

Programmierung in iOS mit Swift Studiengang MI

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- mwilhelm@hs-harz.de
- <http://mwilhelm.hs-harz.de>
- Raum 2.202
- Tel. 03943 / 659 338

Gliederung

Überblick:

- Einleitung, Geschichte, xcode
- **Sprache**
 - elementare Datentypen
 - Variablen
 - Kontrollstrukturen (if, while, for)
 - Arrays, Dictionary, Set, Tupel.
 - Funktionen, Rekursionen, Parameter
 - Klassen, Methoden und Objekte, Protokolle=Interface
 - Funktionale Programmierung
 - try, catch
- **Playground**
- **Grafische Oberfläche**

Links

- <http://swiftdoc.org/>
- https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309
- <https://swift.org/blog/swift-2-2-released/>

Beispiel mit Swift

```
var implicitInteger = 70
```

varianter Datentyp

```
var implicitDouble = 70.0
```

```
var explicitDouble: Double = 70
```

Double Datentyp, Pascal

```
let apples = 3
```

Basic

```
let oranges = 5
```

```
let appleSummary = "Ich habe \ \(apples) Äpfel."
```

Variable: apples (%d)

```
let fruitSummary = "Ich habe \ \(apples + oranges) Früchte."
```

```
print("Hallo Welt!")
```

```
let people = ["Anna": 67, "Julia": 8, "Hans": 33, "Peter": 25]
```

Array

```
for (name, age) in people {
```

foreach

```
    print("\ \(name) ist \ \(age) Jahre alt.")
```

Variable: name, age

```
}
```

Swift: Kommentare:

■ Allgemeine Kommentare

- // nur eine Zeile
- /* Bereich Anfang
- */ Bereich Ende

■ Formatierte Kommentare:

- **italic**
- ****bold****
- ``listingfont``
- `#head1#`
- `##head2##`
- `###head3###`
- `* Listing1`
- `* Listing2`
- `* Listing3`

https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309

Swift: Variablen

Deklaration	Zuweisung
<code>var implicitInteger = 70</code>	<code>implicitInteger = 2.0</code>
<code>var implicitDouble = 70.0</code>	<code>implicitDouble = 70.1</code>
<code>var explicitDouble: Double = 70</code>	<code>explicitDouble = 70.2</code>
<code>let constantDouble: Double = 42</code>	<code>constantDouble = 43</code>
<code>var optionalDouble: Double? = nil</code>	<code>optionalDouble = 1234.56</code> <code>if let definiteDouble=optionalDouble {</code> <code>definiteDouble</code> <code>}</code>

Swift: Datentypen

- Any Allgemeiner Datentyp
- Int
- Float, Double, Float32, Float64, Float80
- Bool
- String
- struct, tupel
- Class, AnyObject

Swift: Datentypen

- Any Allgemeiner Datentyp
- Int
 - UInt8 8-Bit, unsigned-int $0 \rightarrow +255$
 - UInt16 16-Bit, unsigned-int $0 \rightarrow +65535$
 - UInt32 32-Bit, unsigned-in $0 \rightarrow +4294967295$
 - UInt64 64-Bit, signed-int $0 \rightarrow +18446744073709551614$
 - Int8 8-Bit, unsigned-int $-128 \rightarrow +127$
 - Int16 16-Bit, unsigned-int $-32768 \rightarrow 32767$
 - Int32 32-Bit, signed-int $-2.147.483.648 \rightarrow +2.147.483.647$
 - Int 64-Bit, signed-int $-9223372036854775808 \dots 9223372036854775807$
- Floating-Point
 - Float,, Float32
 - Double, Float64
 - Float80
- Bool
- String
- struct, tupel

Swift: Datentypen:

- **UInt8**
 - 8-Bit, unsigned-int $0 \rightarrow +255$
- **UInt16**
 - 16-Bit, unsigned-int $0 \rightarrow +65535$
- **UInt32**
 - 32-Bit, unsigned-int $0 \rightarrow +4294967295$
- **UInt64**
 - 64-Bit, signed-int $0 \rightarrow + 18446744073709551614$

Swift: Datentypen:

■ Int

- Je nach Betriebssystem
- 32-Bit, signed-int -2.147.483.648 -> +2.147.483.647
- 64-Bit, signed-int -9223372036854775808 ... 9223372036854775807

■ Int8

- 8-Bit, unsigned-int -128 → +127

■ Int16

- 16-Bit, unsigned-int -32768 → 32767

■ Int32

- 32-Bit, signed-int -2.147.483.648 -> +2.147.483.647

■ Int

- 64-Bit, signed-int -9223372036854775808 ... 9223372036854775807

Swift: Datentypen:

- Float, Float32
 - Single-Format (1,8,23-Bit, 7 Stellen)
 - Sign Charakteristik Fraction
- Double, Float64
 - Double-Format (1,11,52-Bit, 15-Stellen)
 - Sign Charakteristik Fraction
- Float80
 - Extended-Format (1,15,62-Bit, 17-Stellen)
 - Sign Charakteristik Fraction

- Decimal: leider nicht vorhanden, 0.1 Problematik
- NSNumber: entspricht BigDecimal in Java

- Bool

Swift: String-Datentyp

■ Zuweisung

- `var string1=""`
- `var string2=String()`
- `var string3="abc"`
- `string3+="def"` geht nicht
- `String3.append("def")`
- `string4=string1 + string2;`
- `let stringConst="42 ist die Antwort auf alle Frage"`
- `let dollarSign = "\u{24}"` // \$, Unicode scalar U+0024
- `let blackHeart = "\u{2665}"` // ♥, Unicode scalar U+2665
- `let sparklingHeart = "\u{1F496}"` // 💎, Unicode scalar U+1F496

Swift: String-Datentyp

■ Methoden

- `characters.count` ab 2.0
- `isEmpty`
- `startIndex` 1. Zeichen
- `endIndex` n. Zeichen
- `predecessor`
- `successor`
- `endIndex.predecessor.predecessor` n-2. Zeichen
- `startIndex.successor.successor` 3. Zeichen
- `welcome.insert("!", atIndex: welcome.endIndex)`
- `filename.hasPrefix("\\home\\")`
- `filename.hasSuffix(".EXE")`

Swift: String-Datentyp

■ Abfragen

- String sind keine Referenzen (== vs. Equals)
- `if string1.isEmpty { ... }`
- `for character in "Dog!🐶".characters {
 print(character)
}`

- `let drei = 3`
- `let msg = "\(drei) times 2.5 is \((Double(drei) * 2.5))"`
- `// msg is "3 times 2.5 is 7.5"`

Swift: Konvertierung

```
let s="123"
```

```
var i = Int32(s)
```

```
if i != nil {  
}
```

```
let s="123.34"
```

```
var d = Double(s)
```

```
if d != nil {  
}
```

Swift: Operatoren:

- !=
- !== Identität
- %
- %=
- &
- &&
- &*
- &+
- &-
- &=
- *
- *=
- +
- +=
- -
- -=
- ...
- ..<
- /
- /=
- <
- <<
- <<=
- <=
- ==
- === Identität, nur Objekte
- >
- >=
- >>
- >>=
- ??
- ^
- ^=
- |
- |=
- ||
- ~=
- ~>
- Coalescing Op.
- a ?? b
- wenn a<>nil a!
- sonst b

Swift: is-Operator:

- Der Operator **is** testet, ob eine Variable, Instanz einen Typ entspricht:

```
if item is UInt32 {  
    ...  
}  
if item is MyClass {  
    ...  
}
```


Swift: as-Operator:

Der Operator **as** wandelt eine Variable in einen anderen Typ um. Er hat drei Varianten:

- as
 - Hier wird keine Überprüfung durchgeführt. Es wird nur durchgeführt, wenn der Compiler vorher erkennt, dass das Casting gefahrlos möglich ist. Es ist nur sinnvoll beim UpCasting:

Float nach Double

Int8 nach Int32

Swift: as-Operator:

Der Operator **as** wandelt eine Variable in einen anderen Typ um. Er hat drei Varianten:

- **as?**
 - Hier wird eine Überprüfung durchgeführt. Bei Fehlern kann der Wert nil zurückgegeben. Man kann diesen Ausdruck mit einer if-Anweisung koppeln:

```
if let myInt = item as? UInt32 {  
    print(myInt)  
}  
else {  
    // myInt ist nil, Error  
}
```

Swift: as-Operator:

Er hat drei Varianten:

- as!
 - Hier wird keine Überprüfung durchgeführt. Das Casting wird durchgeführt. Bei Fehlern kann es zu Abstürzen kommen. Man kann den Fehlerfall vermeiden, wenn man vorher den Typ testet.

```
let item:Any = 123 // Typ Int
if item is UInt32 { // ist wahr!
    myInt = item as! UInt32
    print(myInt)
}
else {
    // myInt ist nil, Error
}
```

Swift: if-Anweisungen: Immer mit den Klammern { }

```
var a:Int = 5
var b:Int = 7
if a > b {
    print("a ist größer als b");
}
else {
    print("a ist kleiner als b");
}
```

```
if a > b {
    print("a ist größer als b");
}
```

Swift: case-Anweisungen

```
var a:Int = 4
```

```
switch (a) {  
    case: 1,  
    case: 2:  
        print("1 oder 2");  
    case: 4:  
        print("4");  
    default: // muss immer eintragen werden  
        print("else");  
}
```

Swift: case-Anweisungen

```
var a:Int = 4
```

```
switch (a) {
```

```
    case: 1,2:
```

```
        print("1 oder 2");
```

```
    case: 4:
```

```
        print("4");
```

```
    default:                // muss immer eintragen werden
```

```
        print("else");
```

```
}
```

Swift: case-Anweisungen

```
var a:String = "a"
```

```
switch (a) {  
    case: "a":  
        print("a");  
    case: "b":  
        print("b");  
    default:  
        print("else");  
}
```

Swift: Schleifen

```
var d:Int = 0
```

```
var e:Int = 10
```

```
while d <= e {
```

```
    d++
```

```
    print(String(d))
```

```
}
```

```
repeat {
```

```
    d++
```

```
    print(String(d))
```

```
} while d < e
```


Swift: For-Schleifen: nur mit Range-Objekten

```
for i in 1...10 {  
    print(String(i))  
}
```

```
n=0 // Absturz
```

```
for i in 1...n {  
    print(String(i))  
}
```

```
var n=10
```

```
for i in 0..<n {  
    print(i)  
    print("i: ", i)  
    print("i: \(i)")  
}
```

```
for i in (1..<10).reversed() {  
    print(i)  
    print("i: ", i)  
    print("i: \(i)")  
}
```

```
// i ist nicht bekannt
```

```
for _ in 1...10 {  
    print("Hallo Welt")  
}
```

Swift: for-each-Schleifen

```
var arrayInt:[Int] = [1,2,3,4,5,6,7,8,9,10]
for (i in arrayInt) {
    print("i: \(i)")
}
```

```
let dict = ["one":"eins", "two":"zwei", "three":"drei"]
for engl, germ in dict {
    print("\(engl) und \(germ)")
}
```

Swift: Arrays

```
var person1 = "Paul"  
var person2 = "Susanne"  
  
var array: [String] = [person1, person2]  
Array.append("Michel")  
  
// Ausgabe  
for personname in array {  
    print("Person: \(personname)")  
}
```

Swift: Arrays

```
var array: [String] = []  
var array: [String]()  
Array.append("Michel")  
Array.append("Andrea")  
for personname in array {  
    print("Person: \(personname)")  
}  
for i in 0...len(array) -1 {  
    print("Person: \(array[i])")  
}  
for i in 0..len(array) {  
    print("Person: \(array[i])")  
}
```

ev. FEHLER

Swift: mehrdimensionale Arrays

```
var array = [ [1,2,3] [4,5,6,7,8] , [9,10] ]
```

array.count liefert 3

```
for (row in 0..    for (col in 0..        print("row: \(row) col \(col) : \(array[row][col]) ")  
    }  
}
```

```
var array2D = [ [Int] ]()
```

```
var array3D = [ [ [Int] ] ]()
```

Swift: Enumeration:

- Enumeration sind besser als mehrere Konstante:

- `let Int fb_ai = 1`
- `let Int fb_vw = 2`
- `let Int fb_w = 3`
- `printFB(Int fn) { }`

- Enumeration

- `enum FB {` // Deklaration
 - `case AI`
 - `case VW`
 - `case W`
- `}`
- `var einFB = FB.AI`

Swift: Enumeration mit Zahlenwerten

■ Enumeration

- enum FB { // Deklaration
 - case AI(Int) // Wert ist nicht vorbelegt
 - case VW(Int)
 - case W(Int)
- }
- enum FBSTRG { // Deklaration
 - case AI(String) // Wert ist nicht vorbelegt
 - case VW(String)
 - case W(String)
- }

- var einFB = FB.AI(2)
- var einFB = FB.AI(12)
- var einFB = FB.VW(4)

Swift: eingebaute Funktionen:

abs	
max	
min	
numericCast	
round	<pre>(4.4).rounded() // == 4.0 (4.5).rounded() // == 5.0 (4.0).rounded(.up) // == 4.0 (4.9).rounded(.down) // == 4.0 (4.0).rounded(.down) // == 4.0</pre>
sizeof	Größe der Variable
arc4random_uniform(n) + 1	<pre>var zahl = Int32(arc4random_uniform(10000) + 1)</pre>

Swift: implizit eingebaute Funktionen bei Datentypen:

- `first`
 - Erste Element eines Arrays oder Strings (characters)
- `last`
 - Letzte Element eines Arrays oder Strings(characters)
- `prefix`
 - Erste Elemente eines Arrays oder Strings (characters)
 - `let dummy=x.prefix(2)`
 - `s. (characters).prefix(3)`
- `suffix`
 - Letzte Elemente eines Arrays oder Strings
- `dropFirst`
 - Neues „Array“ ohne des ersten Wertes
- `dropLast`
 - Neues „Array“ ohne dem letzten Wert

Swift: implizit eingebaute Funktionen bei Datentypen:

- `startsWith`
 - Test, ob die ersten Elemente eines Arrays oder Strings einen Wert haben
 - `myField.startsWith([12])`
 - `myField.startsWith([12,45])`

- `startsWith` geht nicht bei String
 - `hasPrefix`
 - `hasSuffix`

- `contains`
 - Test, ob ein Element vorhanden ist (Arrays oder Strings)

Swift: implizit eingebaute Funktionen bei Datentypen:

■ indexOf

- Suche ein Element in einem eines Arrays oder String
- Von 0 bis n-1
- Fehler: Rückgabewert: nil

■ split

- Zerlegt eine Arrays oder String (CSV)
- `let data = [1, 2, 0, 5, 6, 4, 0, 0, 3, 0, 2]`
- `let splitted = data.split(0)`
- 1, 2 5, 6, 4, 3 2

Swift: implizit eingebaute Funktionen bei Datentypen:

■ joinWithSeparator

- Fasst Arrays zusammen
- Erzeugt CVS Daten
- `let data = [[1, 2], [5, 6, 4], [3], [2]] // Arrays in Array`
- `let join = data.joinWithSeparator([0])`

■ filter

- Filtert Arrays und Listen
- `let data = [1, 2, 0, 5, 6, 4, 0, 0, 3, 0, 2]`
- `let result = data.filter { checkItem($0) }`
- `let result = data.filter { $0 % 2 == 0 }`

- `func checkItem(x:Int) -> Bool {`
 - `return (x%2)==0`
- `}`

Swift: implizit eingebaute Funktionen bei Datentypen:

■ map:

- Map erhält eine Referenz einer Funktion, diese wird für alle Elemente aufgerufen
- Beispiel:

```
let data = [1, 2, 0, 5, 6, 4, 0, 0, 3, 0, 2]
data.map(inc)
func inc(inout x:Int) {
    x++
}
```

■ flatMap

- Übergabe einer Funktion
- Benutzt beim Verarbeiten von Arrays

Swift: implizit eingebaute Funktionen bei Datentypen:

Function in Swift, die mit **d** enden, erzeugen eine neue Instanz:

- `myArray.sort()`
 - `myArray2=myArray.sorted()`
-
- `reduce`
 - Übergabe einer Funktion
 - Aufruf der Funktion mit zwei Indizes
 - Bildverarbeitung
 - Summenbildung
 - `myFeld.reduce(0) { (total, number) in total + number } ()`
 - `sort(ed)`
 - Sortieren
 - `reverse(d)`
 - Umkehren der Reihenfolge

Swift: Type-Annotation

- „kleine Klassendefinition“

- `let someTuple: (Double, Double) = (3.14159, 2.71828)`
- **typealias** `Point = (Int, Int)`
- `let origin: Point = (0, 0)`
- benutzt in Funktionen als Rückgabewert oder „kleine“ Klasse

Swift: void Funktionen

```
func print1() {  
    print("hallo")  
}  
print1()
```

```
func printInt() {  
    var a:Int=3;  
    print("a hat den Wert: /(a)")  
}  
printInt()
```

```
func print2(a:Int32) {  
    print("a", a)  
}  
print2(a:33)
```

```
func print3(a:Int32, b:Int32) {  
    print("a", a, "b", b)  
}  
print3(a:33, b:11)
```


Swift: void Funktionen

```
func Inch2Cm(inch:Double) -> Double{  
    return inch*2.54  
}  
let cm=1.23  
let inch=Inch2Cm(inch:cm)
```

Swift: void Funktionen

```
func add1(aa a:Int32, bb b:Int32)-> Int32{  
    return a+b;  
}
```

let c = add1(**aa:12, bb:33**) // alle Parameter haben einen Namen

```
func add2(a:Int32, b:Int32)-> Int32{  
    return a+b;  
}
```

let c = add2(a:12, b:33) // alle Parameter haben einen Namen

```
func add3(_ a:Int32, b:Int32)-> Int32{  
    return a+b;  
}
```

let c = add3(12, **b:33**) // **1. Parameter ohne Namen!**

Swift: Funktionen (C# hat ref und out)

```
func add(int a, int b, c: inout Int) -> {  
    c = a+b  
}
```

// Aufruf

```
var erg=41  
add(a:22, b:33, c:&erg)           // C läßt Grüßen  
print(„Ergebnis: \ (erg) ");
```

Swift: Funktionen (Tupel, unbenannt)

```
func convertcm2InchYard(cm:Double) ->
    (Double, Double) {
    var inch_out = cm/2.54;
    var yard_out = cm*0.01/1.0936;
    return (inch_out, yard_out);
}
```

// Rückgabewert ist ein Tupel

```
var laengen = convertcm2InchYard(120.03);
print("inch: \(laengen.0) yard: \(laengen.1) ");
```

Swift: Funktionen (Tupel, benannt)

```
func convertcm2InchYard(cm:Double) ->
    (inch_out:Double, yard_out:Double) {
    var inch_out = cm/2.54;
    var yard_out = cm*0.01/1.0936;
    return (inch_out, yard_out);
}
```

// Rückgabewert ist ein Tupel

```
var laengen = convertcm2InchYard(120.03);
print("inch: \(laengen.inch_out)  yard: \(laengen.yard_out) ");
```

Swift: Funktionen mit Rückgabewert als Tupel

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..
```

Swift: Funktionen mit optionalen Parametern

```
func addABC(a:Int, b:Int, c:Int=0) -> Int {  
    return a+b+c;  
}
```

```
var summe = addABC(1,2,3);  
print("summe: \(summe) ");
```

```
var summe = addABC(1,2);  
print("summe: \(summe) ");
```

Swift: geschachtelte Funktionen

```
func getModDiv(a:Int, b:Int, c:Int=0) -> (mod: Int, div: Int) {  
    func mod(a:Int, b:Int, c:Int=0) -> Int {  
        return a%b  
    }  
    func div(a:Int, b:Int, c:Int=0) -> Int {  
        return a/b  
    }  
    return ( mod(a,b) , div(a,b) )  
}
```

```
var moddiv = getModDiv(11,3);  
print("mod: \(moddiv.mod)  div: \(moddiv.div) ");
```


Swift: Funktionen mit benannten Parametern

```
func add(a:Int, #b:Int, #c:Int) -> Int {  
    return a + b +c;  
}
```

// Aufruf

```
var x=41
```

```
var y=41
```

```
var summe = add(a:x, c:y, b:11)
```

```
print("summe: \(summe) ");
```

Swift: Funktionen mit variablen Parametern: Variadics

```
func add(numbers: Int...) -> Int {  
    var summe=0  
    for (x in numbers) {  
        summe+=x  
    }  
    return summe;  
}
```

// Aufrufe

```
var summe = add(1,2,3);    print("summe: \(summe) ");  
var summe = add(1,2,3,5); print("summe: \(summe) ");
```

Swift: Dictionaries (HashTable)

- Daten werden paarweise eingetragen
- Dient zur schnellen Suche
- Schlüssel ist der „Index“
- Protokoll Hashable

```
var dict = [ String: Int ]  
var dict = [ "rot": "red", "blau": "blue" ]  
dict[ "grün" ] = "green",  
var farbe = dict[ "red" ]  
print( "grün: \(dict[ "grün" ]) ")
```

```
dict[ "grün" ] = nil           löscht grün  
dict.removeAll()
```

Swift: Set (Mengen)

- Daten werden eingetragen
- Es gibt keine doppelten Einträge
- Protokoll Hashable

```
var set = Set<Int>()
set.insert ( 10)
set.insert ( 2)
set.insert ( 10)
set.insert ( 4)
var array = Array(set).sort(<)
```

Swift: try catch, gibt es nur mit Funktionen

- Fehlerabfangen
- `func canThrowErrors() throws -> String`

```
do {  
    try  
    ...  
} catch pattern1 {  
    ...  
}  
} catch pattern2 {  
    ...  
}
```

Swift: Klassen:

```
class MyClass: OptSuperclass, OptProtokoll1, OptProtokoll2 {  
    var myProperty:String  
    var myOptionalProperty:String?  
    init() {  
        myProperty = "abc"  
    }  
    func doIt() -> Int {  
        return 0  
    }  
    func doIt(a:Int) -> Int {  
        return a+1  
    }  
    var MyProperty:String {  
        get { return myProperty }  
        set { myProperty = newValue }  
    }  
}
```

Swift: Klassen

- class
- Instanz: **ohne new**
- init Konstruktor
- deinit Dekonstruktor, sonst in C++ und C#
- Properties setter / getter, à la C#
- Type Methoden statische Methoden
- Protokoll Interface
- self this
- nil null
- Vererbung ja, aber nur einfache Vererbung

Swift: Klassendefinition

```
class Temperatur{  
    var tKelvin:Double;  
  
    init() {  
        tKelvin = 273.15  
    }  
    init(temp t:Double) {  
        tKelvin=t;  
    }  
}
```

```
    init(tKelvin:Double) {  
        self.tKelvin = tKelvin  
    }  
  
    deinit {  
        tKelvin=0  
    }  
}
```

- **temp**: äußerer Name

Swift: Methoden

```
class Temperatur{  
    var tKelvin:Double;  
    func getT() -> Double{  
        return tKelvin  
    }  
  
    func addBy(intervalT t: Double) {  
        tKelvin += t  
    }  
  
    func reset() {  
        tKelvin = 0  
    }  
}
```

```
let t1 = Temperatur()  
t1.addBy(5)  
  
// benannte Parameter  
t1.addBy(intervalT:5)
```

Swift: Computer Properties (setter, getter)

```
class Temperatur{  
    var tKelvin:Double;  
    init(t:Double) {  
        tKelvin = t  
    }  
    var celcius : Double {  
        get { return t + Nullpoint }  
        set { t = newValue }  
    }  
}
```

```
let t1 = Temperatur(t:300)
```

```
let cel1 = t1.celsius
```

```
t1.celsius = 122
```

```
let cel2 = t1.celsius
```

Swift: Computer Properties (setter, getter)

```
class Temperatur{  
    var tKelvin:Double;  
  
    init(t:Double) {  
        tKelvin = t  
    }  
  
    var celcius : Double {  
        get { return t + Nullpoint }  
        set(value) {  
            t = value  
        }  
    }  
}
```

Swift: Computer Properties (getter)

```
class Temperatur{  
    var tKelvin:Double;  
    init(t:Double) {  
        tKelvin = t  
    }  
    var celcius : Double {  
        get { return t + Nullpoint }  
    }  
}
```

```
let t1 = Temperatur(300)
```

```
let cel1 = t1.celsius
```

```
t1.celsius = 122
```

FEHLER

```
let cel2 = t1.celsius
```

Swift: Klassenvariable und Klassenmethoden

```
class Temperatur{
```

```
    static let Nullpoint = 273.15
```

```
    var tKelvin:Double;
```

```
    static func convertC2K(t:Double) -> Double {
```

```
        return t - Nullpoint
```

```
    }
```

```
}
```

```
let tK = Temperatur.convertC2K(0)
```

Swift: Vererbung und Polymorphismus

```
class Point{
```

```
    var x:Double = 0.0;
```

```
    func draw() { ... }
```

```
}
```

```
class Line : Point{
```

```
    var y:Double = 0.0;
```

```
    override func draw() { ... }
```

```
    func save(filename:String) { ... }
```

```
}
```

Swift: Vererbung

```
class Point{
```

```
    var x:Double = 0.0;
```

```
    init(x:Double) {
```

```
        self.x = x
```

```
    }
```

```
}
```

```
class Line : Point{
```

```
    var y:Double = 0.0;
```

```
    init(x:Double, y:Double) {
```

```
        self.y = y
```

```
        super.init(x)
```

```
    }
```

```
}
```

super.init(x): Reihenfolge ist nicht beliebig, anders als in Java

Swift: Vererbung

```
final class Point{
```

```
    var x:Double = 0.0;
```

```
    init(x:Double) {
```

```
        self.x = x
```

```
    }
```

```
}
```

```
class Line : Point{
```

```
    var y:Double = 0.0;
```

```
    init(x:Double, y:Double) {
```

```
        super.init(x)
```

```
        self.y = y
```

```
    }
```

```
}
```

Fehlerhafte Syntax


```
class Auto:CustomStringConvertible{  
    private var name:String=""  
    init(name:String){  
        self.name=name  
    }  
  
    // toString  
    var description :String {  
        return "Auto: \(name) \(laenge)"  
    }  
} // Auto  
  
var auto:Auto = Auto(name:"Bentley Veyron")  
print(auto.description())
```

Swift: Protokolle, Protocol (Interface)

```
protocol ISave {  
    func save2DB(table:String)  
    func save2TXT(filename:String)  
  
    // ist über dem Klassennamen erreichbar  
    static func convert2Int(x:Double) -> Int  
  
}
```

```
class Student : ISave {  
  
}
```

Swift: Swift-Protokolle:

- **AbsoluteValuable**
- AnyCollectionType
- **AnyObject**
- ArrayLiteralConvertible
- BidirectionalIndexType
- BitwiseOperationsType
- BooleanLiteralConvertible
- **BooleanType**
- CVarArgType
- **CollectionType**
- **Comparable**
- CustomDebugStringConvertible
- CustomLeafReflectable

Swift: Swift-Protokolle:

- CustomPlaygroundQuickLookable
- CustomReflectable
- CustomStringConvertible
- DictionaryLiteralConvertible
- **Equatable**
- ErrorType
- ExtendedGraphemeClusterLiteralConvertible
- FloatLiteralConvertible
- FloatingPointType
- ForwardIndexType
- GeneratorType
- **Hashable**
- **Indexable**

Swift: Swift-Protokolle:

- **IntegerArithmeticType**
- IntegerLiteralConvertible
- **IntegerType**
- **IntervalType**
- LazyCollectionType
- LazySequenceType
- MirrorPathType
- MutableCollectionType
- MutableIndexable
- MutableSliceable
- NilLiteralConvertible
- OptionSetType
- OutputStreamType
- RandomAccessIndexType

Swift: Swift-Protokolle:

- RangeReplaceableCollectionType
- RawRepresentable
- ReverseIndexType
- SequenceType
- SetAlgebraType
- SignedIntegerType
- SignedNumberType
- Streamable
- Strideable
- StringInterpolationConvertible
- StringLiteralConvertible
- UnicodeCodecType
- UnicodeScalarLiteralConvertible
- UnsignedIntegerType

Swift: Klasse mit Hashwert

```
class Person : Hashable{
```

```
    var vname:String = ""
```

```
    var nname:String = ""
```

```
    init(name:String) {
```

```
        self.name=name
```

```
    }
```



```
}
```

```
    var hashable: Int {
```

```
        var hash1 = vname.hashValue
```

```
        var hash2 = nname.hashValue
```

```
        return hash1+ hash2
```

```
    }
```

Swift: Klasse mit Comparable: statische Operator-Overloading

```
class Person : Comparable{  
    var vname:String = ""  
    init(name:String) {  
        self.name=name  
    }  
  
    func < (p1:Person, p2:Person) -> Bool  
        return p1.name < p2.name  
    }  
}
```