



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN



Praktikum iOS-Entwicklung

Sommersemester 2017

Prof. Dr. Linnhoff-Popien

Lenz Belzner, Kyrill Schmid



Gruppentreffen

Wir suchen Ideen für die Praxisphase.

Das heißt:

- Eure Ideen sind gefragt!
- Besprechung der Ideen und Feature-Listen mit uns
- Einzelne Termine in der Zeit vom 15. bis 19. Mai
- Danach Beginn der Programmierphase





LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN



SWIFT

Swift ist eine neue Programmiersprache

- Entwickelt seit 2010
- Erste in Swift geschriebene App im Juni 2014 im AppStore veröffentlicht

Swift ist

- objektorientiert,
- funktional,
- imperativ.

Wurde 2015 in Version 2.0 veröffentlicht

- verbessertes Error Handling
- neue Keywords und Funktionalitäten (do, try, catch, guard, repeat, ...)
- #available-checks
- automatisch generierte „Header files“

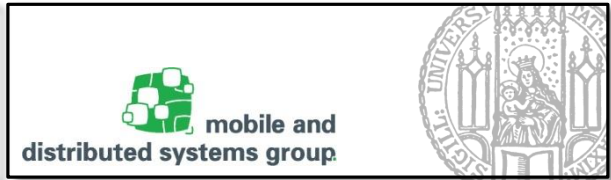
Wurde 2016 in Version 3.0 veröffentlicht

- Neuen API-Designrichtlinien zur Verbesserung der Übersichtlichkeit
- Quellcode nicht mehr kompatibel mit Vorgängerversionen



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Swift - Playground



Swift - Playground

```
class Person {
    var alter: Int           //kein Semikolon nötig
    var groesse: Double;    //aber möglich

    init(){
        self.alter = 23
        self.groesse = 1.78
    }

    init(alter:Int, groesse:Double){
        self.alter = alter
        self.groesse = groesse
    }
}

let erwin = Person()
erwin.alter //23

let peter = Person(alter:25, groesse:1.85)
peter.groesse = 1.90
peter.groesse // 1.90
```

```
class Person {  
    var alter: Int  
    var groesse: Double  
  
    init(){  
        self.alter = 23  
        self.groesse = 1.78  
    }  
  
    init(alter:Int, groesse:Double){  
        self.alter = alter  
        self.groesse = groesse  
    }  
}  
  
let erwin = Person()  
erwin.alter //23  
  
let peter = Person(alter:25, groesse:1.85)  
peter.groesse = 1.90  
peter.groesse // 1.90
```

Die Klasse Person und
die Initialisierung

```
class Person {  
    var alter: Int  
    var groesse: Double  
  
    init(){  
        self.alter = 23  
        self.groesse = 1.78  
    }  
  
    init(alter:Int, groesse:Double){  
        self.alter = alter  
        self.groesse = groesse  
    }  
}  
  
let erwin = Person()  
erwin.alter //23  
  
let peter = Person(alter:25, groesse:1.85)  
peter.groesse = 1.90  
peter.groesse // 1.90
```

Es gibt einen Unterschied
zwischen `var` und `let`!


```
class Person {  
    var alter: Int  
    var groesse: Double  
  
    init(){  
        self.alter = 23  
        self.groesse = 1.78  
    }  
  
    init(alter:Int, groesse:Double){  
        self.alter = alter  
        self.groesse = groesse  
    }  
}  
  
let erwin = Person()  
erwin.alter //23  
  
let peter = Person(alter:25, groesse:1.85)  
peter.groesse = 1.90  
peter.groesse // 1.90
```

Implizite getter und setter!

```
class Student: Person {  
    let matrikelNr: Int  
  
    init(matrikelNr: Int){  
        self.matrikelNr = matrikelNr  
        super.init()  
    }  
}
```

```
let hans = Student(1234567)  
hans.matrikelNr = 1234568 // Cannot assign to "matrikelNr" in "hans"  
hans.alter // 23
```

```
class Student: Person {  
    let matrikelNr: Int  
  
    init(matrikelNr: Int){  
        self.matrikelNr = matrikelNr  
        super.init()  
    }  
}
```

„Student“ erbt von „Person“

```
let hans = Student(1234567)  
hans.matrikelNr = 1234568 // Cannot assign to "matrikelNr" in "hans"  
hans.alter // 23
```

`let` deklariert eine Konstante

`var` deklariert eine Variable

```
let π = 3.14159
var ♣ = "Ich bin ein Kleeblatt"

π = 4 // compile-time error
♣ = "Ich bin eine Rose" // ♣ ist jetzt eine Rose

print("Der Wert von pi lautet \(\pi)\") // Ausgabebefehl
```

Variablen (und Konstanten) haben **immer** einen Typ – Swift ist *type safe*.

Typen können bei der Deklaration von Variablen explizit angegeben werden.

Swift erschließt andernfalls den Typ bei der Initialisierung der Variable.

Typ-Aliase sind alternative Namen für existierende Typen.

Einfache Datentypen sind immer *value types* (auch *Array*, *String* und *Dictionary*).

```
let  $\pi$  = 3.14159 // Swift folgert, dass  $\pi$  vom Typ Double ist  
var  $\clubsuit$ :String
```

```
 typealias Zeichenkette = String  
var word:Zeichenkette
```

```
 $\clubsuit$  = "Ich bin ein Kleeblatt"
```

```
neues $\clubsuit$  =  $\clubsuit$ 
```

```
// Der Wert von  $\clubsuit$  wird kopiert und im Speicher nochmal als neues $\clubsuit$  abgelegt
```

Ein *optional* sagt:

- „Diese Variable hat einen Wert und der ist gleich x“ oder
- „Diese Variable hat keinen Wert“

Optionals können mit `nil` in einen „wertelosen Zustand“ versetzt werden.

(In Obj-C ist `nil` ein Pointer auf ein nicht existierendes Objekt. In Swift steht es für „keinen Wert“)

```
let possibleNumber:String = "123"  
let convertedNumber = Int(possibleNumber)
```

`Int()` wird mit einem String initialisiert. Liefert einen `Int?` oder `optional Int` zurück.


```
var statusCode: Int? = 500  
statusCode = nil
```

```
var anotherCode: Int = 404  
anotherCode = nil // compiler Fehler
```

`statusCode` enthält jetzt keinen Wert mehr. `nil` kann deswegen auch nur mit *optionals* verwendet werden!

Das erzwungene Entpacken eines *optionals* heißt *forced unwrapping* und wird durch ein Ausrufezeichen ausgeführt.

Optional binding ist das temporäre Initialisieren einer Variable.

```
if convertedNumber != nil {
    print("converted Number enthält den Wert \(convertedNumber!)")
}
let possibleString: String? = "An optional string"
let forcedString: String = possibleString!

```



```
if let realNumber = possibleNumber.toInt() {
    print("Der String enthält die Zahl \(realNumber)");
}else{
    print("'\(possibleNumber)' enthält keinen gültigen Integer-Wert")
}
//Problem: realNumber steht hier nicht mehr zur Verfügung
```

Mittels *optional chaining* kann auf Properties von optionals zugegriffen werden, ohne dass ggf. Runtime-Errors entstehen.

Seit Swift 2.0 gibt es als Alternative zum optional binding das *guard* Keyword.

```
let roomCount = john.residence!.numberOfRooms //Error, falls residence nil

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
//roomCount wäre hier wieder nicht mehr verfügbar

guard let unwrappedName = userName else {
    return
}
print("Your username is \(unwrappedName)") //unwrappedName jetzt verfügbar
```


Tupel gruppieren mehrere Werte in einem.

Die einzelnen Typen dürfen unterschiedlich sein.

Funktionen können so mehrere Werte auf einmal als Rückgabewert liefern.

```
let bestNote = (1.0, "sehr gut") // bestNote ist vom Typ (Double, String)

let bestanden = (1.0, 2.0, 3.0, 4.0)
// bestanden ist vom Typ (Double, Double, Double, Double)

let (note, beschr) = bestNote
print("Die Bestnote ist eine \(note) und wird geschrieben als \(beschr)")

let einsKommaNull = bestanden.0 // "entpackt" den ersten Wert

let durchgefallen = (note: 5.0, beschreibung: "mangelhaft")
print("Eine \(durchgefallen.note) reicht nicht")
```

In Swift sind Klassen und structures sehr ähnlich. **Beide** können:

- properties definieren, um Werte zu speichern,
- Initialisierer definieren, um initialen Status zu bestimmen,
- Methoden enthalten,
- Subscripts definieren,
- mittels Extensions erweitert werden und protokollkonform entwickelt werden.

Allerdings können **nur Klassen**

- von anderen Klassen erben,
- mittels *type casting* als bestimmte Klassen zur Laufzeit interpretiert werden,
- deinitialisiert werden und
- unterstützt durch *reference counting* verwaltet werden.

```
struct Punkt {
    var x = 0
    var y = 0
}

class Kreis {
    var punkt = Punkt()
    var radius = 0.0
}

let pkt = Punkt()
let pkt2 = Punkt(x:25, y:50)
print("Die x-Koordinate des Punktes lautet \"(pkt.x)\")

let krs = Kreis()

print("Der Kreisradius ist \"(krs.radius)\")

print("Die x-Koordinate des Kreises lautet \"(krs.punkt.x)\")
krs.punkt.x = 14
print("... und jetzt \"(krs.punkt.x)\")
```

Der wichtigste Unterschied zwischen *structures* und Klassen:

- *structures* sind *value types*
- Klassen sind *reference types*

```
var pkt = Punkt(x:50, y:25)
var pkt2 = pkt

pkt2.x = 25
// pkt2.x ist 25, pkt.x immer noch 50

let krs = Kreis()
let krs2 = krs // hier wird nur die Referenz auf krs kopiert

krs2.radius = 100
// sowohl krs2.radius als auch krs.radius haben jetzt den Wert 100

krs === krs2 // true, da beide dieselbe Instanz referenzieren
pkt !== pkt2 // genauso true, da Instanzen unterschiedlich sind
```

Der wichtigste Unterschied zwischen *structures* und Klassen:

- *structures* sind *value types*
- Klassen sind *reference types*

```
var pkt = Punkt(x:50, y:25)
var pkt2 = pkt
```

```
pkt2.x = 25
// pkt2.x ist 25, pkt.x immer noch 50
```

```
let krs = Kreis()
let krs2 = krs // hier wird nur die Referenz auf krs kopiert
```

```
krs2.radius = 100
// sowohl krs2.radius als auch krs.radius haben jetzt den Wert 100
```

```
krs === krs2 // true, da beide dieselbe Instanz referenzieren
pkt !== pkt2 // genauso true, da Instanzen unterschiedlich sind
```

Die Variablen `krs` und `krs2` sind intern Pointer, wie sie aus C bekannt sind.

Der wichtigste Unterschied zwischen *structures* und Klassen:

- *structures* sind *value types*
- Klassen sind *reference types*

```
var pkt = Punkt(x:50, y:25)  
var pkt2 = pkt
```

```
pkt2.x = 25  
// pkt2.x ist 25, pkt.x immer noch 50
```

```
let krs = Kreis()  
let krs2 = krs // hier wird nur die Referenz auf krs kopiert
```

```
krs2.radius = 100  
// sowohl krs2.radius als auch krs.radius haben jetzt den Wert 100
```

```
krs === krs2 // true, da beide dieselbe Instanz referenzieren  
pkt !== pkt2 // genauso true, da Instanzen unterschiedlich sind
```

Die Klassenvariablen können als Konstanten definiert werden: Die Zuweisung ändert nichts an dem Inhalt von `krs`, sondern an dem Inhalt von `krs.radius` – anders als bei `pkt`

Wann sollte man was verwenden?

structures eignen sich besonders für

- die Kapselung weniger einfacher Datentypen (die ihrerseits *value types* sind),
- Daten, die eher direkt kopiert werden als nur ihre Referenz,
- Strukturen, die keine Vererbung benötigen.

Ein gutes Beispiel ist die Punkt-Modellierung von den vorherigen Folien.

In jedem anderen Fall empfehlen sich Klassen.

Properties sind Variablen einer structure, einer Klasse oder einer Enumeration.

Swift unterscheidet zwei Sorten von Properties:

- *stored* properties und
- *computed* properties


```
struct FixedLengthInterval{  
    var start: Int  
    let length: Int  
}
```

stored properties



```
var rangeOfThree = FixedLengthInterval(start: 0, length:3)  
rangeOfThree.start = 6
```

```
rangeOfThree.length = 4 //compiler error
```

```
class DataImporter{  
    lazy var loader = LoadMyStuff()  
    var fileToLoadFrom: String  
}  
  
let importer = DataImporter()  
importer.fileToLoadFrom = "/path/to/file"  
  
importer.loader.load()
```

lazy stored properties
werden erst initialisiert,
wenn tatsächlich auf sie
zugegriffen wird.

```
struct Interval{
    let start: Int
    var length: Int
    var center: Int {
        get {
            return start + length / 2
        }
        set(newCenter) {
            length = (newCenter - start) * 2
        }
    }
}
```

center wird nicht gespeichert, sondern über get und set bei jedem Aufruf berechnet.

```
var interval = Interval(start: 1, length:4)
interval.center // 3
interval.center = 2
interval.length // 2
```

```
struct Interval{
  let start: Int
  var length: Int
  var center: Int {
    get {
      return start + length / 2
    }
  }
}
```

Setter sind optional.
center ist jetzt eine *read-only computed property*.

```
var interval = Interval(start: 1, length:4)
interval.center // 3
```

Type properties entsprechen statischen Variablen in Objective-C.

Alle Instanzen haben Zugriff auf denselben Wert einer *type property*.

Sowohl *stored* als auch *computed properties* können *type properties* sein.

```
struct SomeStructure {
    static var staticValue = "Some static value."
    static var staticComputation: Int {...}
}

class SomeClass{
    class var staticComputation : Int {...}
/*
    class var staticValue = "Some other value"
    gives compile error: class variables not yet supported
*/
}
```

Swift bietet für jede Property je zwei Observer, die immer aufgerufen werden, wenn der Wert der Property verändert wird:

- `willSet`
- `didSet`

```
struct Progress {
    var status: Int = 0 {
        willSet {
            print("Kurz davor, \(newValue)% abzuschließen")
        }
        didSet {
            if( status > 100 ){
                status = 100
            }
            print("Seit dem letzten Aufruf \(oldValue - status)% weiter")
        }
    }
}
var p = Progress()
p.status = 10
```

Swift bietet für jede Property je zwei Observer, die immer aufgerufen werden, wenn der Wert der Property verändert wird:

- `willSet`
- `didSet`

```
struct Progress {
    var status: Int = 0 {
        willSet {
            print("Kurz davor, \(newValue)% abzuschließen")
        }
        didSet {
            if( status > 100 ){
                status = 100
            }
            print("Seit dem letzten Aufruf \(oldValue - status)% weiter")
        }
    }
}

var p = Progress()
p.status = 10
```

`newValue` und `oldValue` stehen implizit zur Verfügung

```
func sayHello(personName: String) -> String {  
    return "Hello, " + personName + "!"  
}  
  
print(sayHello("Anna")) // Hello, Anna!
```



```
func sayHello(personName: String) -> String {  
    return "Hello, " + personName + "!"  
}  
  
print(sayHello("Anna")) // Hello, Anna!
```

Diese Funktion erwartet einen Parameter vom Typ String und gibt einen String zurück.

Funktionen und Methoden können:

- einen, mehrere oder keine Parameter erwarten
- optionale Parameter erwarten
- ein, mehrere oder keine Ergebnisse zurückliefern
- optionale Ergebnisse zurückliefern

```
func count(from: Int, to: Int) -> Int {...}
```

```
func saySomething() -> String {...}
```

```
func doNothing() {...}
```

```
func getStatusAndDescription() -> (status:Int, description: String) {...}
```

```
func calculateAnOptionalResult(from: Int?) -> (res: Int, res2: Int)? {...}
```

Parameter können

- von außen sichtbare Namen haben, um lesbareren Code zu erzeugen und
- mit Standardwerten belegt werden

```
// Anstelle von
func join(s1: String, s2: String, j:String = " ") -> String {
    return s1 + j + s2
}

// kann man schreiben:
func join(string s1: String, andString s2: String, withJoiner j:String = " ")
    -> String {
    return s1 + j + s2
}

//oder kürzer:
func join(#string: String, #andString: String, #withJoiner: String = " ")
    -> String{
    return string + withJoiner + andString
}
join(string: "hello", andString: "world", withJoiner: ", ") //hello, world
```

Zum Weiterlesen zu Funktionen und Methoden:

- Ein *variadic* Parameter enthält null oder mehr Werte eines bestimmten Typs
- Variable Parameter können innerhalb des Funktionsrumpfs verändert werden
- *in-out*-Parameter können „dauerhaft“ in der Funktion verändert werden
- Funktionen beschreiben selbst Typen und können damit Parameter und Rückgabewert sein

```
func sum(numbers: Int...) -> Int {...} // variadic Parameter
```

```
func padWithZeros(var string: String, count: Int) -> String {...}  
// der Parameter string ist innerhalb der Methode veränderbar
```

```
func swapTwoInts(inout a: Int, inout b: Int) {...}  
// der Speicherbereich von a und b kann direkt manipuliert werden  
swapTwoInts(&one, &two)
```

```
func doTheMagic(with: Int, and: String) -> Bool {...}  
func caller(theFunction: (Int, String) -> Bool, somethingElse: Int){...}
```

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \$(currentSpeed) km/h"
    }
    func makeNoise(){}
}

class Train: Vehicle {
    var numWaggons = 0
    override func makeNoise(){
        print("Mööööp")
    }
}

class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \$(gear)"
    }
}
```

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \$(currentSpeed) km/h"
    }
    func makeNoise(){}
}
```

Train und Car erben von der Basisklasse Vehicle.

Damit erben Train und Car alle Variablen und Methoden.

```
class Train: Vehicle {
    var numWaggons = 0
    override func makeNoise(){
        print("Möööp")
    }
}
```

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \$(gear)"
    }
}
```

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \$(currentSpeed) km/h"
    }
    func makeNoise(){}
}

class Train: Vehicle {
    var numWaggons = 0
    override func makeNoise(){
        print("Mööööp")
    }
}

class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \$(gear)"
    }
}
```

Zum Überschreiben bestimmter Variablen oder Methoden dient das Schlüsselwort **override**. Innerhalb des überschreibenden Blocks kann auf die entsprechende Implementierung der Basisklasse mittels **super** zugegriffen werden.

Um Vererbung zu verhindern, kann das Schlüsselwort `final` eingesetzt werden.

```
final class { ... }
```

```
final var
```

```
final func
```

```
...
```


Swift unterscheidet die Initialisierung von *value types* und *reference types*.
In beiden Fällen dient `init()` als Initialisierungsmethode.

```
struct Celsius {  
    var temperature: Double = 32.0  
}
```

```
struct Celsius {  
    var temperature: Double = 32.0  
}
```

Kurzform für

```
var temperature: Double  
init() {  
    temperature = 32.0  
}
```

```
struct Celsius {  
    var temperature: Double = 32.0  
  
    init(fromFahrenheit fahrenheit:Double){  
        temperature = (fahrenheit - 32.0) / 1.8  
    }  
  
    init(fromKelvin kelvin: Double) {  
        temperature = kelvin - 273.15  
    }  
}
```

```
let test1 = Celsius(fromFahrenheit: 180)  
let test2 = Celsius(fromKelvin: -273.15)
```

```
let test3 = Celsius(37.0)
```

```
struct Celsius {  
    var temperature: Double = 32.0  
  
    init(fromFahrenheit fahrenheit: Double){  
        temperature = (fahrenheit - 32.0) / 1.8  
    }  
  
    init(fromKelvin kelvin: Double) {  
        temperature = kelvin - 273.15  
    }  
}  
  
let test1 = Celsius(fromFahrenheit: 180)  
let test2 = Celsius(fromKelvin: -273.15)  
  
let test3 = Celsius(37.0) // compile error
```

Mehrere `init()`-Methoden ermöglichen angepasste Initialisierung. Implementierungen unterscheiden sich nur in externen Parameternamen. Deshalb können sie nicht ohne aufgerufen werden!

```
struct Celsius {
    var temperature: Double = 32.0

    init(fromFahrenheit fahrenheit:Double){
        temperature = (fahrenheit - 32.0) / 1.8
    }

    init(fromKelvin kelvin: Double) {
        temperature = kelvin - 273.15
    }

    init(_ celsius: Double) {
        temperature = celsius
    }
}

let test3 = Celsius(37.0)
```

```
struct Celsius {  
    var temperature: Double = 32.0  
  
    init(fromFahrenheit fahrenheit:Double){  
        temperature = (fahrenheit - 32.0) / 1.8  
    }  
  
    init(fromKelvin kelvin: Double) {  
        temperature = kelvin - 273.15  
    }  
  
    init(_ celsius: Double) {  
        temperature = celsius  
    }  
}  
  
let test3 = Celsius(37.0)
```

Die spezielle `init()`-
Implementierung mit `init(_ ...)`
ermöglicht Initialisierung ohne
externen Parameternamen.

Reference types haben zwei Arten von Initialisierern:

- *Designated initializer* dienen als primäre Initialisierer. Klassen haben häufig nur einen *designated initializer*.
- *Convenience initializer* sind optional, können aber in bestimmten Fällen hilfreich sein

```
init( parameter ) {  
    ...  
}  
  
convenience init( parameter ) {  
    ...  
}
```

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

    convenience init() {
        self.init(name: "Was leckeres")
    }
}
```

```
var pizza = Essen()
pizza.name // Was leckeres
```

Der convenience Initialisierer belegt die Instanz mit Standardwerten. Er muss die Initialisierung mittels `self.init` delegieren!


```
class Zutat: Essen{
    var menge: Int
    init(name: String, menge: Int){
        self.menge = menge
        super.init(name: name)
    }
    override convenience init(name: String){
        self.init(name: name, menge: 1)
    }
}
```

```
var schinken = Zutat()
schinken.name // ??
```

```
class Zutat: Essen{
    var menge: Int
    init(name: String, menge: Int){
        self.menge = menge
        super.init(name: name)
    }
    override convenience init(name: String){
        self.init(name: name, menge: 1)
    }
}
```

```
var schinken = Zutat()
schinken.name // ??
```

`init(name: String)` ist ein convenience Initialisierer, der Zutaten mit der Standardmenge 1 initialisiert.

```
class Zutat: Essen{
    var menge: Int
    init(name: String, menge: Int){
        self.menge = menge
        super.init(name: name)
    }
    override convenience init(name: String){
        self.init(name: name, menge: 1)
    }
}
```

```
var schinken = Zutat()
schinken.name // ??
```

`init(name: String)` hat aber gleichzeitig dieselbe Signatur wie der *designated Initializer* der Klasse `Essen`.

```
class Zutat: Essen{
    var menge: Int
    init(name: String, menge: Int){
        self.menge = menge
        super.init(name: name)
    }
    override convenience init(name: String){
        self.init(name: name, menge: 1)
    }
}
```

```
var schinken = Zutat()
schinken.name // Was leckeres
```

`init(name: String)` hat aber gleichzeitig dieselbe Signatur wie der *designated Initializer* der Klasse `Essen`.

```
class Zutat: Essen{
    var menge: Int
    init(name: String, menge: Int){
        self.menge = menge
        super.init(name: name)
    }
    override convenience init(name: String){
        self.init(name: name, menge: 1)
    }
}
```

```
var schinken = Zutat()
schinken.name // Was leckeres
schinken.menge // ???
```

Deshalb werden alle *convenience initialisierer* der Elternklasse auch von `Zutat` geerbt

```
class Zutat: Essen{
    var menge: Int
    init(name: String, menge: Int){
        self.menge = menge
        super.init(name: name)
    }
    override convenience init(name: String){
        self.init(name: name, menge: 1)
    }
}
```

```
var schinken = Zutat()
schinken.name // Was leckeres
schinken.menge // 1
```

Swift Regeln für die Vererbung von Initialisierern unter:

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html
-> Automatic Initializer Inheritance

- Vorstellung des 1. Übungsblatts
- Model-View-Controller Pattern