



Praktikum – iOS-Entwicklung

Wintersemester 2019/20

Prof. Dr. Linnhoff-Popien

Markus Friedrich, Christoph Roch

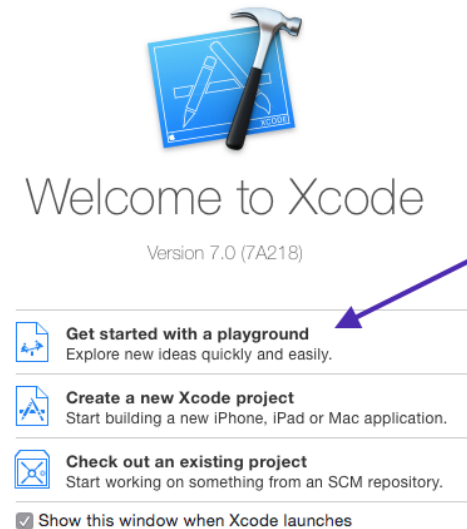
Swift

Weiterführende Konzepte

Swift – Was bisher geschah...

- Einführung ins Swift mit kurzem Vergleich mit ObjectiveC
- Tool, um Swift kennenzulernen: **Playground**
- Betrachtete Sprachelemente:
 - Variablen, Konstanten und Collection Types
 - Verzweigungen und Schleifen
 - Funktionen
 - Strukturen, Klassen und Enumerationen
 - Optionals
 - Protokolle
 - Fehlerbehandlung

×



No Recent Projects

Open another project...

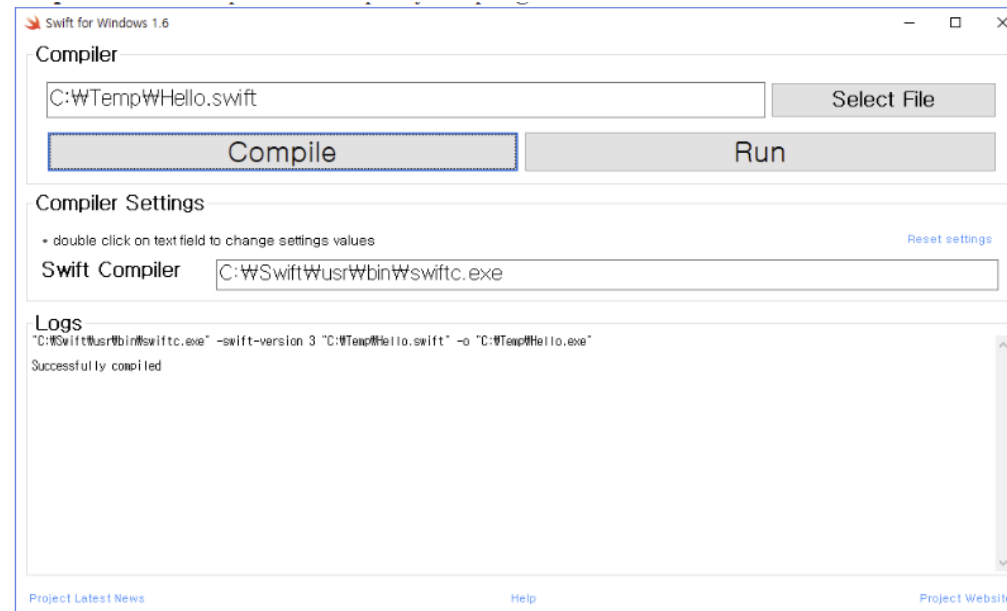
Swift – Agenda für diese Einheit

Betrachtete Sprachelemente:

- Speicherverwaltung
- Vererbung
- Subscripts
- Extensions
- Properties im Detail
- Funktionale Programmierung & Closures
- Generics
- Weitere Standardfunktionen

Hands-On: Playground Alternativen

- <http://online.swiftplayground.run/>
- <https://swiftforwindows.github.io/>



Werttypen vs. Referenztypen

Swift:

- Klassen, Closures => Referenztypen
- Structs, Enums => Werttypen

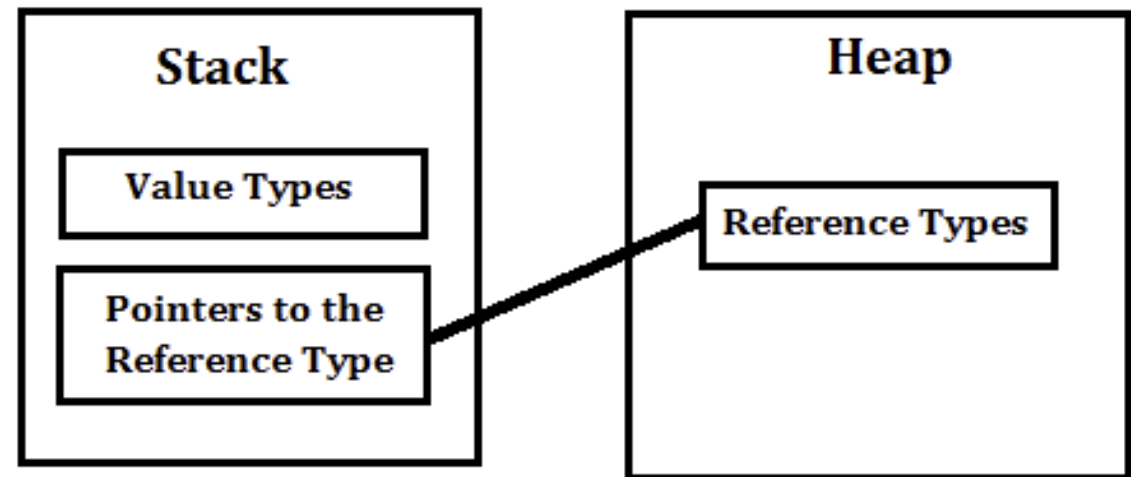
Wann was verwenden?

Structs:

- Kopien mit unabhängigem State
- Daten in mehreren Threads verwenden
- == macht Sinn (Equatable)

Klassen:

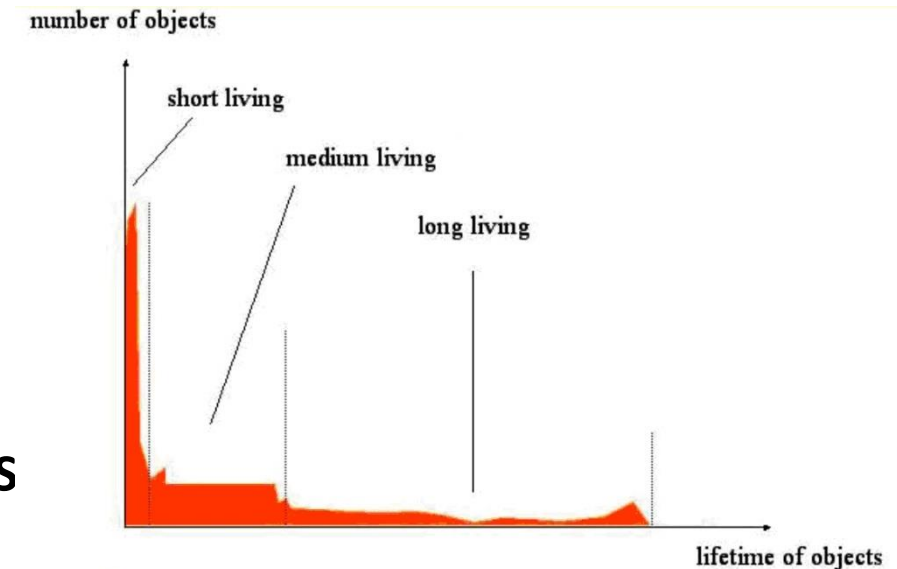
- Wenn geteilter, veränderbarer State gewünscht ist
- === macht Sinn (Identisch, also gleiche Adresse im Heap)



<http://net-informations.com/faq/general/valuetype-referencetype.htm>

Speicherverwaltung – Andere Sprachen

- **C:** manuell
 - Heap Allokationen mit malloc, Deallokationen mit free
- **C++:** manuell
 - Heap Allokationen mit new and delete (böse ☺)
 - Aber: Referenzzähler mit `std::shared_ptr`, `std::weak_ptr`, `std::unique_ptr`
- **C#** (und alle .net Sprachen), **Java** (und alle JVM Sprachen), **JavaS** (Googles V8): Generational Garbage Collection
- **Python:** Referenzzählung + (abschaltbare) Generational Garbage Collection



http://www.angelikalanger.com/Articles/EffectiveJava/49_GC.GenerationalGC/49_GC.GenerationalGC.html

Speicherverwaltung

- Swift verfügt über **keinen Garbage Collector** (vgl. C#, Java).
- „Aufgeräumt“ wird per **Referenzzählung** (Automatic Reference Counting ARC).

```
class Person {  
    init() {print(„init“)}  
    deinit {print(„deinit“)}  
}  
  
var a: Person? = Person()  
var b: Person? = a  
  
a = nil  
b = nil  
  
print(„Finished“)
```

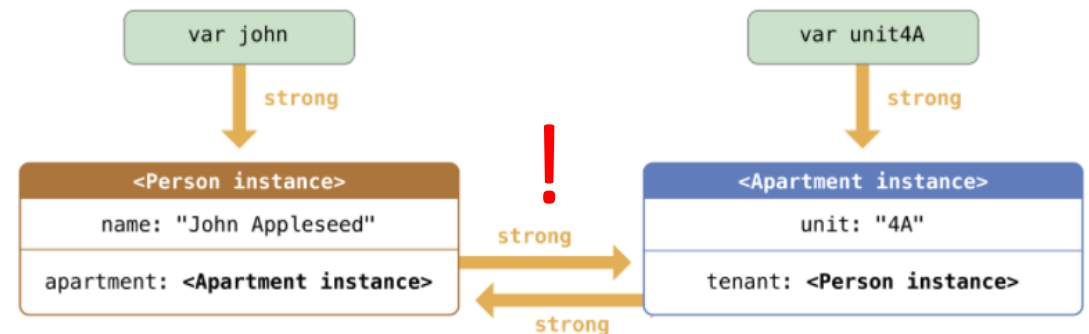
Wichtig: Ist nur relevant für Referenztypen (Klassen, Closures & Funktionen)!

Speicherverwaltung - Referenzzyklen

```
class Person {  
    let name: String  
    var apartment: Apartment?  
  
    init(name: String) {self.name = name}  
    deinit {print("deinit person")}  
}
```

```
class Apartment {  
    let unit: String  
    var tenant: Person?  
  
    init(unit: String) {self.unit = unit}  
    deinit {print("deinit apartment")}  
}
```

```
var john: Person?  
var unit4A: Apartment?  
john = Person(name: "John Appleseed")  
unit4A = Apartment(unit: "4A")  
  
john!.apartment = unit4A  
unit4A!.tenant = john
```



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

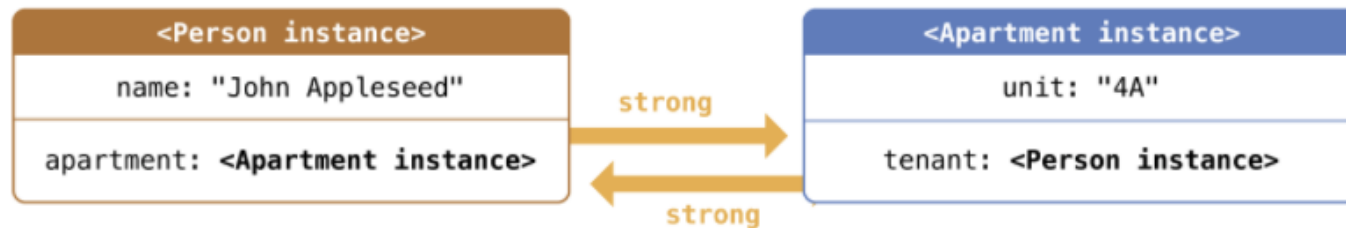
Speicherverwaltung - Referenzzyklen

```
john = nil  
unit4A = nil
```



var john

var unit4A



Beide Objekte
referenzieren sich
gegenseitig.
Memory Leak!

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

Speicherverwaltung – Referenzzyklen brechen

Es gibt drei verschiedene Referenztypen:

Strong:

- Kann zu Referenzzyklen führen

Weak:

- Schwache Referenz
- Wird automatisch auf `nil` gesetzt, wenn referenziertes Objekt deallokiert wird.

Unowned:

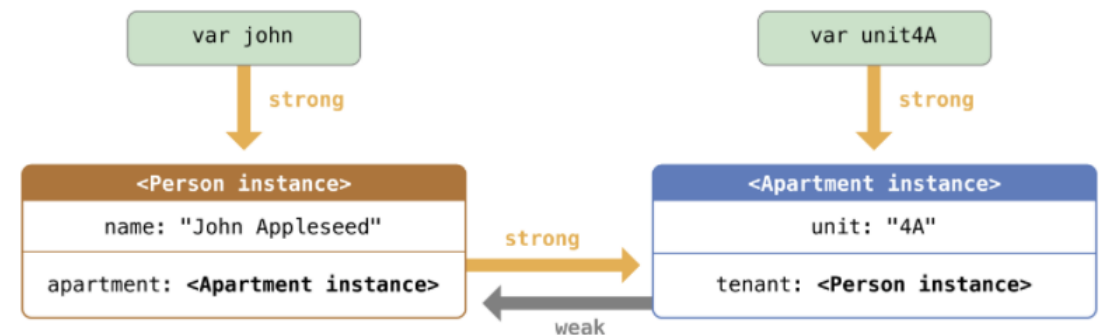
- Schwache Referenz
- Wird **nicht** automatisch auf `nil` gesetzt, wenn referenziertes Objekt deallokiert wird.
- Wenn auf deallokiertes Objekt zugegriffen wird, erfolgt ein **Laufzeitfehler**.

Speicherverwaltung – Schwache Referenzen

```
class Person {  
    let name: String  
    var apartment: Apartment?  
  
    init(name: String) {self.name = name}  
    deinit {print("deinit person")}  
}
```

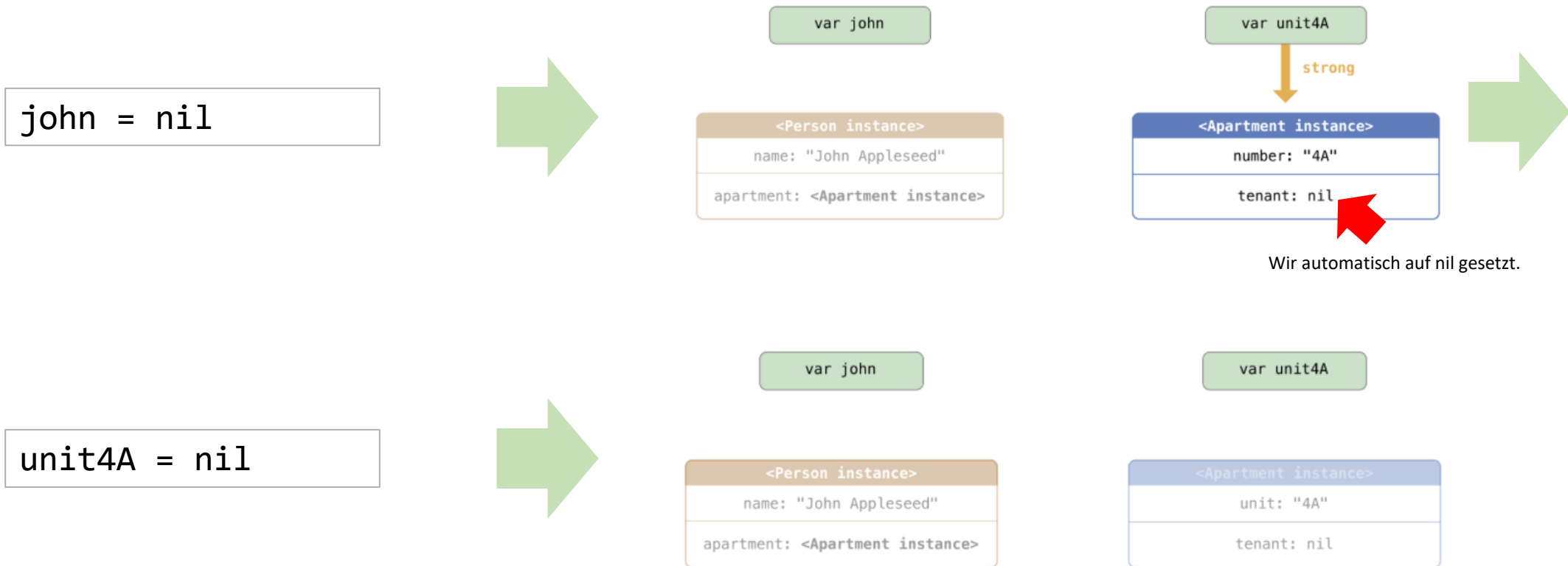
```
class Apartment {  
    let unit: String  
    weak var tenant: Person?  
  
    init(unit: String) {self.unit = unit}  
    deinit {print("deinit apartment")}  
}
```

```
var john: Person?  
var unit4A: Apartment?  
john = Person(name: "John Appleseed")  
unit4A = Apartment(unit: "4A")  
  
john!.apartment = unit4A  
unit4A!.tenant = john
```



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

Speicherverwaltung – Schwache Referenzen



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

Speicherverwaltung – „Unowned“ Referenzen

- Für eine „unowned“ Referenz wird angenommen, dass sie **immer einen Wert hat**.
- **Szenario:** Das Objekt, das die „unowned“ Referenz hält, hat die gleiche oder eine kürzere Lebenszeit.

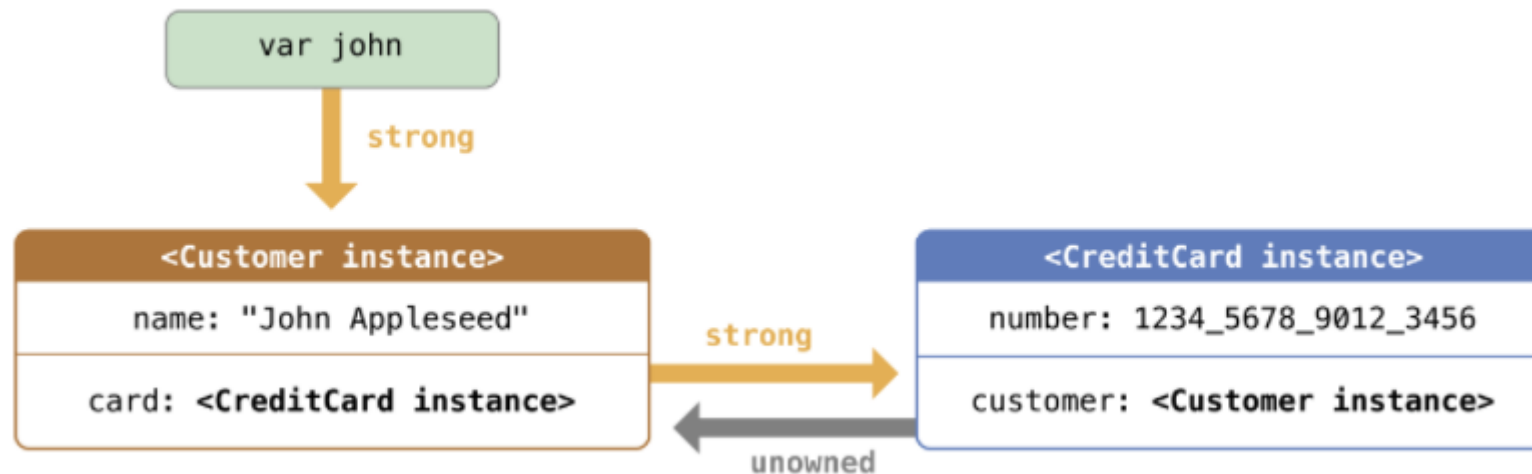
```
class Customer {  
    let name: String  
    var card: CreditCard?  
    init(name: String){  
        self.name = name  
    }  
    deinit {print("Customer deinit")}  
}
```

```
class CreditCard {  
    let number: UInt64  
    unowned let customer: Customer  
    init(number: UInt64, customer: Customer){  
        self.number = number  
        self.customer = customer  
    }  
    deinit {print("Card deinit")}  
}
```

- Customer Instanzen „leben“ länger als die korrespondierende CreditCard Instanz (hoffentlich).

Speicherverwaltung – „Unowned“ Referenzen

```
var john: Customer?  
john = Customer(name: "John Appleseed")  
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

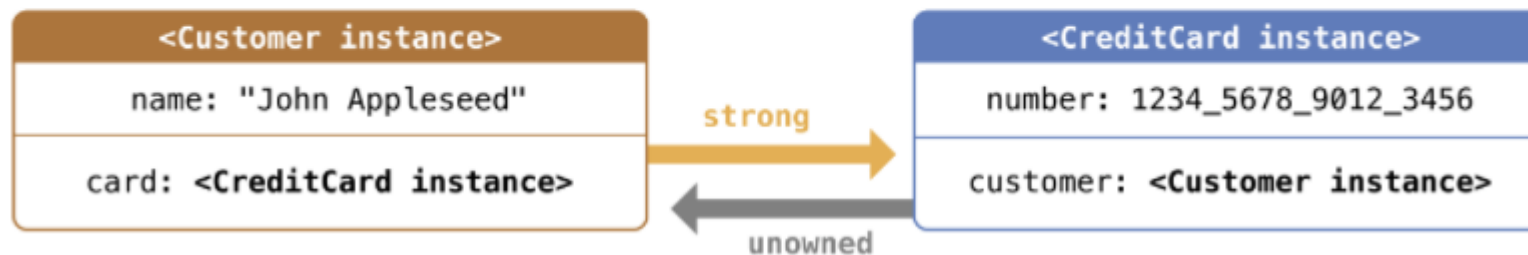


https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

Speicherverwaltung – „Unowned“ Referenzen

```
john = nil
```

```
var john
```



Beide Objekte
werden deallokiert.

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

Speicherverwaltung - Zusammenfassung

- Die Speicherverwaltung von Referenztypen (z.B. bei Klassen) basiert auf Referenzzählung.
- Zirkuläre Referenzen führen zu Speicherlöchern.
- Lösung: Schwache Referenzen
- Zwei Typen von schwachen Referenzen:
 - **Weak:** Keine Instanz muss Anforderungen an die Lebenszeit erfüllen. Beide können `nil` annehmen.
 - **Unowned:** Die Instanz, auf die ein anderes Objekt per „unowned“ Referenz referenziert muss länger leben.
- Wenn deren Anwendung möglich ist, sind „unowned“ Referenzen leichter zu benutzen, da kein `Optional` verwendet wird und damit der `nil`-Test wegfällt.
- Mehr Details:
https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

Eigene Datentypen – Weiterführende Konzepte

- **Letztes Einheit:** Basics zu Strukturen, Klassen und Enumerationen
- **Dieses Einheit:** Weiterführende Konzepte
 - Properties
 - Vererbung
 - Subscripts
 - Extensions

Properties

Swift kennt verschiedene Arten von Properties:

- (Variable|Constant) Stored Properties
- Lazy Stored Properties
- (Read-only) Computed Properties
- Type Properties

Bisher bereits betrachtet: Variable|Constant Stored Properties:

```
struct Car {  
    var speed: Int  
    let size: Int  
}
```

Properties – Lazy Stored Properties

Initialisierung des Properties erst beim ersten, lesenden Zugriff (und **nicht** bei Erstellung der Instanz):

```
class TCPStream {  
    func send( msg : String) { /*...*/}  
}  
  
class ChatBot {  
    lazy var tcp = TCPStream()  
  
    func chat() {  
        tcp.send(msg:"random noise.")  
    }  
}
```

Wichtig: Wenn Erstzugriff aus zwei verschiedenen Threads erfolgen kann, ist nicht garantiert, dass das Property nur einmal initialisiert wird.

Wichtig: Lazy Properties müssen immer als Variablen deklariert werden, weil sie erst bei Erstzugriff initialisiert werden und für Konstanten gilt, dass sie bereits vor der Instanz-Initialisierung bereits initialisiert sein müssen.

Properties – Computed Properties

Speichert nichts direkt, sondern nutzt **Getter/Setter-Syntax** für den Zugriff auf andere Properties:

```
struct Car {  
    let tireAirPressure = 2.0  
    var tireWear = 2.0  
    var maxSpeed : Double {  
        get {  
            return tireWear / tireAirPressure;  
        }  
        set(newMaxSpeed) {  
            tireWear = newMaxSpeed * tireAirPressure;  
        }  
    }  
}
```

Wichtig: Computed Properties müssen immer als Variable deklariert werden (auch wenn sie read-only sind, also keinen Setter haben), weil ihr Wert nicht zur Initialisierung feststeht.

Ohne Setter => **Read-only** Computed Property.

Properties – Syntaktische Abkürzungen

Für Computed Properties gibt es **syntaktische „Abkürzungen“**, die einem Schreibe- und Lesearbeit abnehmen:

```
struct Car {  
    /*...*/  
    var maxSpeed : Double {  
        get {  
            return tireWear / tireAirPressure;  
        }  
        set {  
            tireWear = newValue * tireAirPressure;  
        }  
    }  
}
```

Für Read-only Computed Properties:

```
struct Car {  
    /*...*/  
    var maxSpeed : Double {  
        return tireWear / tireAirPressure;  
    }  
}
```

Properties – Type Properties

Properties können auch auf Typebene (Strukturen, Klassen, Enumerationen) definiert werden.

```
struct Structure {  
    static var storedTypeProperty = "Some value."  
    static var readOnlyStoredTypeProperty: Int {  
        return 1  
    }  
}  
  
print(Structure.storedTypeProperty)
```

Wichtig: Stored Type Properties werden immer „lazy“ initialisiert und es wird garantiert immer nur einmal initialisiert (auch wenn der Zugriff aus unterschiedlichen Thread erfolgt != Lazy Stored Properties).

Wichtig: Stored Type Properties müssen immer mit einem Standardwert initialisiert werden, denn es könnte bei Instanzinitialisierung bereits zu spät sein.

Properties – Property Observers

Zur **Überwachung von Properties** bieten sich sog. „Property Observers“ an:

```
var propertyA: Int {  
    didSet(newValue) { // propertyA hält alten Wert.  
        print(„New Value \$(newValue). Old Value: \$(propertyA)“)  
    }  
    didSet(oldValue) { // propertyA hält neuen Wert.  
        print(„New Value \$(propertyA). Old Value: \$(oldValue)“)  
    }  
}
```

Wichtig: Weder `willSet` noch `didSet` wird bei der Initialisierung von Properties aufgerufen.

Vererbung

- Vererbung in Swift ist nur für Klassen vorgesehen und dann auch nur einfach (!= C++, == Java,C#).
- Es gibt keine universale Basisklasse (== C++, != Java,C#).

```
class Car {  
    var speed : Double  
    var pos : Point  
    func move() { /*...*/ }  
}  
  
class FlyingCar : Car {  
    override func move() { /*...*/ }  
}  
  
var car = FlyingCar()  
car.move()
```

- **override** kann bei Methoden, Init-Funktionen, Computed Properties und Subscripts angewandt werden (**nicht** bei Stored Properties).
- **override** bei Computed Properties: Basisklassen Code wird weiterhin ausgeführt.
- **override** bei Methoden und Init-Funktionen: Basisklassen-Code wird nur bei der Verwendung von `super.init()` ausgeführt (Ausnahme: Parameterlose Init-Funktionen).

Einschub: Designated und Convenience Init

- Unterscheidung: Vollwertige Init-Funktionen (**Designated**) und **Convenience** Init-Funktionen.

```
class A {  
    var a : Int  
    var b : String  
  
    init(a: Int, b: String) {  
        self.a = a  
        self.b = b  
    }  
  
    convenience init(a : Int) {  
        self.init(a: a, b: „b“)  
    }  
}
```

Regeln für **Convenience** Init-Funktionen:

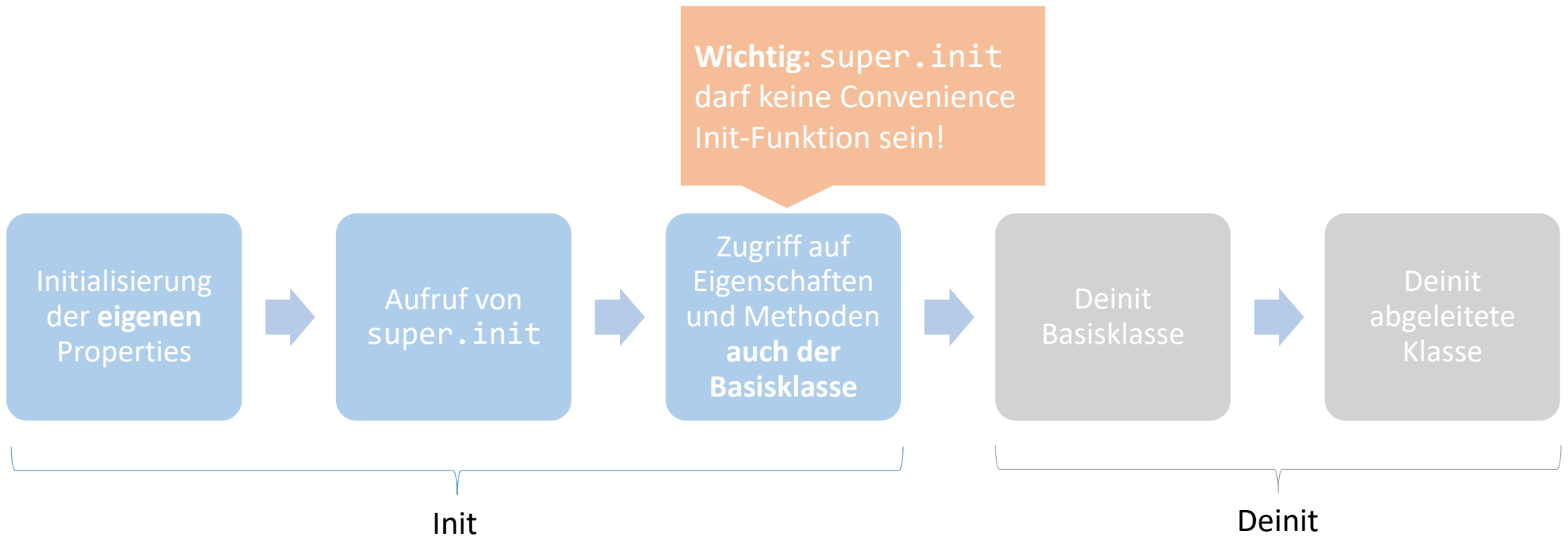
- Es dürfen keine Eigenschaften initialisiert werden.
- Andere Konstruktoren werden mit `self.init` aufgerufen.
- Keine Init-Funktionen der Basisklasse dürfen aufgerufen werden.

Wann verwenden?

- Wenn init Methoden mit langen Parameterlisten existieren, die je nach use case auch mit Default-Parametern belegt werden könnten.

Vererbung – Lebenszyklus

Ziel: Alle Eigenschaften der kompletten Vererbungshierarchie müssen initialisiert werden.



Vererbung – Weitere Regeln

- Hat eine abgeleitete Klasse keine Init-Funktion, erbt sie alle Init-Funktionen der Basisklasse.
- ...
- Mehr Details:
https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html

Vererbung – final und required

- Mit dem `final` Schlüsselwort lassen sich Klassen, Methoden und Computed Properties definieren, die nicht überschrieben werden können:

```
final class A {...}  
class B {  
    final func foo() {...}  
}
```

Hintergrund: Das `final` Schlüsselwort lässt Compiler-Optimierungen zu, da der Methodenaufruf zur Compile-Zeit und nicht erst zur Laufzeit aufgelöst werden kann.

- Mit dem Schlüsselwort `required` können Init-Funktionen in der Basisklasse markiert werden, die von abgeleiteten Klassen implementiert werden **müssen**.
- Die Bedingung ist ebenfalls erfüllt, wenn die abgeleitete Klasse keine Init-Funktion implementiert. Dann werden die Init-Funktionen der Basisklasse **automatisch vererbt** (Siehe Folie 25).

Vererbung – Polymorphie und Downcasts

Polymorphie:

```
class Base {
    func write() {print(„Base“)}
}

class Derived : Base {
    var a = 3
    override func write()
    {print(„Derived“)}
}

var objects = [Base(), Derived()]
for obj in objects {
    obj.write()
}
```

Downcasting mit **as!** Und **as?**:

```
var objects = [Base(), Derived()]
for obj in objects {

    if obj is Derived {
        let castedObj = obj as! Derived
        print(castedObj.a)
    }

    if let castedObj = obj as? Derived {
        print(castedObj.a)
    }
}
```

Subscripts

- Individueller Indexzugriffsoperator (vgl. C++: []-Operator Überladung)
- Einsatz in der Standardbibliothek: **Type Collections**

```
struct CustomIntArray {  
    var internalArray : [Int]  
  
    subscript( index : Int) -> Int {  
        get { return internalArray[index] }  
        set { internalArray[index] = newValue}  
    }  
}
```

- Die Definition mit Subscripts mit **mehr als einem Parameter** ist ebenfalls möglich.

Extensions

- Erweiterung existierender **Klassen**, Strukturen, Enumerationen und **Protokolle**
- Was kann man hinzufügen:
 - Computed Properties
 - **Methoden**
 - **Init-Funktionen**
 - Subscripts
 - **Protokolle**
- Funktioniert auch für **generische Typen** (später mehr).

Beispiel:

```
class A { /*...*/ }

extension A {
    func foo(b : Int) { /*...*/ }
}

extension A {
    init(b : Int) { /*...*/ }
}

extension A : P1 {
    /*Implementierung von P1*/
}
```


Extensions - Protokolle

- Protokollerweiterungen müssen mit einer **Standardimplementierung** versehen werden, um kompatibel mit vorhandenem Code zu bleiben, die aber überschrieben werden kann:

```
protocol P1 {  
    func foo() -> Int  
}  
  
extension P1 {  
    func bar() -> Double {  
        return 3.4  
    }  
}
```

- Es ist zudem möglich, selektiv festzulegen, für welche Typen Erweiterungen gelten sollen:

```
extension P1 where Self : P2
```

Funktionale Programmierung

In Swift sind Funktionen „**first class citizen**“:

Wichtig: Funktionen sind Referenztypen.

- Funktionen als Datentyp

```
var f : (Double, Int) -> (String, Bool)

func foo(a : Double, b : Int) -> (String, Bool) {
    return ("1", false)
}

f = foo
```

- Funktionen als Parameter & Rückgabewert

```
func bar (f: (String) -> Double) -> ((Double) -> Double) {
    /*...*/
}
```

Funktionale Programmierung & Closures

- Closures sind **anonyme Funktionen** (Java, C#, C++: Lambda-Ausdrücke).
- Sie reduzieren den Schreibaufwand, der mit der klassischen func Schreibweise anfällt:

```
let data = [1.0,2.0,3.0,4.0,5.0]

func transform(v: Double) {
    return v * -1.0
}
var res1 = data.map(transform))

var res2 = data.map( { (v: Double) -> Double in return v * -1.0 } )
```

Closures – Verkürzte Schreibweisen

Es gibt einige Möglichkeiten, Closures verkürzt darzustellen:

```
var res2 = data.map( { (v: Double) -> Double in return v * -1.0 } ) //Standard.  
var res3 = data.map( { (v: Double) -> Double in v * -1.0 } ) //Kein return mehr.  
var res4 = data.map( { (v) -> Double in v * -1.0 } ) //Parametertyp wird inferiert.  
var res5 = data.map( { $0 * -1.0 } ) //Returntyp wird inferiert. Namenloser Parameter.  
var res6 = data.map() { $0 * -1.0 } //Reduzierte Verschachtelung. „Trailing Closure“.  
var res7 = data.map(-) //Direkte Verwendung des - Operators.
```

Closures – Capturing Values

Closures haben Zugriff auf Variablen und Konstanten, die in ihrem Scope liegen:

```
func createIncrementor(delta: Int) -> ()->Int {  
    var total = 0  
    return {  
        total += delta  
        return total  
    }  
}  
  
let f1 = createIncrementor(delta: 1)  
let f2 = createIncrementor(delta: 2)  
  
for i in 1...10 {  
    print(„\((f1())) \((f2()))“)  
}
```

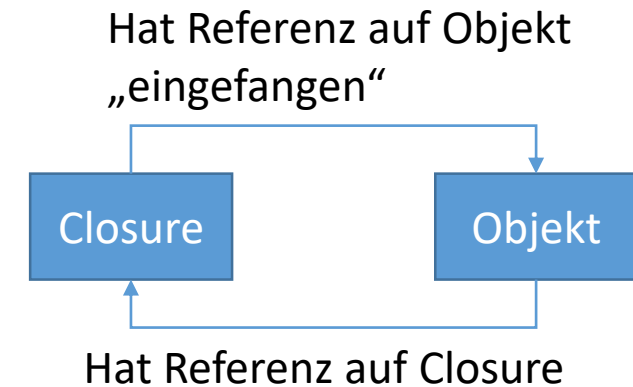
Hintergrund: Closures „fangen“ Variablen und Konstanten: Jede Closure-Instanz hat eine Kopie der zugegriffenen Werttypen und eine Referenz auf die zugegriffenen Referenztypen.

Closures – Capture List

Problem: Referenzzyklen => Speicherloch

Scenario:

```
class A {  
    var closure : (() -> Void)?  
  
    func runClosure() {  
        closure = {  
            print("That is me: \(self)")  
        }  
        closure!()  
    }  
}  
  
var a: A? = A()  
a?.runClosure()  
a = nil
```



Lösung: Capture List:

```
func runClosure() {  
    closure = { [weak self] in  
        print("That is me: \(self)")  
    }  
    closure!()  
}
```

Generics

- **Ziel:** Algorithmen und Datenstrukturen mit Typen parametrisieren.
- **Arten:**
 - Generische Funktionen
 - Generische Typen
 - (Generische Protokolle)
- Vergleichbare Konzepte in anderen Sprachen:
 - Generics in Java und C#
 - Templates in C++

Generics – Generische Funktionen

```
func swap(_a: inout Int, _b: inout Int) {  
    func swap(_a: inout Double, _b: inout Double) {  
        func swap (_a: inout Bool, _b: inout Bool) {  
            let temporaryA = a  
            a = b  
            b = temporaryA  
        }  
    }  
}
```



```
func swap<T>(_ a: inout T, _ b: inout T) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```



```
var a = 3  
var b = 107  
swap(&a, &b)  
var c = "1"  
var d = "2"  
swap(&c, &d)
```


Generics – Generische Datentypen

```
struct IntStack {  
    struct DoubleStack {  
        struct BoolStack {  
            var items = [Bool]()  
            mutating func push(_ item: Bool) {  
                items.append(item)  
            }  
            mutating func pop() -> Bool {  
                return items.removeLast()  
            }  
        }  
    }  
}
```



```
struct Stack<Element> {  
    var items = [Element]()  
    mutating func push(_ item: Element) {  
        items.append(item)  
    }  
    mutating func pop() -> Element {  
        return items.removeLast()  
    }  
}
```



```
var stack = Stack<String>()  
Stack.push("1")
```

Generics – Typbedingungen (Type Constraints)

Problem: == Operator ist nicht für jeden Typ definiert.

```
func findIndex<T>(of toFind: T, in array:[T]) -> Int? {  
    for (index, value) in array.enumerated() {  
        if value == valueToFind {  
            return index  
        }  
    }  
    return nil  
}
```

Protokolle
definieren
geforderte
„Fähigkeiten“



```
func findIndex<T: Equatable>(of toFind: T, in array:[T]) -> Int? {  
    for (index, value) in array.enumerated() {  
        if value == valueToFind {  
            return index  
        }  
    }  
    return nil  
}
```

Generics – Erweiterungen und Typbedingungen

- Typebedingungen können auch für die Erweiterung generischer Typen verwendet werden.
- Typbedingungen können dabei ebenfalls benutzt werden:

```
extension Array: TextRepresentable where Element: TextRepresentable {  
  
    var textualDescription: String {  
        let itemsAsText = self.map { $0.textualDescription }  
        return "[" + itemsAsText.joined(separator: ", ") + "]"  
    }  
  
}
```

Generics – Generische Protokolle

- `protocol GenericProtocol<T> {...}` ist nicht möglich.
- **Alternative:** Generische Protokolle über sog. **Protocol Associated Types (PAT)** definieren:

```
protocol GenericProtocol {  
    associatedtype Type  
    func foo() -> Type  
}  
  
struct SuperStruct<T> : GenericProtocol {  
    typealias Type = T  
  
    var bar : T  
    func foo() -> T {  
        return bar  
    }  
}
```

Hier wird der
Typparameter T mit
dem assoziierten Typ
Type verknüpft.

Standardbibliothek – Ausgewählte Inhalte

- Standardprotokolle:
 - Hashable, ErrorProtocol (schon gesehen)
 - Equatable: `protocol { public static func ==(lhs: Self, rhs: Self) -> Bool}`
 - Comparable: Definiert `<=`, `>=`, `<`, `>`
 - Any, AnyObject
 - Serialisierung: `protocol Codable: Encodable, Decodable => Geeignet für JSON De-(serialisierung)`

- zip

```
var a = [1,2,3]
var b = [„1“, „2“, „3“]
Var combinedTuple = zip(a,b)
```

- filter

```
var a = [„a“, „ab“, „abc“]
var b = a.filter() {$0.count > 1}
// b == [„ab“, „abc“]
```

- split/join

```
var a = [1,2,0,3].split() {$0 == 0} // a == [[1,2],[3]]
var b = [„1“, „2“, „3“].joined(separator: „#“) // b == „1#2#3“
```