parameter-Wert

Aufruf



```
1  func greet(person: String) {  
2      print("Hello, \ (person)!")  
3  }  
4  greet(person: "Dave")  
5  // Prints "Hello, Dave!"
```

Name der Funktion

`greet(person:)`

```

1  func printAndCount(string: String) -> Int
    {
2      print(string)
3      return string.characters.count
4  }
5  func printWithoutCounting(string: String) {
6      let _ = printAndCount(string: string)
7  }
8  printAndCount(string: "hello, world")
9  // prints "hello, world" and returns a
    value of 12
10 printWithoutCounting(string: "hello,
    world")
11 // prints "hello, world" but does not
    return a value

```

Das Ergebnis der
Funktion wird ignoriert

Optionales Return

Prüfung, ob die Funktion etwas geliefert hat.



```
1 func minMax(array: [Int]) -> (min: Int, max: Int)? {  
2     if array.isEmpty { return nil }  
3     var currentMin = array[0]  
4     var currentMax = array[0]  
5     for value in array[1..<array.count] {  
6         if value < currentMin {  
7             currentMin = value  
8         } else if value > currentMax {  
9             currentMax = value  
10        }  
11    }  
12    return (currentMin, currentMax)  
13 }
```

```
1 if let bounds = minMax(array: [8, -6, 2,  
2     109, 3, 71]) {  
3     print("min is \(bounds.min) and max is  
4         \(bounds.max)")  
5 }  
6 // Prints "min is -6 and max is 109"
```

Die Funktion muss nicht unbedingt einen Wert liefern.



```
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)
```

Parameter-Name

Parameter-Label
hier gleich wie der
Parameter-Name

Jeder Parameter hat zwei Komponenten: Parameter-Label und Parameter-Namen

```
1  func someFunction(argumentLabel parameterName: Int)
      {
2      // In the function body, parameterName refers to
          the argument value
3      // for that parameter.
4  }
```


Funktionsparameter

Spezifikation der Labels



```
1  func someFunction(argumentLabel parameterName: Int)
    {
2      // In the function body, parameterName refers to
        the argument value
3      // for that parameter.
4  }
```

Wie sieht der Aufruf der
Funktion mit dem
Parameter 10 aus?



```
1 func sayHello(to person: String, and anotherPerson: String) -> String {  
2     return "Hello \ \(person) and \ \(anotherPerson)!"  
3 }  
4  
5
```

Was macht die Funktion?
Wie sieht der Aufruf der Funktion mit den Parametern
„Bill“ und „Ted“ aus?



- Label “_” angeben.
- Dies bedeutet „ich befreie Dich von der Pflicht, beim Funktionsaufruf das Label anzugeben“.

```
1  func someFunction(_ firstParameterName: Int,  
2      secondParameterName: Int) {  
3      // In the function body, firstParameterName and  
4      // secondParameterName  
5      // refer to the argument values for the first  
6      // and second parameters.  
7  }
```

Aufruf mit den Parametern 1 und 2?



- Label “_” angeben.
- Dies bedeutet „ich befreie Dich von der Pflicht, beim Funktionsaufruf das Label anzugeben“.

```
1  func someFunction(_ firstParameterName: Int,  
2      secondParameterName: Int) {  
3      // In the function body, firstParameterName and  
4      // secondParameterName  
5      // refer to the argument values for the first  
6      // and second parameters.  
7  }  
8  someFunction(1, secondParameterName: 2)
```



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

Initialwert

Initialwert
überschrieben.



- 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[]`





```
1  func arithmeticMean(_ numbers: Double...) -> Double {
2      var total: Double = 0
3      for number in numbers {
4          total += number
5      }
6      return total / Double(numbers.count)
7  }
8  arithmeticMean(1, 2, 3, 4, 5)
9  // returns 3.0, which is the arithmetic mean of these
   five numbers
10 arithmeticMean(3, 8.25, 18.75)
11 // returns 10.0, which is the arithmetic mean of these
   three numbers
```



- Die Funktionsparameter können wir nicht ändern.
- Wenn wir dies möchten, müssen wir „Inout“-Parameter definieren.
- Keine variadic Parameter, keine default Werte

```
1  func swapTwoInts(_ a: inout Int, _ b: inout Int)
    {
2      let temporaryA = a
3      a = b
4      b = temporaryA
5  }
```



inout Parameter



```
1 func swapTwoInts(_ a: inout Int, _ b: inout Int)
    {
2     let temporaryA = a
3     a = b
4     b = temporaryA
5 }
```

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

Aufruf mit „&“



```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2     return a + b  
3 }  
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int  
5     {  
6     return a * b  
7 }
```

Zwei kleine
Funktionen

`(Int, Int) -> Int`

Typ der beiden Funktionen

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

Typ von *printHelloWorld*

`() -> Void`



```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2     return a + b  
3 }  
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int  
5     {  
6     return a * b  
7 }
```

Zwei kleine
Funktionen

`(Int, Int) -> Int`

Typ der beiden Funktionen

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

Variable

Ist eine „(Int, Int)->Int“-
Funktion

... Und das ist ihr Wert...



```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2     return a + b  
3 }  
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int  
5     {  
6     return a * b  
7 }
```

`(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"
```



```
1  func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2      return a + b  
3  }  
4  func multiplyTwoInts(_ a: Int, _ b: Int) -> Int  
    {  
5      return a * b  
6  }
```

`(Int, Int) -> Int.`

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

```
let anotherMathFunction = addTwoInts
```

```
// anotherMathFunction is inferred to be of type (Int, Int)  
    -> Int
```



```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2     return a + b  
3 }  
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int  
5     {  
6     return a * b  
7 }
```

```
1 func printMathResult(_ mathFunction: (Int, Int)  
2     -> Int, _ a: Int, _ b: Int) {  
3     print("Result: \(mathFunction(a, b))")  
4 }  
5  
6 printMathResult(addTwoInts, 3, 5)  
7  
8 // Prints "Result: 8"
```



```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2     return a + b  
3 }  
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int  
5     {  
6     return a * b  
7 }
```

```
1 func printMathResult(_ mathFunction: (Int, Int) -> Int,  
2     _ a: Int, _ b: Int) {  
3     print("Result: \"(mathFunction(a, b))\"")  
4 }  
5
```

Name der Funktion??

```
5 // Prints "Result: 8"
```



```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2     return a + b  
3 }  
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int  
5     {  
6     return a * b  
7 }
```

```
printMathResult(_:_:_:)
```

```
1 func printMathResult(_ mathFunction: (Int, Int) -> Int,  
2     _ a: Int, _ b: Int) {  
3     print("Result: \(mathFunction(a, b))")  
4 }  
5 printMathResult(addTwoInts, 3, 5)  
6 // Prints "Result: 8"
```




```
1 func stepForward(_ input: Int) -> Int {  
2     return input + 1  
3 }  
4 func stepBackward(_ input: Int) -> Int {  
5     return input - 1  
6 }
```

`(Int) -> Int`


Typ der beiden
Funktionen.

Die Funktion liefert..

.. Eine „(Int)→Int“ -
Funktion

```
1 func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
2     return backward ? stepBackward : stepForward  
3 }
```

Klar, oder???



```
1 func stepForward(_ input: Int) -> Int {
2     return input + 1
3 }
4 func stepBackward(_ input: Int) -> Int {
5     return input - 1
6 }
```

```
1 func stepForward(_ input: Int) -> Int {
2     return input + 1
3 }
4 func stepBackward(_ input: Int) -> Int {
5     return input - 1
6 }
```

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

chooseStepFunction(backward:)



```
1  func stepForward(_ input: Int) -> Int {  
2      return input + 1  
3  }  
4  func stepBackward(_ input: Int) -> Int {  
5      return input - 1  
6  }
```

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

```
1  var currentValue = 3  
2  let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)  
3  // 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
```

```
1  print("Counting to zero:")  
2  // Counting to zero:  
3  while currentValue != 0 {  
4      print("\(currentValue)... ")  
5      currentValue = moveNearerToZero(currentValue)  
6  }  
7  print("zero!")  
8  // 3...  
9  // 2...  
10 // 1...  
11 // zero!
```



```
1  func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
2      func stepForward(input: Int) -> Int { return input + 1 }  
3      func stepBackward(input: Int) -> Int { return input - 1 }  
4      return backward ? stepBackward : stepForward  
5  }
```

Funktion in Funktion.



```
1  func chooseStepFunction(backward: Bool) -> (Int) -> Int {
2      func stepForward(input: Int) -> Int { return input + 1 }
3      func stepBackward(input: Int) -> Int { return input - 1 }
4      return backward ? stepBackward : stepForward
5  }

6  var currentValue = -4
7  let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
8  // moveNearerToZero now refers to the nested stepForward() function
9  while currentValue != 0 {
10     print("\(currentValue)... ")
11     currentValue = moveNearerToZero(currentValue)
12 }
13 print("zero!")
14 // -4...
15 // -3...
16 // -2...
17 // -1...
18 // zero!
```



- 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(by:)` ist eine Swift-Standardmethode. Sie hat einen Parameter:
 - Ein Closure. Das zeigt in welcher Reihenfolge zwei Elemente sein sollen
 - Zweistellig: zwei Array-Elemente
 - Rückgabewert: Bool `(String, String) -> Bool`
 - `true`: Erster Parameter soll im sortierten Array vor dem zweiten Parameter erscheinen. Sonst `false`.

backward als
Funktion

```
1 func backward(_ s1: String, _ s2: String) -> Bool {  
2     return s1 > s2  
3 }  
4 var reversedNames = names.sorted(by: backward)  
5 // reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

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



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

Beispiel

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



```
1 func backwards(s1: String, _ s2: String) -> Bool {  
2     return s1 > s2  
3 }  
4 var reversed = names.sort(backwards)  
5 // reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

```
1 reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in  
2     return s1 > s2  
3 })
```

Ein Stück Code,
direkt im
Funktionsaufruf.

Wieso ist das
gut?



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

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

Der Rückgabebetyp auch.

Das kann auch weg.

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



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



```
reversedNames = names.sorted(by: { $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.

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

würde auch gehen.





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

```
1  func someFunctionThatTakesAClosure(closure: () -> Void)
2      {
3      }
4
5  // Here's how you call this function without using a
6      trailing closure:
7
8  someFunctionThatTakesAClosure(closure: {
9      // closure's body goes here
10 })
11
12 // Here's how you call this function with a trailing
13     closure instead:
14
15 someFunctionThatTakesAClosure() {
16     // trailing closure's body goes here
17 }
```



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

..wenn das Closure der einzige Parameter ist,
dann geht es auch ohne Klammern

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



- In Swift Standardbibliothek ist eine Variante map-Funktion vorhanden.
- klassische Funktion für Collections/Bäume...
- map braucht eine *Funktion*, die auf eine Datenstruktur verwendet wird
„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.
- Das Closure wird auf jedes Element des ursprünglichen Arrays angewendet.





numbers

[16, 58, 510]

?

numbers.map { *meinClosureRumpf* }

Typ des Closure (Int)->String

["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]
```

```
1  let strings = numbers.map {  
2    (number) -> String in  
3  
4  
5  
6  
7  
8  
9    return output  
10 }
```



```
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]
```

```
1  let strings = numbers.map {  
2      (number) -> String in  
3      var number = number  
4      var output = ""  
5      repeat {  
6          output = digitNames[number % 10]! + output  
7          number /= 10  
8      } while number > 0  
9      return output  
10 }  
11 // strings is inferred to be of type [String]  
12 // its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

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]
```

Wieso??

```
1  let strings = numbers.map {  
2    (number) -> String in  
3    var number = number  
4    var output = ""  
5    repeat {  
6      output = digitNames[number % 10]! + output  
7      number /= 10  
8    } while number > 0  
9    return output  
10 }  
11 // strings is inferred to be of type [String]  
12 // its value is ["OneSix", "FiveEight", "FiveOneZero"]
```



- „Capture“, auf Deutsch ... kapern 😊
- 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...



Capture (=Kapern)



```
1  func makeIncrementer(forIncrement amount: Int) -> () -> Int
    {
2      var runningTotal = 0
3      func incrementer() -> Int {
4          runningTotal += amount
5          return runningTotal
6      }
7      return incrementer
8  }
```

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.

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.



```
1  func makeIncrementer(forIncrement amount: Int) -> () -> Int
    {
2      var runningTotal = 0
3      func incrementer() -> Int {
4          runningTotal += amount
5          return runningTotal
6      }
7      return incrementer
8  }
```

- Was und wie wird gekapert, entscheidet Swift. Referenz, Wert, Kopie.....
- Die Speicherverwaltung erledigt Swift auch von alleine.
- Wir müssen nicht steuern, 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()
```

1 Was wird hier geliefert?

```
incrementByTen()
```

2 Was wird hier geliefert?

```
incrementByTen()
```

3 Was wird hier geliefert?

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()
```

Was wird hier geliefert?

```
incrementByTen()
```

Was wird hier geliefert?

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...



```
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()
```

Was wird hier geliefert?



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

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





```
1  var completionHandlers: [() -> Void] = []
2  func someFunctionWithEscapingClosure(completionHandler:
3      @escaping () -> Void) {
4      completionHandlers.append(completionHandler)
5  }
```

Escaping Closure.
Wird nach dem Funktionsreturn
aufgerufen. Es wird explizit auf self
innerhalb des Closures referenziert.

Das Closure ist jetzt erzeugt und wird
Später aufgerufen.

Escaping Closure



```
1  var completionHandlers: [() -> Void] = []
2  func someFunctionWithEscapingClosure(completionHandler:
    @escaping () -> Void) {
3      completionHandlers.append(completionHandler)
4  }
```

```
1  func someFunctionWithNonEscapingClosure(closure: () -> Void) {
2      closure()
3  }
4
5  class SomeClass {
6      var x = 10
7      func doSomething() {
8          someFunctionWithEscapingClosure { self.x = 100 }
9          someFunctionWithNonEscapingClosure { x = 200 }
10     }
11 }
12
13 let instance = SomeClass()
14 instance.doSomething()
15 print(instance.x)
16 // Prints "200"
17
18 completionHandlers.first?()
19 print(instance.x)
20 // Prints "100"
```




- Ausdrücke werden in die Closures „eingehüllt“.
- Auf diese Art werden aus den Ausdrücken automatisch Closures gemacht.
- Sie haben keine Parameter und liefern den Wert des „eingehüllten“ Ausdrucks.





```
1  var customersInLine = ["Chris", "Alex", "Ewa", "Barry",  
    "Daniella"]  
  
2  print(customersInLine.count)  
3  // Prints "5"  
4  
5  let customerProvider = { customersInLine.remove(at: 0) }  
6  print(customersInLine.count)  
7  // Prints "5"  
8  
9  print("Now serving \(customerProvider())!")  
10 // Prints "Now serving Chris!"  
11 print(customersInLine.count)  
12 // Prints "4"
```



```
1  var customersInLine = ["Chris", "Alex", "Ewa", "Barry",  
    "Daniella"]  
2  print(customersInLine.count)  
3  // Prints "5"  
4  
5  let customerProvider = { customersInLine.remove(at: 0) }  
6  print(customersInLine.count)  
7  // Prints "5"  
8  
9  print("Now serving \(customerProvider())!")  
10 // Prints "Now serving Chris!"  
11 print(customersInLine.count)  
12 // Prints "4"
```

Die Variable ist durch Autoclosure definiert. Sie hat den Typ ()->String

remove(at:) liefert das aus dem Array „rausgeschmissene“ Element.



```
1 // customersInLine is ["Alex", "Ewa", "Barry",  
    "Daniella"]  
2 func serve(customer customerProvider: () -> String) {  
3     print("Now serving \(customerProvider())!")  
4 }  
5 serve(customer: { customersInLine.remove(at: 0) } )  
6 // Prints "Now serving Alex!"
```



```
1 // customersInLine is ["Alex", "Ewa", "Barry",  
   "Daniella"]  
2 func serve(customer customerProvider: () -> String) {  
3     print("Now serving \(customerProvider())!")  
4 }  
5 serve(customer: { customersInLine.remove(at: 0) } )  
6 // Prints "Now serving Alex!"
```

Hier als Funktionsparameter

```
1 // customersInLine is ["Ewa", "Barry", "Daniella"]  
2 func serve(customer customerProvider: @autoclosure () -> String) {  
3     print("Now serving \(customerProvider())!")  
4 }  
5 serve(customer: customersInLine.remove(at: 0))  
6 // Prints "Now serving Ewa!"
```



```
1 // customersInLine is ["Barry", "Daniella"]
2 var customerProviders: [() -> String] = []
3 func collectCustomerProviders(_ customerProvider:
4     @autoclosure @escaping () -> String) {
5     customerProviders.append(customerProvider)
6 }
7 collectCustomerProviders(customersInLine.remove(at: 0))
8 collectCustomerProviders(customersInLine.remove(at: 0))
9
10 print("Collected \(customerProviders.count) closures.")
11 // Prints "Collected 2 closures."
12 for customerProvider in customerProviders {
13     print("Now serving \(customerProvider())!")
14 }
15 // Prints "Now serving Barry!"
16 // Prints "Now serving Daniella!"
```



1. Wie sehen folgende Funktionsaufrufe aus

Funktion

```
func addTwoInts(a: Int, _ b: Int) -> Int
```

mit den Parametern 10 und 50 ?

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

Mit den Parametern 10 und 50?

```
func someFunction(parameterWithDefault: Int = 12)
```

- Mit dem Parameter 10?
- Mit dem Parameterwert 12?



Funktion

```
func arithmeticMean(numbers: Double...) -> Double
```

mit den Parametern 10.1 und 50.1 ? Mit dem Parameter 1.1?

```
func someFunction(firstParameterName: Int, _ secondParameterName: Int)
```

Mit den Parametern 10 und 50?

```
func sayHello(to person: String, and anotherPerson: String) -> String
```

- Mit den Parametern „Billie“ und „Willie“?

```
func someFunction(externalParameterName localParameterName: Int)
```

- mit dem Parameter 2.



Funktion

```
func someFunction(firstParameterName: Int, secondParameterName: Int)
```

mit Parametern 10 und 50?

```
func minMax(array: [Int]) -> (min: Int, max: Int)
```

Mit dem Parameter [1, 2, 3]?



2. Die Funktion *map* mit einem *Closure* aufrufen

- Definieren Sie für die Standardfunktion *map* ein Closure, das für einen beliebigen Array *myArray*: *[Int]* einen neuen Array *newArray*: *[Int]* liefert, dessen Elemente die Quadrate von den Elementen von *myArray* sind.
- Das Closure soll als Parameter der eingebauten Funktion *map* verwendbar sein. Rufen Sie die Funktion *map* entsprechend auf und ordnen Sie dem *newArray* den gelieferten Wert zu (s. dazu Beispiel aus der Vorlesungspräsentation). Dabei soll gelten:

$$newArray[i] = myArray[i] * myArray[i]$$





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.

