Swift: Classes, Structures, Enumerations

Dr.-Ing. Thomas Springer
M.Sc. Martin Weißbach
Optionals

Schrödinger’s Cat in terms of Values
• used for situations where the value of a property, the return value of a function etc. may be absent (nil)
• an Optional states
  • that there is value that equals x OR
  • that there is no value at all
• var surveyAnswer: String?
  • set to nil automatically
  • if that would be a property, it would not have been explicitly given a value during initialization

• static func toInt(string: String) -> Int?
  • returns either nil or the integer representation of the string
Accessing Optional Values

- **1\textsuperscript{st} If:** *forced unwrapping*
  - exclamation mark added to optional explicitly unwraps the optional
  - causes runtime error if the optional’s value is nil

- **2\textsuperscript{nd} If:** *optional binding*
  - bind the optional’s value to a temporary variable to work with, if the optional’s value is not nil
Accessing Optional Values

- **1st If: forced unwrapping**
  - exclamation mark added to optional explicitly unwraps the optional
  - causes runtime error if the optional’s value is nil

- **2nd If: optional binding**
  - bind the optional’s value to a temporary variable to work with, if the optional’s value is not nil

```swift
var optionalNumber: Int?
optionalNumber = Int("1989")

if optionalNumber != nil {
    print("The optional number is \(optionalNumber!)")
}

if let number = optionalNumber {
    print("The optional number is \(number)")
}
```
Exclamation mark required to explicitly unwrap the optional value

```swift
let optionalInteger : Int? = 1234
let forcedInteger : Int = optionalInteger!
```

```swift
let implicitInteger : Int! = 1234
let integer = implicitInteger
```

No exclamation mark required, because it is already implicitly unwrapped
Implicitly Unwrapped Optionals

- behave like Optionals, but do not need to be explicitly unwrapped when the underlying value is read

- used when the Optional can never be `nil` anytime again during runtime based on the program structure (according to Apple’s documentation [1])

- a runtime error will be raised, when an implicitly unwrapped Optional will be read when the value is `nil`
Optional Chaining

- used for querying or calling properties, methods or subscripts of an optional that might be nil
- if the optional is nil the property, method or subscript returns nil
- multiple queries or calls can be chained and the chain fails gracefully (without application crash) if one link returns nil

```swift
var optionalNumber = Int(“2016”)
optionalNumber?.distance(to: 2015)
```
Properties

values for classes, structures, or enumerations
• variable stored properties (var keyword)
• constant stored properties (let keyword)
• set a default value for either of them as part of the definition
• modification of initial value during initialization
  → this is also true for constant stored properties
• value types assigned to a constant stored property are completely immutable
  • it is not possible to change the properties of the assigned value type
Lazy Stored Properties

• value not calculated until it is used for the first time
• must not be declared as constant
  • a constant property’s value must always be known before initialization completes
• a lazy stored properties might not be evaluated when initialization finishes
• useful when complex computation is required to evaluate the property or when value depends on outside factors that might not be known during initialization
Computed Properties

• do not store a certain value
• used to set and get other properties indirectly
• must not be declared as constants because their value is inherently not fixed
Type Properties

- Previously, only instance properties were shown.
  - Their values are always associated with a single instance of a type.
- Type properties belong to the type instead of the instance.
  - At runtime, there will be only one copy of that property independent of the number of created instances.
- Value types allow for stored and computed type properties.
- Reference types allow for computed type properties only.
Type Properties

```swift
struct ExampleStruct {
    let storedProperty : Int

    static let storedStaticImmutableProperty : Int = 1
    static var storedStaticMutableProperty : Int = 0

    static var computedProperty : Int {
        get {
            return self.storedStaticImmutableProperty + 3
        }
        set(newValue) {
            self.storedStaticMutableProperty = newValue
        }
    }
}
```

```swift
class ExampleClass {
    let storedProperty = 0

    class var computedProperty : Int {
        get {
            return 42
        }
    }
}
```
Instance Initialization
• Initializers are used to create new instance of a particular type
• Initializers do not have a return value
• when initialization is completed, all stored properties of a type must be initialized with a sound value
• initializers are automatically generated, based on the manifestation of the stored properties

```c
init() {
    // Initialization Task performed here
}
```
struct GeometriePoint {
    var x = 0, y = 0, z = 0
}

• Default Initializer
    init() {} 

• Memberwise Initializer
    init(x: Int, y: Int, z: Int)
    {
        self.x = x; self.y = y; self.z = z
    }

• Custom Initializer
    init(withPlanePoint: Point)
    {
        x = point.x; y = point.y
    }
<table>
<thead>
<tr>
<th>Given</th>
<th>Default Initializer</th>
<th>Memberwise Initializer</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable Stored Properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with default value</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>no default value</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>immutable Stored Properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with default value</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>no default value</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Custom Initializer</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Legend: ✔️ True, ✗ False
• have other initializers to perform initialization tasks to avoid code duplication
• because value types do not support inheritance, structures and enumerations can only call initializers they provide themselves
• use `self.<initializer>` to delegate the initialization task
• remember, that delegation to default or memberwise initializer is not possible from custom initializer
Instance Initialization
A closer look on Classes
• stored properties must be assigned an initial value during initialization
  • that includes inherited properties, too

• own properties must be initialized before inherited properties

• designated and convenience initializers available
Designated Initializers

- primary initializers of a class
- fully initialize the properties provided by the class
  - forwards initialization of inherited properties to one of the superclass’ designated initializer
- each class must have at least one designated initializer
  - inherit the designated initializers from the superclass
    might be sufficient

```cpp
init(/*parameters*/) {
    // Initialization
}
```
Convenience Initializers

- secondary, supporting initializers
- can only forward initialization to designated initializers of the same class as the convenience initializer
- used to provide shortcut or use-case specific initialization processes of a class

```cpp
convenience init(/*parameters*/) {
    // Initialization
}
```
• 3 rules apply:
  • designated initializers must call designated initializers from their immediate superclass
  • convenience initializers must call another initializer from the same class
  • convenience initializers must ultimately call a designated initializer
Two-Phase Initialization

• Phase 1:
  • stored properties are assigned initial values by the class that introduces it

• Phase 2:
  • commences as soon as all properties are in their default configuration
  • each class can modify the default values of their stored properties
• Swift’s compiler performs sanity checks to ensure two-phase initialization performs without error
• Check 1:
  • all introduced properties of a class must be initialized before initialization is delegated to the superclass’ initializer

• Check 2:
  • designated initializer must forward to the superclass’ designated initializer before values of inherited properties can be changed
• Check 3:
  • convenience initializers must delegate the process to designated initializers before values of any properties (inherited or self) can be modified
• Check 4:
  • initializers cannot call instance methods, read values of instance properties or refer to `self` as a value until the first phase of the Two-Phase Initialization is completed
Two Phase Initialization: Overview

Phase 1

Root Class

- Memory Uninitialized
- ... → Memory Initialized
- self now usable → Properties Customized

Immediate Superclass

- Memory Uninitialized
- ... → Memory Initialized
- designated initializer hands off to superclass

Class

- Memory Allocated
- designated initializer confirms that all stored properties have a value

Phase 2

- Memory Initialized
- ... → Properties Customized
- modify properties

Instance

- Properties Customized
- modify properties
- ready for usage
• Initializers are not bequeathed to subclasses by default
• inheriting initializers is still possible under certain conditions:
  1. if subclass defines no own designated initializers, all the superclass’ initializers are inherited
  2. if subclass provides all designated initializers of the superclass, it inherits *all* convenience initializers of the superclass
• can be achieved through 1 or by providing a custom implementation according to definition of the initializer