Fachbereich Automatisierung und Informatik	▲ Hochschule Harz Wernigerode

Vorlesung „iOS mit Swift“

Skript zu Swift

Version 29.08.2021

Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
Friedrichstraße 57 - 59
38855 Wernigerode

Raum: **2.202**
Tel.: **03943/659-338**
Fax: **03943/659-399**
Email: **mwilhlem@hs-harz.de**
Web: **http://www.miwilhelm.de**

Inhaltverzeichnis

Abbildungen

1 Tastatur

1.1 Apple-Sondertasten

	Commandkey, Apple-Taste
	Controlkey, Steuerungstaste
	Options or Alt Taste
	fn (Funktion-Taste)

1.2 Cmd, Befehlstaste-, Apple-Tasten

Taste	**Beschreibung**
Cmd C	Kopieren
Cmd V	Einfügen
Cmd X	Ausschneiden
Cmd Z	Undo / Redo
Cmd A	Alles markieren
Cmd F	Suchen
Cmd G	Weitersuchen
Cmd N	Neue Datei / Projekt
Cmd O	Öffnen Datei
Cmd P	Drucken
Cmd W	Schließen
Cmd Tab	Wechsel zwischen Programmen
Cmd+←	Home
Cmd+→	Ende

1.3 Fn-Tasten

Taste	Beschreibung
fn Backspace	Delete Taste "Entf"

1.4 Sonderzeichen

Alt	5	[
Alt	6	]
Alt	8	{
Alt	9	}
Alt	L	@
Alt	N	~
Alt+Shift+7		\
Alt+7		|

1.5 „Taskmanager“

Aufruf: Cmd+Alt+Esc

2 Einstieg in Swift

2.1 Swift 3.0

https://swift.org/blog/swift-3-0-preview-1-released/

2.2 Variablen-Deklaration

```
let max=100                          "Konstanter Ausdruck

var implicitInteger = 70             varianter Datentyp
var implicitDouble = 70.0
var explicitDouble: Double = 70      Double Datentyp
```

2.3 Datentypen

- **Any** **jeder allgemeine Datentyp**
- UInt8 8-Bit, unsigned-int 0 → +255 **UInt8.min UInt8.max**
- UInt16 16-Bit, unsigned-int 0 → +65535
- UInt32 32-Bit, unsigned-int 0 → +4294967295
- UInt64 64-Bit, signed-int 0 → + 18446744073709551614
- UInt 32-Bit, unsigned-int 0 → +4294967295
  - wenn ein 32-Bit System
- UInt 64-Bit, signed-int 0 → + 18446744073709551614
  - wenn ein 64-Bit System
- 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 32-Bit, signed-int -2.147.483.648 -> +2.147.483.647
  - wenn ein 32-Bit System
- Int 64-Bit, signed-int -9223372036854775808 ... 9223372036854775807
  - wenn ein 64-Bit System
- Float Single-Format (1,8,23-Bit, 7 Stellen) Sign Charakteristik Fraction
- Double Double-Format (1,11,52-Bit, 15-Stellen) Sign Charakteristik Fraction
- Float32 Single-Format (1,8,23-Bit, 7 Stellen) Sign Charakteristik Fraction
- Float64 Double-Format (1,11,52-Bit, 15-Stellen) Sign Charakteristik Fraction
- **Float80** **Extended-Format (1,15,63-Bit, 15-Stellen)** **Sign Charakteristik Fraction**
- NSNumber siehe 2.9, Seite 13
- CGFloat bei 32-Bit eine float, sonst eine double-Zahl
- Bool
- String
- Struct,
- Tupel (1,2)

2.3.1 Casting

- Int(…) ab Version 7.x
- Float(…) ab Version 7.x
- Double(…) ab Version 7.x
- String(12.45) ab Version 7.x

2.4 Boolean toggling

```
extension Bool {
  mutating func toggle() {
    self = !self
  }
}
```

However, the end result makes for much more natural Swift code:

```
var loggedIn = false
loggedIn.toggle()
```

2.5 Modulo

```
let rowNumber = 4

if rowNumber % 2==0) {
   print("Even")
} else {
   print("Odd")
}
```

```
let rowNumber = 4

if rowNumber.isMultiple(of: 2) {
   print("Even")
} else {
   print("Odd")
}
```

2.6 count

```
let scores = [100, 80, 85]
let passCount = scores.count { $0 >= 85 }
```

oder

let pythons = ["Eric Idle", "Graham Chapman", "John Cleese", "Michael Palin", "Terry Gilliam", "Terry Jones"]
let terryCount = pythons.count { $0.hasPrefix("Terry") }

2.7 Konvertierung

2.7.1 String nach einem Datentypen

Version ab 7.x:

String → Int

```
let input:String=uiInputInt.text!
var zahl1=Int(input)
if zahl1  = =  nil {
_uiLabel.text="error"
}
else {
_let i:Int=zahl1!
_uiLabel.text="zahl1: "+String(i)
}
```

String → Double

```
let input:String=uiInputInt.text!
var zahl1=Double(input)
if zahl1==nil {
_uiLabel.text="error"
}
else {
_let value:Int=zahl1!
_uiLabel.text="zahl1: "+String(".f",value)
_uiLabel.text="zahl1: "+String(".4f",value)
}
```

2.7.2 Datentypen nach einem String

Die Konvertierung funktioniert mit der Methode String und einem Format-Parameter:
String str = String(format:"…. ",Datentyp)

Konvertierung eines Int in einem String:

- %d: 32 Bit sign-integer number
- %u: 32 Bit unsigned number
- %x: 32 Bit unsigned number, with hexadecimal-code

Konvertierung eines Floats, Doubles in einem String:

- %f: float number
- %10.3f: float number
  - 7 Stellen vor dem Komma
  - 3 Stellen nach dem Komma
- %.3f: float number
  - beliebige Stellen vor dem Komma
  - 3 Stellen nach dem Komma
- %e: double number
  - Die Zahl wird in der Exponential-Darstellung konvertiert: 1.34E+3
- Format eines Strings:
  - %@: String-Format

2.7.3 String-Methoden

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

String sind keine Referenzen (= = vs. Equals)

```
var s1="abcdefgh"
var s2=s1.characters.count                Anzahl der Zeichen

if s1.isEmpty {
    print("s1 is empty")
}
else {
  print("s1 is not empty")
}

s1[s1.startIndex]
s1[s1.startIndex.successor()]
```

```
s1+="def"                          Addition

var s3="abc"+"def"                   Addition
let ch: Character = "!"                  Addition
s3.append(ch)  // nur Character
s3.appendContentsOf("123456")          Addition

s3.insertContentsOf("xyz".characters, at: s3.startIndex)
  Insert

let range = s3.endIndex.advancedBy(-3)..<s3.endIndex.predecessor()
s3.removeRange(range)
```

2.7.4 long String Definition

```
assert( xml == """
    <?xml version="1.0"?>
    <catalog>
        <book id="bk101" empty="">
            <author>\(author)</author>
            <title>XML Developer's Guide</title>
            <genre>Computer</genre>
            <price>44.95</price>
            <publish_date>2000-10-01</publish_date>
            <description>An in-depth look at creating applications
with XML.</description>
        </book>
    </catalog>
    """ )
```

```
let str1 = """↲
  line one \↲
  line two \      ↲
  line three↲
  """↲

let str2 = "line one \↲
line two \      ↲
line three"↲
```

2.7.5 split

```
let s = "one two three"
s.split(separator: " ")
```

2.7.6 Raw-String

let rain = #"The "rain" in "Spain" falls mainly on the Spaniards."#

```
let answer = 42
let dontpanic = #"The answer to life, the universe, and everything
is \#(answer)."#

let regex1 = "\\\\[A-Z]+[A-Za-z]+\\.[a-z]+"
Thanks to raw strings we can write the same thing with half the
number of backslashes:

let regex2 = #"\\[A-Z]+[A-Za-z]+\.[a-z]+"#
```

2.8 ImplicitlyUnwrappedOptional

Variablen haben normalerweise einen Wert. Zum Beispiel Null oder Eins. In Java werden alle Variablen mit Default-Werten initialisiert. In Swift gibt es ähnlich wie in Datenbanken auch einen **Nullwert**. Diese Technik wird bei der Abfrage von Textinhalten häufig benötigt. Trotzdem haben die Variablen **weiterhin** den „Nullwert“.
Beispiel:
let t: String? = textfield.text! // t hat einen Wert oder keinen Wert, also Null

Der Datentyp String? definiert eine Variable, die einen String oder einen Nullwert besitzen kann.
Der Datentyp Int? definiert eine Variable, die eine Zahl oder einen Nullwert besitzen kann.
Die Anweisung „!“ prüft die Variable und wandelt diese in den Datentyp um.

Beispiele:

let x: Int! = 5	x ist deklariert als ein IUO
let y = x	y ist deklariert als ein Int?
let z = x + 0	z ist deklariert als ein Int, es gibt kein Operator +? (Zwang)

2.9 NS-Datentypen

Diese Datentypen sind in ObjectiveC implementiert. Sie haben aber einige Eigenschaften, die Swift-Datentypen noch nicht haben.

NSNumber:

- NSObject NSValue NSNumber
- Entspricht vollständig den Swift-Datentypen

NSDecimalNumber:

- Sie entsprechen dem Big-Decimal in Java.
- Wichtig für Nachkommastellen. Problem 0,1

- Es gibt aber nur einfache mathematische Funktionen.

Beispiel:

```
let nulldrei:Float=0.3
let nullneun:Float=0.9
let erg = nulldrei+nulldrei+nulldrei
if erg==nullneun {
    print("equal")
}
else {
    print("not equal:",(erg-nullneun))
}
```

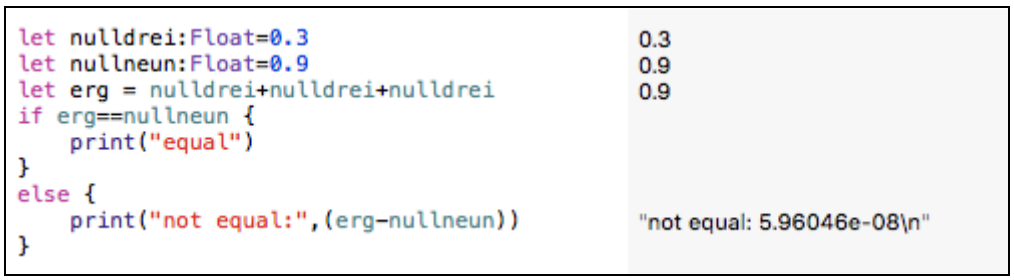

Abbildung 1 3*0.3 vs. 0.9 Problematik

```
let d3 = NSDecimalNumber(string:"0.3")
let d9 = NSDecimalNumber(string:"0.9")    // 0.91
var erg9:NSDecimalNumber=d3.decimalNumberByAdding(0.3)
erg9=erg9.decimalNumberByAdding(0.3)
if erg9==d9 {
    print("equal")
}
else {
    print("not equal:",(d9.decimalNumberBySubtracting(erg9)))
}
```

Methoden:

- decimalNumberByAdding:withBehavior:
- decimalNumberBySubtracting:
- decimalNumberByMultiplyingBy:
- decimalNumberByDividingBy:
- decimalNumberByRaisingToPower:
- decimalNumberByMultiplyingByPowerOf10:
- decimalNumberByAdding:withBehavior:
- decimalNumberBySubtracting:withBehavior:
- decimalNumberByMultiplyingBy:withBehavior:
- decimalNumberByDividingBy:withBehavior:
- decimalNumberByRaisingToPower:withBehavior:
- decimalNumberByMultiplyingByPowerOf10:withBehavior:

NSDate

```
var datum = NSDate()
let dtformat = NSDateFormatter()
dtformat.dateFormat = "dd.MM.yyyy '-' hh:mm"
let ausgabe = dtformat.stringFromDate(datum)
print(ausgabe)
```

NSRange

```
let str:NSString = "abc  defghij xyz"
var range:NSRange = str.rangeOfString("defghij")
print(range)       // Ausgabe Position und Laenge (5,7)  ab 0
```

NSTimeInterval

```
var intv:NSTimeInterval   // = NSTimeIntervall()
var t1 = NSDate.timeIntervalSinceReferenceDate()  //480863416.
var t2 = NSDate.timeIntervalSinceReferenceDate()
intv = t2-t1
print(intv)
```

NSArray

- speichert jeden Datentyp

```
var feld: NSArray = NSArray(array:[2,6,12,77])
```

2.10 Kommentare

Allgemeine Kommentare

```
//                                      nur eine Zeile

/*                                      Bereich Anfang
*/                                      Bereich Ende
```

Formatierte Kommentare:

```
*italic*
**bold**
`listingfont`

#head1#
##head2##
###head3###

* Listing1
* Listing2
* Listing3
```

2.11 Enumeration

```
enum FB {    // Deklaration
  case  AI
  case  VW
  case  W
}

var einFB = FB.AI


enum FB {    // Deklaration
  case  AI(int)
  case  VW(int)
  case  W(int)
}

var einFB = FB.AI(2)          // Zuweisung einer Zahl
var einFB = FB.AI(12)
var einFB = FB.VW(4)
```

2.11.1 unknown, switch

```
enum PasswordError: Error {
   case short
   case obvious
   case simple
}

func showOld(error: PasswordError) {
   switch error {
   case .short:
      print("Your password was too short.")
   case .obvious:
      print("Your password was too obvious.")
   default:
      print("Your password was too simple.")
   }
}


func showNew(error: PasswordError) {
   switch error {
   case .short:
      print("Your password was too short.")
   case .obvious:
      print("Your password was too obvious.")
   @unknown default:
      print("Your password wasn't suitable.")
```

```
    }
}
```

Einfügen eines neuen Falles: “old” würde

2.12 Operationen

! log. Not	+	>
!=	+=-	>=
!== Identität	-=	>> Shift rechts
%	... for range 1…10	>>=
%=	..< for range 1..<10	? nicht definiert
&	/	^
&&	/=	^=
&*	<	| Bitweise Oder
&+	<< Shift links	|=
&-	<<=	|| Logisches Oder
&=	<=	~= Bitweise Negierung
*	= =	~>
*=	= = = Identität, nur bei Objekten	

2.12.1 is-Operator

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

- `if item is UInt32 {…}`
- `if item is MyClass {…}`

2.12.2 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
- as?
  - Hier wird eine Überprüfung durchgeführt. Bei Fehlern kann wird 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
}
```

- 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        Fehler
print(myInt)                       Fehler
}
else {
  // myInt ist nil, Error
}
```

2.13 If-Anweisungen

Immer mit Klammern { und }

```
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")
}
if a > b  &&  b>0 {
  print("a ist größer als b und a ist größer Null")
}
```

2.14 Switch/Case

```
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")
}
```

Oder

```
switch (a) {
  case: 1,2:
…
}
```

2. Beispiel:

```
var  a:Int = 4

switch (a) {
  case: 1...10:
    print("1  bis 10")
  case: 11...20:
    print("11 bis 20")
  default:                  // muss immer eintragen werden
    print("else")
}
```

2.14.1 Range-Bereich

Für die For-Schleifen wird ein Vergleich mit einem Bereich, Range-Operator, verwendet.
Dies ist auch in Python mit der Funktion „range“ realisiert.

Testen, ob ein Wert in einem Bereich liegt:

- `-2...+2  ~=  8  false`
- `-2...+12  ~=  8  true`
- `+1.7...<+2.0  ~=  1.9  true`
- `+1.7...<+2.0  ~=  2.0  false`
- `"a"..."z"  ~=  "f"  true`
- `"a"..."z"  ~=  "F"  false`

Intern werden die Funktionen “ClosedIntervall” und „HalfOpenIntervall” benutzt.

ClosedIntervall(1.0 , 3.4) `-1.0...+3.4`
HalfOpenIntervall(1.0 , 3.4) `-1.0...<+3.4`

2.15 Schleifen

2.15.1 For-Schleifen

```
for  i  in 1...10 {
  print(String(i))
}
                                  Ausgabe:1,2,3,4,5,6,7,8,9,10
```

```
var n=10
for  i  in 0..<n {
  print(String(i))
}
```
Ausgabe:0,1,2,3,4,5,6,7,8,9

```
for  i  in 1..<10 {
  print(i)
  print("i: ", i)
  print("i: \(i)")
}
```
Ausgabe:1,2,3,4,5,6,7,8,9

```
for  i  in (1..<10).reversed() {
  print(i)
  print("i: ", i)
  print("i: \(i)")
}
```
Ausgabe:9,8,7,6,5,4,3,2,1

```
for  _  in 1...10 {
  print("Hallo Welt")   i ist nicht bekannt
}
```
Ausgabe:10x Hallo Welt

Foreach-Schleife

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

// **Tupel**

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

2.16 Implizite Schleife

```
let feld = [1, 3, 5, 7, 22]
var sum = 0
feld.forEach() { sum += $0 }
print(sum)
```

2.17 Sequence

```
for x in sequence(first:0.1, next:{ $0*2 }).prefix(while:{$0< 4 })
{
    // 0.1, 0.2, 0.4, 0.8, ...
}
```

Hinweis:

- Bitte die Rundungsfehler beachten

```
for x in sequence(first:0.1, next:{ $0*2 }).prefix(while:{$0< 4 })
{
    // 0.1, 0.2, 0.4, 0.8, ...
}
```

for view in **sequence**(first: someView, next: { $0.superview }) {
  // someView, someView.superview, someView.superview.superview, ...
}

2.17.1.1 Anzeige aller Daten

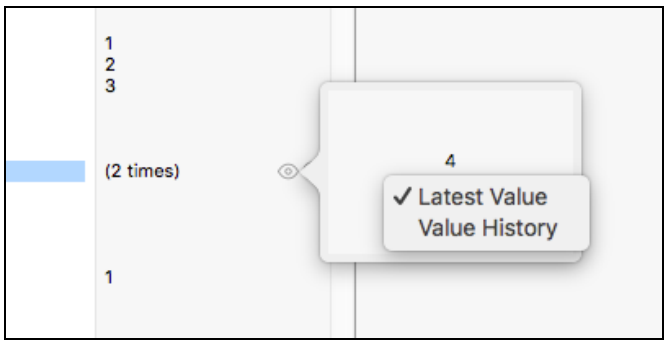

Abbildung 2 Anzeigen aller Daten einer Schleife bzw. Aufrufe

Ablauf:

- Anklicken der Ausgabe
- Anklicken des ersten Symbol
- Rechte Maustaste, Eintrag „Value History“

2.17.2 While-Schleifen

```
var d:Int = 0
var e:Int = 10

while (d<=e) {
```

```
  d+=1
  print(String(d))
}
```

2.17.3 Repeat-Schleifen

```
var d:Int = 0
var e:Int = 10

repeat {
  d+=1
  print("d: ",d)
} while(d<e)
```

2.18 Arrays

Ein Array ist in Swift eine Arraylist. Man kann also zur Laufzeit Datensätze löschen oder hinzufügen.

2.18.1 Erstellen eines Arrays

```
var arrayInt:[Int] = []              // leeres Feld
var arrayInt:[Int]()                 // leeres Feld
var arrayKunde:[Kunde]()             // leeres Feld

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


var person1 = "Paul"
var person2 = "Susanne"
var array:[String] = [person1, person2, "Hanna"]
for  personname  in  array {
  print("Person: \(personname)")
}


var feld2 : [Int] = [1,2,3, 0, 4,5,6,0, 7,8, 0, 9,10]
let feld3 = feld2.split( 0 )


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

```
let feld4 = feld2.filter { ($0 % 2) == 0  }
let feld5 = feld2.filter { checkItem($0)  }

checkItem(4)
checkItem(5)
```

2.18.2 Methoden für Arrays

Methode	**Beispiel**
append(item)	feld.append(1.23)
insert(item, int	feld.insert(0.25, at: 1)
capacity()	Die interne Kapazität des Feldes. Die Kapazität ist immer größer gelich der Anzahl der Elemente: feld.capacity)
count()	Die Anzahl der gespeicherten Elemente feld.count
remove(int)	feld.remove(at: 2)
removeAll	feld.removeAll()
reversed	Das Array wird in der Reihenfolge vertaucht. feld.reversed()
reversed()	Das Feld wird umgedreht und als Ergebnis zuückgegeben. let feld2 = feld.reversed()
contains(item)	Prüft, ob ein Element im Array ist. feld.contains(33)
last	Gibt das letzte Elemente zurück. feld.last
first()	Gibt das erste Elemente zurück. feld.first
removeFirst()	feld.removeFirst()
removeLast()	feld.removeLast()
indexOf	Sucht ein Element in einem Arrays. Returnwert: o Von 0 bis n-1 o Fehler: Rückgabewert: nil

2.18.3 Checking Array

```
let scores = [85, 88, 95, 92]
```

We could decide whether a student passed their course by checking whether all their exam results were 85 or higher:

```
let passed = scores.allSatisfy { $0 >= 85 }
```

2.18.4 Mehrdimensionale Arrays

```
var array2D = [  [Int]  ]()                leere Felder
var array3D = [  [  [Int]  ] ]()


var array = [  [1,2,3] [4,5,6,7,8] , [9,10]  ]
array.count    liefert 3
for (row in 0..<array.count) {
  for (col in 0..<array[row].count) {
    print("row: \(row)   col \(col)  : \(array[row][col])   ")
  }
}
```

2.19 Eingebaute Funktionen

https://developer.apple.com/reference/swift/1693602-swift_standard_library_functions

abs	
max	
min	
numericCast	

round	(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

sizeof	Größe der Variable
arc4random_uniform(n) + 1)	var zahl = Int32(arc4random_uniform(10000) + 1)
native swift random	let randomInt = Int.random(in: 1..<5) let randomBool = Bool.random()

2.19.1 numericCast

2.19.2 readLine

Liest von der Konsole bis zum Zeilenende oder EOF bzw. nil wurde erreicht.
Beispiel:

```
var s = readLine()
var s = readLine(false)
var s = readLine(true)            default: stripNewline
```

2.19.3 sizeof

Anzeige der Speichergröße eines Datentyps oder Variablen.

2.19.4 sizeofValue

Anzeige der Speichergröße eines Datentyps oder Variablen.

2.20 Implizit eingebaute Funktionen bei String oder Arrays

- 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
- startWith
  - Test, ob die ersten Elemente eines Arrays oder Strings einen Wert haben
  - myField.startWith([12])
  - myField.startWith([12,45])
- startWith geht nicht bei String
  - hasPrefix
  - hasSuffix

- contain
  - Test, ob ein Element vorhanden ist(Arrays oder Strings)
- indexOf
  - Sucht ein Element in einem 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
```

- joinWithSeparator
  - Fasst Arrays zusammen
  - Erzeugt CVS Daten

```
// Arrays in Array
let data = [ [1,2], [5,6,4], [3], [2]  ]
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
}
```

- map
  - Übergabe einer Funktion
  - For-Schleife mit Aufruf

```
let data = [1, 2, 0, 5, 6, 4, 0, 0, 3, 0, 2]
data.map(inc)

func inc(inout x:Int)  {
  x+=1
}
```

- flatMap
  - Übergabe einer Funktion
  - Benutzt beim Verarbeiten von Arrays
- reduce
  - Übergabe einer Funktion
  - Aufruf der Funktion mit zwei Indizes
  - Bildverarbeitung
- sort
  - Sortieren
- reverse
  - Umkehren der Reihenfolge

2.21 Reguläre Ausdrücke

Reguläre Ausdrücke werden benutzt, um Eingaben zu verifizieren:

- Datum
- Positive Zahlen
- KFZ-Eingaben
- Email Adressen

2.21.1 Regeln

- . beliebiges Zeichen (Punkt)
- [ABCacZ] Mengengruppe
- [A-Z] Mengengruppe

- [^ABC] negierte Gruppe
- ^ Zeilenanfang, an der Spitze
- $ Zeilenende, Dollar ist am Ende
- \< steht für Wortanfang, links oder rechts steht ein space, tab. CR, Anfang
- \> steht für Wortende
- \b steht Wortanfang oder Wortende
- ( ) Gruppe mit Alternative Oder-Bedingung
- | Oder-Beziehung
  wolf|abc|editor
  (abc|ABC)'

-

2.21.2 Wiederholungs-Operatoren

- 0,1,n
- ? 0,1
- + 1,n
- {n} das vorangegangene Zeichen tritt n-mal auf
- {n,} das vorangegangene Zeichen tritt n-mal oder öfter auf
- {n,m} das vorangegangene Zeichen tritt mindestens n-mal und maximal
- m-mal auf

2.21.3 Beispiele:

Datum
- [0-9]{1,2}\.[0-9]{1,2}\.[0-9]{1,4}

Ipv4
- \<(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\>'

UNummer
- [uU][1-9][0-9]{4}'

Quellcode:
```
let str_number = "123"
let regel ="^[+-]{0,1}[1-9][0-]{0,5}$"
let regex = try NSRegularExpression(pattern: regel, options; [])
let matches =
    regex.numberOfMatches(  in: str_number,
        options: [],
        range; NSMakeRange(0, str_number, str_number.utf16.count)
if matches == 1 {
  print("Es ist eine korrekte Zahl")
}
```

3 Funktionen, Dictionaries und Klassen

3.1 Funktionen

Eigenschaften:

- Parameter sind per Default nicht veränderbar (immutable)
- Ein var-Parameter kann in der Methode verändert werden.
- **Inout-Parameter werden bei der Übergabe kopiert und beim Verlassen der Methode komplett wirder zurückkopiert!**
  - *copy-in copy-out* or *call by value result*
  - https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID545

```
https://github.com/apple/swift-
evolution/blob/master/proposals/0046-first-label.md

func print1() {
    print("hallo")
}
Aufruf:  print1()


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


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


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



// Mit Benennung des Parameters
func Inch2Cm(inch value:Double) -> Double{
    return value *2.54
}
```

```
Aufruf:
let cm=1.23
let inch=Inch2Cm(inch:cm)
```

```
func add1(aa a:Int32, bb b:Int32)-> Int32{
    return a+b
}
Aufruf:
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
}
Aufruf:
let c = add3(12, b:33)
```

3.1.1 Reference-Parameter durch inout-Parameter

```
func add2(a:Int32, b:Int32, inout c:Int32  ){
    c=a+b
}
var c2:Int32=0
add2(a: 12,b: 33, c:&c2)
```

3.1.2 Tupel-Rückgabewert

```
func addsub(a:Int32, b:Int32)->(Int32, Int32){
    return (a+b,a-b)
}
Aufruf:
let ergs = addsub(12,4)
print(ergs)                    (8,16)
print(ergs.0, ergs.1)          (8,16)
```

```
func addsub(a:Int32, b:Int32)->(add:Int32, sub:Int32){
    return (a+b,a-b)
}
Aufruf:
var items=addsub(a:12,b:4)
print("add: ",items.add,"sub: ",items.sub)


// mit einem optionalen Parameter
func addsub2(a:Int32, b:Int32,c:Int32=12)->(add:Int32, sub:Int32){
    return (a+b+c,a-b+c)
}
Aufruf:
var items2=addsub2(12,b:4)
print("add: ",items2.add,"sub: ",items2.sub)
```

3.1.3 optionale 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) ")
```

3.1.4 Variable 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) ")
```

3.1.5 defer

Mit defer wird ein Bereich angegeben, der immer nach der return-Anweisung ausgeführt wird. Ein quasi finally-Block in einer Funktion:

```
func read(filename:String) -> Int {
  let infile = open(filename)
  while data = read_from(infile) {
    ...
    if data=="error" {
      return
    }
  }

  defer {
    infile.close()
  }
}
```

Beispiel mit zwei defer-Anweisungen:

```
func copy(filename1:String, filename2:String) -> Int {
  if let infile = open(filename1) {
    defer {
      infile.close()
    }
    if let outfile = open(filename2) {
      defer {
        outfile.close()
      }
      while data = read_from(infile) {
        let ok = write_to(outfile, data)
        if !ok{
          return
        }  // if
      }  // while
    }  // if let outfile = open(filename2) {
  } // if let infile = open(filename1) {
}
```

Wenn ein Fehler auftritt, warden die defer-Anweiosung von unten nacho ben abgearbeitet.
Egal was passiert, die Dateien werden immer geschlossen.

3.2 Verschachtelte Funktionen

Swift erlaubt, im Gegensatz zu Java, Funktionen in Funktionen zu deklarieren und aufzurufen.

```
func addsub(a:Int32, b:Int32)->(add:Int32, sub:Int32){
  func add(a:Int32, b:Int32)->Int32 {
```

```
    return a+b
  }
  func sub(a:Int32, b:Int32)->Int32 {
    return a-b
  }
  return ( add(a,b), sub(a,b) )
}
var items=addsub(12,b:4)

print("add: ",items.add,"sub: ",items.sub)
```

3.3 Closures

Im Prinzip eine anonyme Funktion.

Alte Technik:

```
let  data  = Array(1...20)
func  calc (n:Int32) -> Double {
    return sin(Double(n)*0.1)
}
let result = data.map(calc)
```

Neue Technik (3 Varianten):

```
let  data  = Array(1...20)
let result = data.map
    (
      {
        (n:Int32) -> Double in return sin(Double(n)*0.1)
      }
    )
let result = data.map
    (
      {
        (n:Int32) -> Double in sin(Double(n)*0.1)
      }
    )
let result = data.map
    (
      {
        sin(Double($0)*0.1)
      }
    )
```

3.4 try-catch, nur in Funktionen

Try-catch funktioniert nur innerhalb von Funktionen.

Beispiel:

```
func div(zaehler:Int32, nenner:Int32) throws ->Int32{
    enum MyErrors:ErrorType{
        case divZero(explanation:String)
        case other(explanation:String)
    }

    if nenner==0 {
        print("div 0")
        throw MyErrors.divZero(explanation:"Div by zero")
    }
    else {
        print("div")
        return zaehler/nenner
    }

}

let a:Int32=34
var b:Int32=2
b=0

do {
    var c:Int32 = try div(a,nenner:b)
}
```

3.5 Dictionaries

- 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()
```

3.6 Set, Mengen

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

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

3.7 Strukturen, Struct

- struct — Deklaration
- Instanz: — ohne new
- init — Konstruktor
- deinit — Dekonstruktor, sonst in C++ und C#
- Properties — setter / getter, à la C#
- self — this
- nil — null

Beispiel:

```
struct Mypoint {
    var x:Int32=0
    var y:Int32=0

    func printXY() {
        print("x: \(x) y: \(y) ")
    }
}
```

Aufruf:

```
var p1 = Mypoint()

p1.x=3
p1.y=4

p1.printXY()
```

3.8 struct, equal

```
struct Person: Equatable {
   var firstName: String
   var lastName: String
   var age: Int
   var city: String

   static func ==(lhs: Person, rhs: Person) -> Bool {
      return lhs.firstName == rhs.firstName && lhs.lastName == rhs.lastName && lhs.age == rhs.age
&& lhs.city == rhs.city
   }
}
```

Fortunately, Swift 4.1 can synthesize conformance for Equatable for us – it can generate an == method automatically, which will compare all properties in one value with all properties in another, just like above. So, all you have to do now is add Equatable as a protocol for your type, and Swift will do the rest:

3.8.1 struct, protocol

```
struct Book: Purchaseable {
  func buy() {
    print("You bought a book")
  }
}
```

4 Klassen

- class — Deklaration
- 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

4.1.1 Schutzebenen der Klassen

- private — nur innerhalb der Klasse
- public — komplett öffentlich
- internal — im Projekt öffentlich
- final — kann nicht verändert werden

4.1.2 Schutzebenen der Methoden

- private — nur innerhalb der Klasse
- public — komplett öffentlich
- internal — im Projekt öffentlich
- final — kann nicht verändert werden
- lazy — dient der späteren Inititialisierung
- weak — nicht im Automatic Reference Counting
- optional — optionale Methode in Protokollen
- required — Konstruktor muss bei Ableitung benutzt warden.

4.2 Klassendefinition

```
class  Temperatur{
  static let Nullpoint = 273.15
  var tKelvin:Double

var Temp : Double {
    get {  return  tKelvin  }
    set {  tKelvin = newValue  }
  }

var Temp : Double {
    get {  return  tKelvin  }
    set(value) {  tKelvin = value  }       auch optional
```

```
    }

    static func convertC2K(t:Double) -> Double {
      return  t - Nullpoint
    }
    init() {
      tKelvin = 273.15
    }
    init(temp  t:Double) {                    temp ist äußerer Name
      tKelvin=t
    }
}
    init(tKelvin:Double) {
      self.tKelvin = tKelvin
    }

    deinit {
      tKelvin=0
    }

    func getT() -> Double{
      return tKelvin
    }
    func addBy(intervalT t: Double) {
      tKelvin += t
    }
}
```

4.2.1 Anwendung:

```
let  t1 Temperatur()
t1.addBy(5)
// benannte Parameter
t1.addBy(intervalT:5)
let cel1 = t1.Temp                          Properties
t1.Temp = 122
// Aufruf einer statischen Methoden
let  tK = Temperatur.convertC2K(0)
```

4.3 Vererbung

```
final class  Point{                      darf nicht erweitert werden
  var x:Double = 0.0

init(x:Double)  {
    self.x = x
  }

func draw()  { … }
```

```
  }

  class  Line  :  Point{
    var y:Double = 0.0

    init(x:Double, y:Double)  {
      self.y = y
      super.init(x:x)       hier nicht als erste Anweisung, korrekt
    }

    override func draw()  { … }
    func save(filename:String)  { … }
  }
```

Beispiel:

```
class Auto:CustomStringConvertible{

    private var name:String=""
    private var laenge:Double=0.0

    // required: muss immer aufgerufen werden
    required init(name:String, laenge:Double){
        self.name=name
        self.laenge=laenge
    }
    init(name:String){
        self.name=name
        self.laenge=100
    }

    // toString
    var description :String {
        return "Auto: \(name)  \(laenge)"
    }

    // Computed Properties
    var Name:String {
        get { return name }
        set { name=newValue }
    }
    var Laenge:Double {
        get { return laenge }
        set { laenge=newValue }
    }

    func drucken() {                          Polymorphismus
        print(description)
    }
} // Auto
```

```
var auto:Auto = Auto(name:"Bentley Veyron",laenge:4.46)  // 175.7"
print(auto)
auto.name="myVeyron"    // funktioniert nur innerhalb einer Datei !
print(auto)

auto.Name = "Bentayga"
print(auto)
auto.drucken()



class Bentley:Auto{

    private var ps:Int32=1001

    init(name:String, laenge:Double, ps:Int32){
        self. ps = ps
        //super.init(name:name,laenge:laenge)       1. Variante
        super.init(name:name)                       2. Variante
    }

    required init(name: String, laenge: Double) {
        fatalError("init(name:laenge:) has not been implemented")
    }

    // toString
    override var description :String {
        return  super.description +  "Bentley: \(ps)"
    }

    // Computed Properties
    var Ps:Int32 {
        get { return ps }
        set { ps=newValue }
    }


    override func drucken() {
        print(description)
    }

}

var bently1:Bentley =
          Bentley (name:"Bentley Veyron", laenge:4.46, ps:1001)
print(bently1)
bently1.drucken()
```

4.4 toString

```
class Point:CustomStringConvertible{
  var x:Double = 0.0

    // toString
    var description :String {
        return "x: \(x)"
    }

} // Point
```

4.5 convenience

Benötigt, wenn man mehr als einen Konstruktor haben möchte.

```
class BaseClass {
    var myString: String

    // default Konstruktor
    init () {
        myString = "mi"
    }

    // ein anderer Konstruktor
    init (aString: String) {
        myString = aString
    }

    // ein convenience Konstruktor
    convenience init(aInt: Int) {
        // Aufruf des anderen constructor , Java: this(1,2,3)
        self.init(aString: "Hallo, MI'ler")
    }
}

// 1. Abgeleitette Klasse, hat keinen designated initializer
class FirstSubclass : BaseClass {

    convenience init(aOBJ: AnyObject) {
        self.init(aString: "First")
    }
}
```

```
// SecondSubclass does not have a superclass with a designated initializer
// thus the following code with raise a compilation error - you cannot have
// a class without a designated initializer

class SecondSubclass {
    convenience init(unsedVar: AnyObject) {

    }
}

// Raises a "Super.init isn't called before returning from initializer" compilation
// error. When subclassing, all designated initializers must call one of the base
// class's designated init methods

class ThirdSubclass : BaseClass {
    var data: NSData

    init(someData: NSData) {
        data = someData
    }
}
```

4.6 Property Observers

Zwei Ereignisse können bei einer Änderung definiert werden:

- willSet — Aufgerufen, bevor einer Änderung durchgeführt wird
- doSet — Aufgerufen, nach einer Änderung durchgeführt wird

```
class Schiff {
    private var laenge:Int32=1000  {
      willSet(neueLaenge) {
        print("alter Name:", laenge,"Neue Länge: ", neueLaenge)
      }
      didSet(neueLaenge) {
        print("neuer Name wurde gespeichert: ", laenge)
      }
    }
}
```

4.7 Computed Properties

Dieses Verfahren wird auch mehr oder minder in C# benutzt. Man kann mit

```
class Schiff {
  private var laengeMeter:Double=0.0

  init(Meter laenge:double) {
    this.laengeMeter = laenge;
  }

  init(Yard laengeyard:double) {
    laengeMeter = laengeyard*0.9144;
  }

  var Yard: Double {
    get {  return laengeMeter /0.9144;}
    set {  laengeMeter = newvalue*0.9144; }
  }

  var Meter: Double {
    get {  return laengeMeter;}
    set {  laengeMeter = newvalue; }
  }


}
```

5 Protokoll, Protocol

```
protocol  ISave {
  func  save2DB(table:String)
  func  save2TXT(filename:String)

  //  ist über dem Klassennamen erreichbar
  static func  convert2Int(x:Double)  -> Int
}

final  class  Point: ISave{
  var x:Double = 0.0
  init(x:Double)  {
    self.x = x
  }
  func  save2DB(table:String)  {
  }
  func  save2TXT(filename:String)  {
  }

}
Anwendung:
var p1:Point = Point(12)
var save = p1 as ISave

if let save = p1 as? ISave {
  print(save)
}
else{
  print("kein ISave")
}
```

5.1.1 Swift-Protokolle

AbsoluteValuable
    func <(lhs: Self, rhs: Self) -> Bool
    func <=(lhs: Self, rhs: Self) -> Bool
    func ==(lhs: Self, rhs: Self) -> Bool
AnyCollectionType
AnyObject
ArrayLiteralConvertible
BidirectionalIndexType
BitwiseOperationsType
BooleanLiteralConvertible
BooleanType
CVarArgType
CollectionType
Comparable

CustomDebugStringConvertible
CustomLeafReflectable
CustomPlaygroundQuickLookable
CustomReflectable
CustomStringConvertible
DictionaryLiteralConvertible
Equatable
    func ==(lhs: Self, rhs: Self) -> Bool
ErrorType
ExtendedGraphemeClusterLiteralConvertible
FloatLiteralConvertible
FloatingPointType
ForwardIndexType
GeneratorType
Hashable
Indexable
IntegerArithmeticType
IntegerLiteralConvertible
IntegerType
IntervalType
    var start: Self.Bound { get }
    var end: Self.Bound { get }
    var isEmpty: Self.Bound { get }
    func clamp(intervalToClamp: Self) -> Self
    func overlaps<I : IntervalType where I.Bound == Bound>(other: I) -> Bool
LazyCollectionType
LazySequenceType
MirrorPathType
MutableCollectionType
MutableIndexable
MutableSliceable
NilLiteralConvertible
OptionSetType
OutputStreamType
RandomAccessIndexType
RangeReplaceableCollectionType
RawRepresentable
ReverseIndexType
SequenceType
SetAlgebraType
SignedIntegerType
SignedNumberType
Streamable
Strideable
StringInterpolationConvertible
StringLiteralConvertible
UnicodeCodecType
UnicodeScalarLiteralConvertible
UnsignedIntegerType

5.1.2 Protocol Hashable

```
class  Person  :  Hashable{
  var vname:String = ""
  var nname:String = ""
  init(name:String)  {
    self.name=name
  }
  var hashable: Int  {                          getHashwert
    var hash1 = vname.hashValue
    var hash2 = nname.hashValue
    return hash1+ hash2
  }
}
```

5.1.3 Protocol Comparable

```
class Person : Comparable{
   var vname:String = ""
   init(name:String)  {
      self.name=name
   }

   func  < (p1:Person, p2:Person) -> Bool
      return  p1.name  < p2.name
   }


   func  <= (p1:Person, p2:Person) -> Bool
      return  p1.name <= p2.name
   }


   func  > (p1:Person, p2:Person) -> Bool
      return  p1.name  > p2.name
   }


   func  >= (p1:Person, p2:Person) -> Bool
      return  p1.name  >= p2.name
   }

}
```

6 Versionsnummern

Abfrage der Versionen

6.1 iOS-Abfrage

```
if #available(iOS 9.2, macOS 10.11, *) {
  // ausführen, wenn zumindest iOS 9.2 oder macOS 10.11
  // zur Verfügung steht
  print("iOS>=9.2 oder macOS >= 10.11")
} else {
  // bei älteren Versionen ausführen
  print("zu alt")
}
```

6.2 Abfrage nach tvOS

```
#if os(tvOS)
  print("läuft unter tvOS")
#else
  print("anderes OS")
#endif
```

6.3 Abfrage nach der Intel-Plattform

```
#if arch(x86_64)
  print("x86-Platform mit 64-Bit-CPU")
#endif
```

6.4 Abfrage nach der Swift-Version

```
#if swift(>=3.0)
  print("Die Swift-Version ist 3.0 oder größer.")
#endif
```

7 Debug-Modus

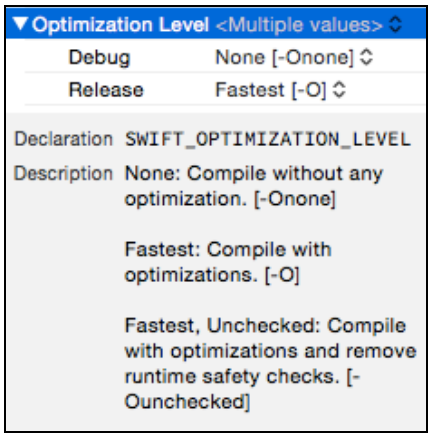

Abbildung 3 Einstellen des Debug-Modus

Parameter:

- SWIFT_OPTIMIZATION_LEVEL = -Onone // debug
- SWIFT_OPTIMIZATION_LEVEL = -O // release
- SWIFT_OPTIMIZATION_LEVEL = -Ounchecked // unchecked release

7.1 assert

Diese Funktion dient der Überprüfung eines Wertebereiches einer Variablen oder Konstanten. Sie stammt aus der Programmiersprache C. **Die Überprüfung findet nur im Debug-Modus statt. In der Release-Version werden diese Anweisungen entfernt.** Der is-Operator prüft, ob der Datentyp korrekt ist. Die Funktion „assert“ überprüft detaillierter.

Aufruf:

assert(Bedingung, Fehlertext)

Beispiel:

```
assert(x>=0,"Modul calc, Methode Wurzel, x muss >= 0 sein")
assert(item!=nil,"Modul DB, Methode save2DB, item darf nicht nil sein")
```

7.2 assertionFailure

Diese Funktion zeigt dem Compiler, der Laufzeitumgebung an, dass der folgende Code nie erreicht wird. Das Programm stoppt. Man kann eine Zeilennummer mitgeben. Es zeigt also an, das ein interner Fehler verursacht wurde (siehe switch/case/default-Fall).

Deklaration

func **assertionFailure**(

@autoclosure message: () -> String = default,
file: StaticString = default, line:
UInt = default
)

Beispiel:

```
assertFailure("hier geht es nicht weiter")
assertFailure("hier geht es nicht weiter","Modul save")
assertFailure("hier geht es nicht weiter","Modul save", 100)
```

7.3 precondition

Diese Variante ist eine Sicherheitsfunktion, die eine Bedingung abprüft. Sie bleibt auch im Release-Modus.

Beispiel:

```
let x=-1
precondition(x>=0,"Modul calc, Methode Wurzel, x muss >= 0 sein")
print("die Wurzel wurde berechnet")        wird nicht ausgeführt
```

7.3.1 preconditionFailure

Diese Variante ist eine Sicherheitsfunktion, die eine Bedingung abprüft. Sie bleibt auch im Release-Modus. Hier wird auch eine Meldung ausgegeben.

Beispiel:

```
let x=-1
precondition(x>=0,"Modul calc, Methode Wurzel, x muss >= 0 sein")
print("die Wurzel wurde berechnet")        wird nicht ausgeführt
```

7.4 fatalError

Dieser Funktion sollte eine interne, eigene Abprüfung voran gehen. Beim Aufruf erscheint eine Meldung und das Programm terminiert.
Deklaration:

func **fatalError**(
@autoclosure message: () -> String = default,
file: StaticString = default, line:
UInt = default
)

Beispiel:

```
fatalError("hier geht es nicht weiter, Ende")
fatalError("hier geht es nicht weiter, Ende ","Modul save")
```

```
fatalError("hier geht es nicht weiter, Ende ","Modul save", 100)
```

7.5 CheatSheet

	debug	**release**	**release**
function	-Onone	-O	-Ounchecked
assert()	Ja	Nein	Nein
assertionFailure()	Ja	Nein	Nein **
precondition()	Ja	Ja	Nein
preconditionFailure()	Ja	Ja	Ja**
fatalError()*	Ja	Ja	Ja

Erläuterung:

- Ja Programm terminiert
- Nein Programm terminiert nicht
- * Programm terminiert immer
- ** Der Optimierer nimmt an, die Funktion wird nicht aufgerufen

7.6 debugPrint

Diese Funktion schreibt Zwischenwert in die Default-Ausgabe.
Deklaration:

```
func debugPrint(items: Any..., separator: String = default, terminator: String = default)
```

7.7 dump

Ausgabe einer variable, Liste in die Default-Ausgabe.
Deklaration:

```
func dump<T, TargetStream : OutputStreamType>(_: T, inout _: TargetStream, name: String?, indent: Int, maxDepth: Int, maxItems: Int)
```

Aufruf:

- x->dump()

8 Literatur und Links

8.1 Literatur

Switft 2
Michael Kofler
Rheinwerk
ISBN: 978-3-8362-3651-5

iOS-Apps programmieren mit Swift
Christian Bleske
dpunkt.verlag
ISBN 978-3-86490-263-5

Durchstarten mit Swift
Stefan Popp & Ralf Peters
O'Reilly
ISBN: 978-3-96009-005-2

8.2 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/

9 Indexverzeichnis

!

$

?

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

V

W

Z