Initialization, Deinitialization and Access Control

Florian Klammer

Initialization

Initializers

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

Bei Konstanten muss während initialisierung Wert zugeteilt werden danach ist dieser nicht mehr veränderbar

Initialization Parameters

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(_ kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
```

Designated and Convenience Initializers

Designated Initializers initialisiert alle Properties der Klasse und ruft einen passenden superclass initializer um die initialization fortzuführen

Convenience Initializer sind secondary supportende Initializer die einen designated initializer schneller und mehr angenehm zum Arbeiten machen

```
init(param1, param2, param3, ... , paramN) {
    // code
}
// can call this initializer and only enter one parameter,
// set the rest as defaults
convenience init(myParamN) {
    self.init(defaultParam1, defaultParam2, defaultParam3, ... , myParamN)
}
```

Initializer Delegation for Class Types

Rule 1

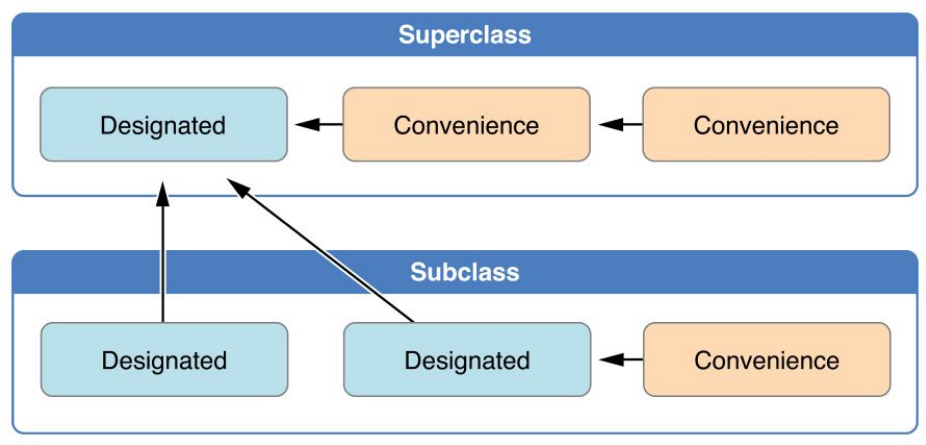
A designated initializer must call a designated initializer from its immediate superclass.

Rule 2

A convenience initializer must call another initializer from the *same* class.

Rule 3

A convenience initializer must ultimately call a designated initializer.



Two-Phase Initialization

Klassen initialization in 2 Phasen unterteilt

-> In der ersten Phase wird jeder stored Property ein Anfangswert zugeteilt

-> In der zweiten Phase wird jeder Klasse die Möglichkeit gegeben die properties zu verändern

Swift Safety Check

Safety check 1

A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer.

Safety check 2

A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn’t, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

Swift Safety Check 2

Safety check 3

A convenience initializer must delegate to another initializer before assigning a value to *any* property (including properties defined by the same class). If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer.

Safety check 4

An initializer cannot call any instance methods, read the values of any instance properties, or refer to `self` as a value until after the first phase of initialization is complete.

Initializer Inheritance and Overriding

Swift Sub-Klassen erben nicht automatisch super Klassen Initializers

Bei Überschreibung eines Superclass designated initializer in einer Sub-Klasse wird Override benötigt

Dies gilt auch für Default Initializer

```
class Vehicle {
    var numberOfWheels = 0
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}
class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}
```

Automatic Initializer Inheritance Rules

Rule 1

If your subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

Rule 2

If your subclass provides an implementation of *all* of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

Failable Initializers

Es ist möglich Initializers zu schreiben die versagen können

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}
```

Required Initializers

Mit dem required modifier bevor eines Klassen Initializer wird angegeben das jede Subklasse diesen implementieren muss (required wird auch in jederm sub-klassen initializer benötigt

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}
```

```
class SomeSubclass: SomeClass {
    required init() {
        // subclass implementation of the required
initializer goes here
    }
}
```

Deinitialization

Deinitializers

Werden vor deallocation automatisch aufgerufen

Deinitializers können nicht vom Code aufgerufen werden

```
var counter = 0; // for reference counting
class baseclass {
   init() {
      counter = 1
   }
   deinit {
      counter = 0;
   }
}
var print: baseclass? = baseclass()

print(counter) //Hier 1
print = nil
print(counter) //Hier 0
```

Access Control

Access Control

Schränkt teile Codes von anderen Code in anderen Source Files oder Modules ein

A *module* is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift's `import` keyword.

Access Levels

Open access and *public access -> Source File from defining Module, Source File from other module that imports defining module*

Internal Access -> Used from any Source File from defining Module

File-private Access -> Use only in defining source file

private Access -> restricts use of an entity to enclosing declaration

Difference Open and Public

- Classes with public access, or any more restrictive access level, can be subclassed only within the module where they’re defined.
- Class members with public access, or any more restrictive access level, can be overridden by subclasses only within the module where they’re defined.
- Open classes can be subclassed within the module where they’re defined, and within any module that imports the module where they’re defined.
- Open class members can be overridden by subclasses within the module where they’re defined, and within any module that imports the module where they’re defined.

Default Access Level ist internal

```
public class SomePublicClass {}

internal class SomeInternalClass {}

fileprivate class SomeFilePrivateClass {}

private class SomePrivateClass {}
```

Custom Types

A public type defaults to having internal members, not public members. If you want a type member to be public, you must explicitly mark it as such

Tuples benutzen immer den am meist einschränkenden Access Level

Bei Funktionen ist es wichtig die Funktion denselben Access Level zu geben wie den niedrigsten Access Level Parameter

Bei Enumerations erhalten die internen Cases automatisch den gleichen wie der angegebene des Enums

Eine Variable/Konstante/Property/Getter/Setter/Initializer kann kein höheres Access Level als der zugewiesene Typ erhalten

Subclassing

Eine Subklasse kann keinen höheren Access Level als die Super Klasse erhalten

Jedoch kann man Class members mithilfe Override höheren Access geben

```
public class A {

    fileprivate func someMethod() {}
}
internal class B: A {
    override internal func someMethod() {}
```

Beispiel

Erstelle eine internal Klasse mit min. 3 Variablen und 2 Initializers eine designated und eine convenience

Erstelle ein deinit das die Deallocation des Objektes ausgibt sobald es ausgeführt wird

Erstelle eine file-private Subklasse mit 2 neuen Variablen + failable Initializer

Erstelle mehrere Objekte und gib diese aus um sie zu testen