

Praktikum iOS-Entwicklung

Wintersemester 2015/16

Prof. Dr. Linnhoff-Popien

Florian Dorfmeister, Marco Maier



Gemeinsames Themen-Brainstorming am 2.12.!

Wir suchen Ideen für die Praxisphase!

Das heißt:

- Eure Ideen sind gefragt!
- Vorstellen der Ideen in 5-minütigen Präsentationen
- Vergabe der Themen mit Beginn der Programmierphase

OBJECTIVE-C UND SWIFT

Zunächst wurde für iOS hauptsächlich in **Objective-C** entwickelt.

Auf der WWDC 2014 wurde die bis dahin unbenutzte Sprache **Swift** vorgestellt.
Aktuelle Version ist Swift 2.0 (Open Source).

Code in **C** und **C++** wird von iOS genauso unterstützt.

Und was machen wir?

Hi Michael,

[...] Swift is a new option for developing on the platform. We have no plans to drop C, C++ or Objective-C. If you're happy with them, please feel free to keep using them.

-Chris

<http://lists.apple.com/archives/xcode-users/2014/Jun/msg00024.html>

Verfügbare Apps im iOS-AppStore (Stand Juni 2015):

ca. 1.500.000

Gesamtanzahl der App-Downloads:

> 100.000.000.000

Fast alle dieser Apps sind in Objective-C, C oder C++ entwickelt worden.

In dieser Veranstaltung werden wir uns deshalb mit beiden Sprachen beschäftigen.

GRUNDLAGEN IN OBJECTIVE-C

Wikipedia:

- Objective-C is a general-purpose, high-level, object-oriented programming language that adds Smalltalk-style messaging to the C programming language.

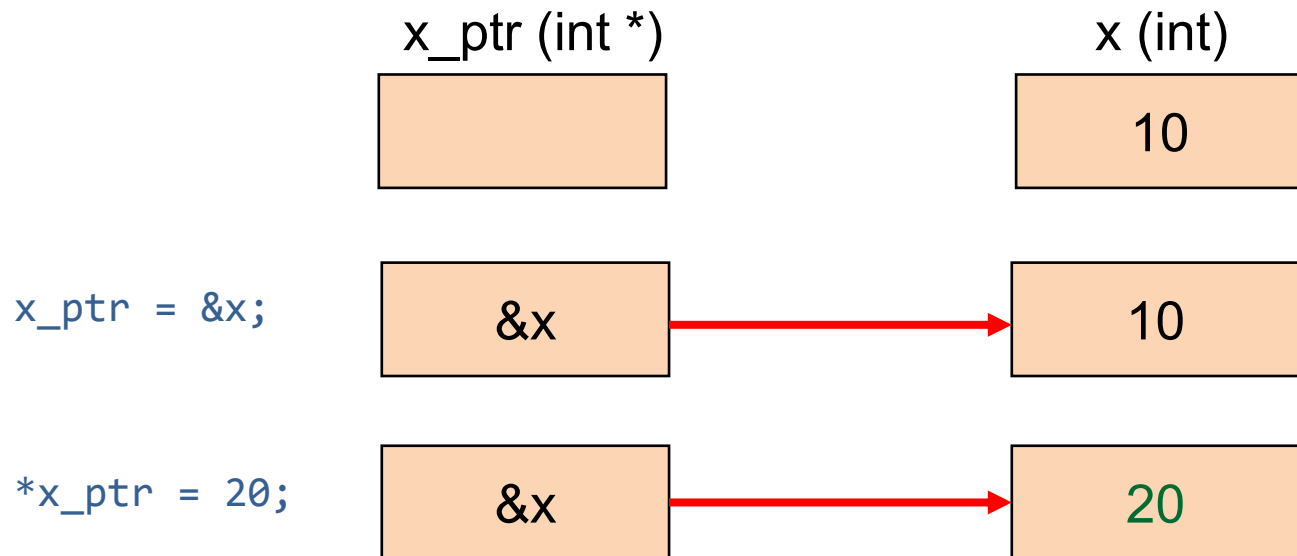
Objective-C basiert auf C.

- Jedes C Programm kann mit einem Objective-C-Compiler kompiliert werden. Die Objective-C LLVM erlaubt sogar überall die Verwendung von C und C++!
- Objective-C erweitert C um Objektorientierung.
- Methodenaufrufe erfolgen mittels **message passing**
- iOS: Objective-C und Cocoa Touch Library (UI Framework)

Swift und Objective-C können zusammen im selben Projekt verwendet werden!
(s. Hausaufgabe)

```
int x = 10; // Variable x mit Wert 10
&x; // Adresse von x Bsp.: 0x000001
```

```
int *x_ptr; // Pointer vom Typ int *
&x_ptr; // Adresse von x_ptr Bsp.: 0x000034221
```



```
#include <stdio.h>
int main(int argc, const char *argv[])
{
    int i = 1337;
    int *addressOfI; //Pointer vom Typ 'int *'
    addressOfI = &i; //Referenzübergabe: Übergeben der Adresse von i

    printf("address of i: %p\n", addressOfI); //address of i: 0xbffff738
    printf("value of i: %d\n", i); //value of i: 1337

    addressOfI = 10;
    //Error: Incompatible Integer to Pointer Conversion Assigning to 'int *'
    from 'int'

    *addressOfI = 10;
    printf("value of i: %d\n", *addressOfI); //value of addressOfI: ?
    printf("value of i: %d\n", i); //value of i: ?
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>

typedef struct { //Deklaration eines zusammenhängenden Datenblocks
    int alter;
    float groesse;
} Person;

void agePerson(Person *p){
    p->alter += 1;
    //'->': Zeiger dereferenzieren und auf enthaltene members zugreifen
}

int main(int argc, const char *argv[]){
    Person *erwin = (Person *)malloc(sizeof(Person));
    //Speicher auf dem Heap allokieren

    erwin->groesse = 1.85;
    erwin->alter = 23;
    agePerson(erwin);
    printf("Erwin ist %d Jahre alt\n", erwin->alter); //...24 Jahre...
    free(erwin); //Speicher wieder freigeben
    erwin = NULL; //Zeiger 'löschen'
    return 0;
}
```

Objekte sind wie `struct` Abschnitte im Speicher.

Objekte haben Instanzvariablen anstelle von members.

- (Instanzvariablen haben je einen Namen und einen Typ)

Die Struktur der Objekte wird in Klassen definiert.

Objekte müssen instanziiert **und** initialisiert werden:

```
Person *erwin = [[Person alloc] init];
```

```
Person *erwin = [Person new]; //Alternative, einfacher
```

Objekte haben ausführbare Methoden.

Objekte werden in Header-Files (.h) deklariert...

- (vergleichbar mit einem Interface in Java)
- ...und in Source-Files (.m) implementiert.
- beschreiben die Funktionalität einer Klasse, ohne die Umsetzung zu zeigen

Methodensignaturen müssen innerhalb einer Klasse eindeutig sein!

- (Signaturen müssen sich in den Argumenten unterscheiden und nicht nur in den Rückgabewerten.)

```
#import <Foundation/Foundation.h>

@interface Person : NSObject // Person erbt von NSObject
{
    // alle Instanzvariablen gehören in diesen Block
    @public int alter;
    //@public implizit. Möglich: @protected, @private (analog zu Java)

    float groesse;
}

// alle Methoden außerhalb der geschweiften Klammern
- (void)setAlter:(int)neuesAlter;
- (int)alter;
- (void)setGroesse:(float)neueGroesse;
- (float)groesse;
- (void)setAlter:(int)neuesAlter undGroesse:(float)neueGroesse;

@end
```

```
#import "Person.h"
@implementation Person

- (void)setAlter:(int)neuesAlter {
    alter = neuesAlter;
}

- (int)alter {
    return alter;
}

- (void)setGroesse:(float)neueGroesse {
    groesse = neueGroesse;
}

- (float)groesse {
    return groesse;
}

- (void)setAlter:(int)neuesAlter undGroesse:(float)neueGroesse{
    alter = neuesAlter;
    groesse = neueGroesse;
}
@end
```

@property (nonatomic) typ name;

- Implizite Deklaration von getter und setter im Interface.
- Und: implizite Deklaration von Instanzvariablen (Prefixed:_name)!
- Alle Objekttypen müssen als Pointer (*name) angegeben werden!

@synthesize name;

- Automatische Implementierung der getter und setter.
- Seit iOS 6 nur noch nötig, falls getter und setter selbst implementiert werden (autosynthesize)

```
#import <Foundation/Foundation.h>

@interface Person : NSObject

@property (nonatomic, readwrite) int alter;
@property (nonatomic, readwrite) float groesse;

/**
 * nonatomic: Sollte immer angegeben werden.
 * readwrite (standard): erzeugt getter UND setter
 * readonly: erzeugt nur getter
 */

- (void)setAlter:(int)neuesAlter undGroesse:(float)neueGroesse;

@end
```

```
#import "Person.h"

@implementation Person

@synthesize alter, groesse; //kann i.d. Regel weggelassen werden

- (void)setAlter:(int)neuesAlter undGroesse:(float)neueGroesse{
    alter = neuesAlter;
    groesse = neueGroesse;
}

@end
```

Statische Variablen werden mit `static` deklariert.

Statische Variablen 'leben' nicht auf dem Stack und werden beim Verlassen der Methode nicht gelöscht.

Auf `theAnswer` kann nur aus der Methode `doSomethingWeird` heraus zugegriffen werden.

Mit statischen Variablen kann das Singleton-Pattern realisiert werden.

```
+ (int)doSomethingWeird
{
    static int theAnswer;
    if (!theAnswer)
        theAnswer = 42;
    return theAnswer;
}
```

```
// MyCoolObject.m

import "MyCoolObject.h"

@interface MyCoolObject()
// someValue ist nur innerhalb dieser Klasse zugänglich.
@property (nonatomic) int someValue = -1;
@end

@implementation MyCoolObject : NSObject

// valueFromPublicInterface ist in der MyCoolObject.h definiert, und daher eine
// öffentlich zugängliche Instanzvariable(sozusagen Teil der Public-API).
@synthesize valueFromPublicInterface = _valueFromPublicInterface;
@synthesize someValue = _someValue;

[...]

@end
```

```
#import <Foundation/Foundation.h>
#import "Person.h"

int main(int argc, const char *argv[])
{
    @autoreleasepool { //Bestandteil des ARC. Details folgen.
        Person *erwin = [[Person alloc] init]; //new Person() in Java

        [erwin setGroesse:1.85];
        NSLog(@"Erwin ist %.2f Meter groß", [erwin groesse]);

        [erwin setAlter:23 undGroesse:1.87];
        NSLog(@"Erwin ist %.2f Meter groß", [erwin groesse]);

        erwin = nil; //Zerstört das Objekt
    }
    return 0;
}
```

Methodenaufruf in Java:

- `myObject.hasAWeirdMethod(thatTakesAnArgument);`

Methoden sind immer an Code-Block gebunden (Der Teil zwischen den geschweiften Klammern)

Ziel des Methodenaufrufs steht zur compile-Zeit fest.

Objective-C: **Message Passing** anstelle von Calling!

Objective-C: Methodenaufruf ist Nachricht an ein Objekt.

Ziel erst zur Laufzeit bekannt. Das heißt:

- es ist kein type-checking möglich und
- es ist nicht sicher, ob Objekte auf Nachricht reagieren.

Aber: Das macht Objective-C sehr dynamisch!

- Stichwort: Categories (Details folgen)

Receiver Selector Argument

↓ ↓ ↙

`[person setAlter:neuesAlter
 undGroesse:neueGroesse];`

Instanz-Methoden werden an die Instanz,

- (z.B. `init`)

Klassen-Methoden dagegen an die Klasse gesendet.

- (z.B. `alloc`)

Klassen-Methoden haben keinen Zugriff auf Instanzen oder deren Variablen
(Analog zu statischen Methoden in Java)

Klassen-Methoden werden mit '+' deklariert, Instanzmethoden mit '-'.

- `+ (returnSomething)AndDoSomethingWeird;`

```
Person *erwin = [[Person alloc] init];
```

- `alloc` wird an die Klasse `Person` gesendet => Instanz der `Person`
- Impliziter Initialisierer `init` wird an Instanz geschickt.
- Möglich: Eigene Initialisierer definieren!
- In `Person.h`:
 - - `(id)initWithAlter:(int)nAlter groesse:(float)nGroesse;`
- In `Person.m`
 - - `(id)initWithAlter:(int)nAlter groesse:(float)nGroesse {`

```
    self = [super init];
    if(self){ // Falls self instanziiert wurde
        [self setAlter:nAlter];
        [self setGroesse:nGroesse];
    }
    return self;
}
```

Aufruf

- `Person *erwin = [[Person alloc] initWithAlter:23 groesse:1.85];`

id:

- "Ein Zeiger auf ein beliebiges Objekt"
Return-Typ aller init-Methoden - muss unspezifisch sein wegen Vererbung!
Analog zu `(void *)` in C

isa:

- Jedes Objekt hält seinen eigenen Klassennamen
Entscheiden darüber, welche Klasse für einen Methodenaufruf ausgewählt wird.

self:

- Impliziter Pointer auf das 'eigene' Objekt
- in anderen Sprachen häufig `this`

Alle Objekte erben von `NSObject`

- (Analog zur Oberklasse `Object` in Java)

Keine Mehrfachvererbung möglich!

- (Objekte haben genau eine Oberklasse, die mit `super` angesprochen werden kann) Bsp.:
`self = [super init];`

Methoden überschreiben möglich

- (Methoden können denselben selector haben, aber keine unterschiedlichen return-types.)
- Bsp.:
`-(id)init //überschreiben des Standard-Initialisierers`

```
#import "Person.h"

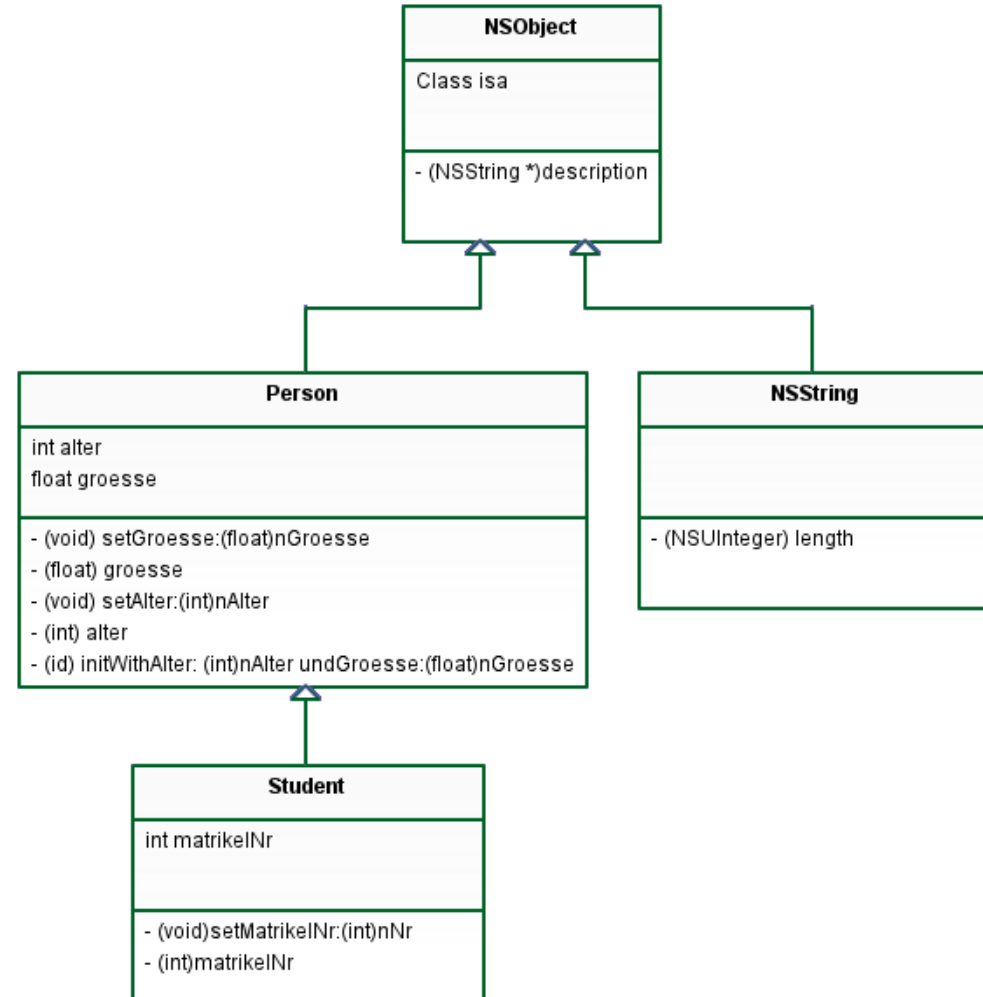
@interface Student : Person

@property (nonatomic) int matrikelNr;

- (void)setMatrikelNr:(int)nNr;
//oder @synthesize in Student.m

@end
```

Kinder erben Instanzvariablen
und Methoden ihrer Eltern.



Jede Klasse sollte genau einen designierten Initialisierer haben! (implizit oder selbst implementiert)

```
#import "Student.h"

@implementation Student

- (id)init //überschreiben des Standard-Initialisierers
{
    return [self initWithMatrikelNr:1337];
}

- (id)initWithMatrikelNr:(int)nNr //neuer designierter Initialisierer
{
    self = [super init];    //Initialisieren der Oberklasse Person
    if(self){
        //Nur wenn Initialisierung von Person erfolgreich
        [self setMatrikelNr:nNr]; //setzen der Matrikelnummer
    }
    return self;
}

@end
```

Instanzvariablen vom Typ eines beliebigen Objekts

- Bisher wurden nur atomare Datentypen betrachtet...
- Nicht atomar Bsp.:
 - `@property (strong, nonatomic) NSString *name;`
 - `// erzeugt implizit die Instanzvariable NSString* _name`

Objektinstanzvariablen sind immer Zeiger.

- Objekte "befinden" sich nicht in anderen Objekten.

Objekte "besitzen" andere Objekte!

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, guard, repeat, defer, ...)
- #available-checks
- automatisch generierte „Header files“

```
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  $\pi$  = 3.14159
var ♣ = "Ich bin ein Kleeblatt"

 $\pi$  = 4 // compile-time error
♣ = "Ich bin eine Rose" // ♣ ist jetzt eine Rose

println("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*.

```
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 = possibleNumber.toInt()
```

`toInt()` ist eine String-Methode. Sie liefert einen `Int?` oder `optional Int` zurück.

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

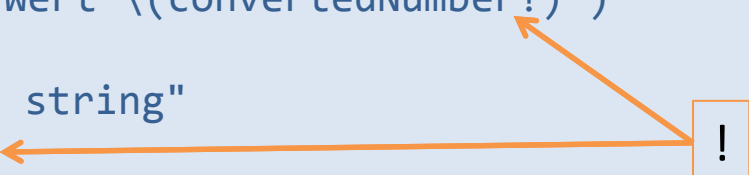
```
var anotherCode: Int = 404  
anotherCode = nil // compiler Fehler
```

`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 {  
    println("converted Number enthält den Wert \((convertedNumber!)\)")  
}  
let possibleString: String? = "An optional string"  
let forcedString: String = possibleString!  
  
if let realNumber = possibleNumber.toInt() {  
    println("Der String enthält die Zahl \((realNumber)\)");  
}else{  
    println("'\'(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 Werte 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
println("Die Bestnote ist eine \(note) und wird geschrieben als \(beschr)")

let (nurDieBesteNote, _) = bestanden // "entpackt" nur den ersten Wert
let einsKommaNull = bestanden.0      // s.o.

let durchgefallen = (note: 5.0, beschreibung: "mangelhaft")
println("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* mehrmals gleichzeitig initialisiert 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)
println("Die x-Koordinate des Punktes lautet \ \(pkt.x)")

let krs = Kreis()

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

println("Die x-Koordinate des Kreises lautet \ \(krs.punkt.x)")
krs.punkt.x = 14
println("... 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 {  
            println("Kurz davor, \ (newValue)% abzuschließen")  
        }  
        didSet {  
            if( status > 100 ){  
                status = 100  
            }  
            println("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 {  
            println("Kurz davor, \(newValue)% abzuschließen")  
        }  
        didSet {  
            if( status > 100 ){  
                status = 100  
            }  
            println("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 + "!"  
}  
  
println(sayHello("Anna")) // Hello, Anna!
```

```
func sayHello(personName: String) -> String {  
    return "Hello, " + personName + "!"  
}  
  
println(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(){
        println("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 numWagons = 0  
    override func makeNoise(){  
        println("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(){}  
}
```

```
class Train: Vehicle {  
    var numWagons = 0  
    override func makeNoise(){  
        println("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`.

SPEICHERVERWALTUNG

```
#import <stdlib.h>

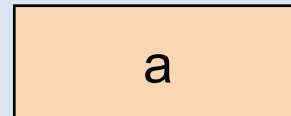
int main(int argc, const char *argv[]){

    int *a, *b;

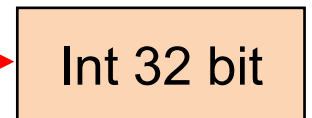
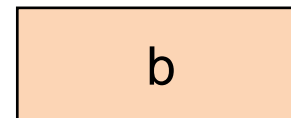
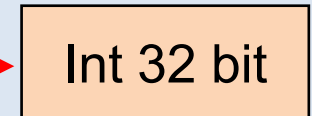
    a = malloc(sizeof(int));
    b = malloc(sizeof(int));

}
```

Pointer (int)



Speicher



```
#import <stdlib.h>

int main(int argc, const char *argv[]){

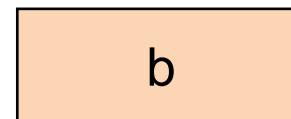
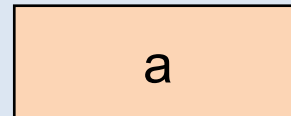
    int *a, *b;

    a = malloc(sizeof(int));
    b = malloc(sizeof(int));

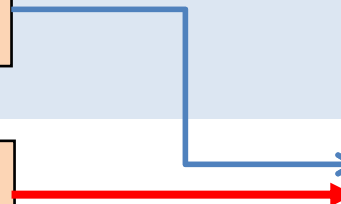
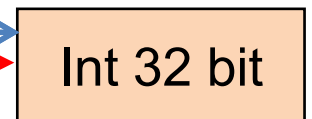
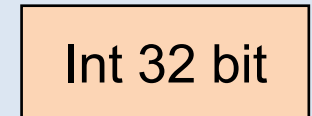
    a = b;

    return 0;
}
```

Pointer (int)



Speicher



Objekte ohne Besitzer sollen zerstört werden.

Objekte mit mindestens einem Besitzer dürfen nicht zerstört werden!

In C:

- Verwaltung von Hand mit `malloc` und `free`

In Objective-C ursprünglich:

- `retain` und `release`

ARC zählt Referenzen auf Objekte.

ARC verwaltet "Besitz" von Objekten.

- (Objekte gehen automatisch in den Besitz dessen über, der die Instanziierung veranlasst.)

ARC löscht Objekte, die keinen Besitzer haben automatisch aus dem Speicher.

- (Ähnlich der Garbage Collection in Java)

Nur Objective-C:

Alle Objekte, die innerhalb eines `@autoreleasepool{}` erzeugt werden, verlieren ihre Besitzer beim Verlassen des Blocks.

Ziel: Einem Objekt seinen Besitzer entziehen.

Die Variable, die auf ein Objekt zeigt...

- auf ein anderes Objekt zeigen lassen
- auf `nil` setzen
- wird selbst gelöscht

Swift bietet für Klassen Deinitialisierer `deinit`, der aufgerufen wird, direkt bevor das Objekt zerstört wird.

```
class Person{  
    var name: String  
    ...  
    deinit{  
        println("\(name) wird jetzt gelöscht")  
    }  
}
```

Problem: Zwei Objekte zeigen gegenseitig aufeinander!

Lösung:

- Beziehung zwischen Objekten als Eltern-Kind-Beziehung begreifen.
- Eltern 'besitzen' ihre Kinder, aber nicht umgekehrt.

Lösung:

- schwache Referenzierung, damit kein Besitz vom referenzierten Objekt ergriffen wird!

Objective-C: `__weak Student *myStudent;`

Swift: `weak var myStudent: Student?`

`myStudent` wird automatisch `nil` wenn das Objekt keinen Besitzer mehr hat!

Über properties können Memory Management Attribute definiert werden

Bsp.:

- `@property (nonatomic, strong) NSString *name;`
- Möglich sind:
 - **Strong** (Default)
 - Starke Referenzierung (Besitz von Objekt)
 - **Weak**
 - Schwache Referenzierung (Keinen Besitz)
 - **Copy**
 - Referenziert auf die Kopie des Objekts
 - Bsp.: Sicherheitsaspekt, falls nicht bekannt, was mit Originalobjekt gemacht wird
 - **Unsafe_unretained**
 - für direct assignments Bsp.: int

<https://developer.apple.com/library/ios/navigation/>

→ The Swift Programming Language → Language Guide

<http://www.raywenderlich.com/74438/swift-tutorial-a-quick-start>

Was ist neu in Swift 2?

<http://www.raywenderlich.com/108522/whats-new-in-swift-2>