App-Entwicklung für iOS und OS X

SS 2016

Stephan Gimbel

Datentypen & Optionals

Variablen und Konstanten

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

Dabei steht let für eine Konstante und var für eine Variable
Swift ist Type Safe, der Typ einer Variablen/Konstante wird durch Type Inference ermittelt.

```
var welcomeMessage:String
```

Der Typ kann explizit festgelegt werden.

```
let π = 3.14159
let 你好 = “你好世界"
let 🐶🐮 = "dogcow"
```

Auch Unicode.

Datentypen & Optionals

```
print(friendlyWelcome)
print("The current value of friendlyWelcome is \(friendlyWelcome)")
```

Die Ausgabe von Variablen und Literalen in Swift unterstützt String Interpolation.

```
typealias AudioSample = UInt16
var maxAmplitudeFound = AudioSample.min
```

Type Alias als alternativer Name für einen existierenden Typ

Datentypen & Optionals

Tuples

```
let http404Error = (404, "Not Found")
```

Tuples mit beliebigem Typ, z.B. (Int, Int, Int) oder (String, Bool)

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// prints "The status code is 404"
print("The status message is \(statusMessage)")
// prints "The status message is Not Found"
```

Decomposition von Tuples

Datentypen & Optionals

```
let http404Error = (404, "Not Found”)

let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)”)
// prints "The status code is 404"
```

Teile des Tuples können mittels Underscope (_) ignoriert werden.

```
print("The status code is \(http404Error.0)”)
// prints "The status code is 404”
print("The status message is \(http404Error.1)”)
// prints "The status message is Not Found"
```

Alternativ kann über den Index auf die Elemente zugegriffen werden…

```
let http200Status = (statusCode: 200, description: “OK")

print("The status code is \(http200Status.statusCode)”)
// prints "The status code is 200”
print("The status message is \(http200Status.description)”)
// prints "The status message is OK"
```

…oder durch den Names des Elementes.

Datentypen & Optionals

Optionals

Manchmal hat eine Variable keinen Wert (nil - vgl. Null in C++, Java, etc.).
In einem solchen Fall ist eine Variable ein Optional.

```
let possibleNumber = “123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?", or "optional Int"
```

Optionals werden nach dem Typ mit einem ? versehen, z.B. `Int?` (nicht `Int`). Der Initializer kann u.U. fehlschlagen.

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

Optionals können auf nil gesetzt werden. Dies ist auch der Default:

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

Datentypen & Optionals

Beim Zugriff (z.B. Ausgabe) auf ein Optional, sollte sichergestellt sein, dass dies auch ein Value hat. Ansonsten droht ein Crash!

```
if convertedNumber != nil {
   print("convertedNumber has an integer value of \(convertedNumber!).”)
}
// prints "convertedNumber has an integer value of 123."
```

Dies ist ein forced unwrapping des Wertes eines Optionals.

Der Wert eines Optionals kann temporär über eine Variable/Konstante verfügbar gemacht werden. Dies geschieht per optional binding.

```
if let constantName = someOptional {
   someOptional
}
```

Datentypen & Optionals

Beispiel:

```
if let actualNumber = Int(possibleNumber) {
   print("\'\(possibleNumber)\' has an integer value of \(actualNumber)”)
} else {
   print("\'\(possibleNumber)\' could not be converted to an integer”)
}
// prints "'123' has an integer value of 123"
```

```
if let firstNumber = Int("4"), secondNumber = Int("42") where firstNumber <
   secondNumber {
      print("\(firstNumber) < \(secondNumber)”)
}
// prints "4 < 42"
```

Funktioniert auch mit multiple optional bindings.

Datentypen & Optionals

Wenn durch die Programmstruktur klar ist, dass ein Optional einen Wert hat (nach der ersten Zuweisung), dann kann auf auf Check und Unwrapping bei jedem Zugriff verzichtet werden.

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation mark

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation mark
```

Dies wird meist bei Klassen Initialisierung verwendet und ist als implicitly unwrapped optionals bekannt und können wie bisher verwendet werden.

```
if assumedString != nil {
    print(assumedString)
} // Prints "An implicitly unwrapped optional string."

if let definiteString = assumedString {
    print(definiteString)
} // Prints "An implicitly unwrapped optional string."
```

Der Zugriff auf ein implicitly unwrapped optional dessen Wert nil ist, führt zu einem Runtime Error!

Error Handling & Assertions

Wenn ein Fehler in einer Funktion auftritt (Funktionen schauen wir uns gleich an), kann dieser Abgefangen und durch den Caller behandelt werden.

```
func canThrowAnError() throws {
    // this function may or may not throw an error
}
```

Um einen Error abzufangen wird ein new containing scope erstellt.

```
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

Error Handling & Assertions

Fehler können ebenfalls nach Typ unterschieden werden.

```
do {
    try makeASandwich()
    eatASandwich()
} catch Error.OutOfCleanDishes {
    washDishes()
} catch Error.MissingIngredients(let ingredients) {
    buyGroceries(ingredients)
}
```

Wir schauen uns Fehlerbehandlung später noch etwas detaillierter an.
Für die Neugierigen: Swift Programming Language: Error Handling

Error Handling & Assertions

Wenn eine bestimmte Bedingung true sein muss, damit das Programm weiter ausgeführt werden kann, können Assertions verwendet werden um dies sicherzustellen.

```
let age = -3
assert(age >= 0, "A person's age cannot be less than zero")
// this causes the assertion to trigger, because age is not >= 0
```

Schlägt die Bedingung fehl, so wird das ausführen unterbrochen und das Programm kann debugged werden.

Beispiel für Anwendung:

- Invalider Integer Subscript Index wird an Subscript Implementierung übergeben
- Ungültiger Parameter wird an Funktion übergeben
- Optional ist nil, muss aber non-nil sein zur Ausführung von folgendem Code

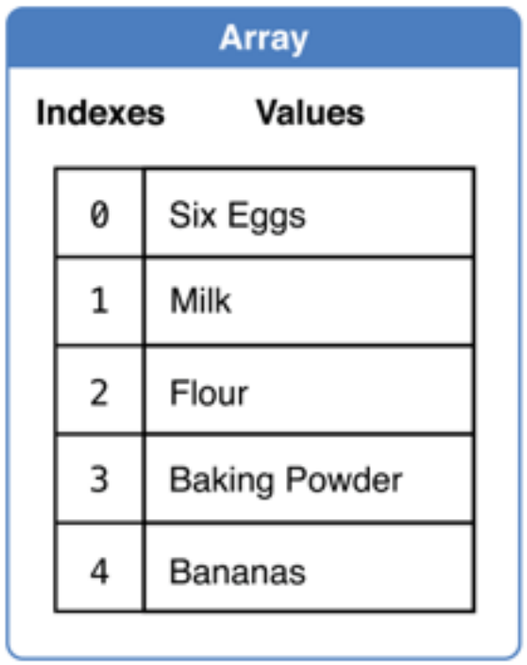
Ähnlich werden Assertions auch beim Unit Testing eingesetzt.

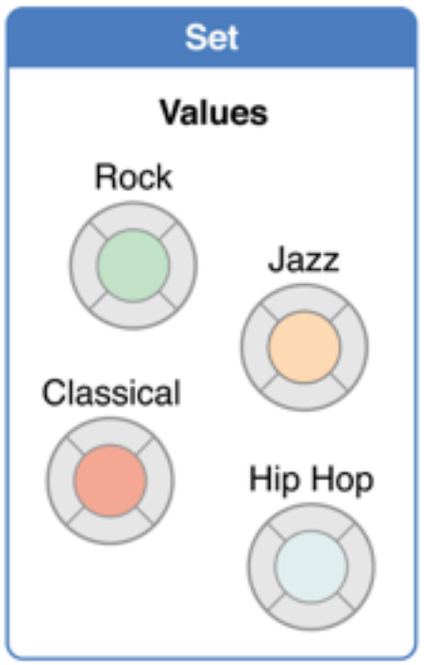
Collection Types

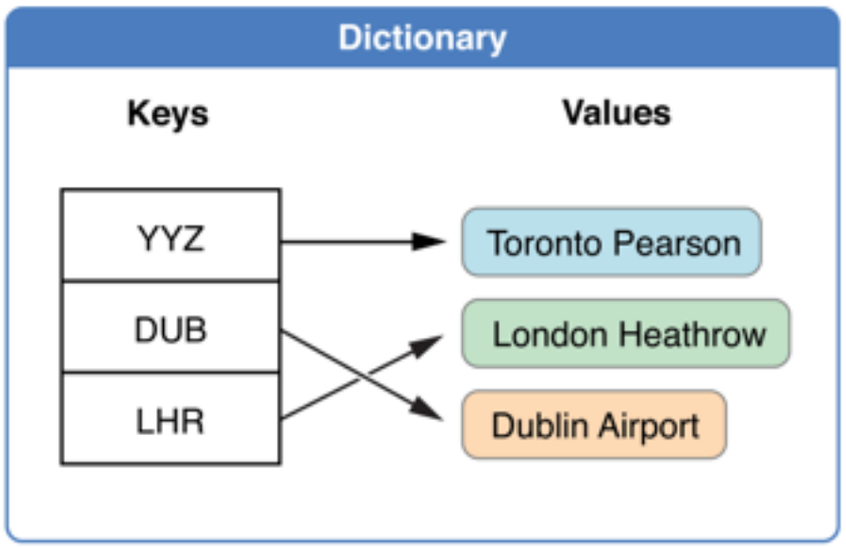
In Swift gibt es drei primäre Collection Types:

- Arrays
- Sets
- Dictionaries

Arrays sind geordnete Collections von Values
Sets sind nicht-geordnete Collections von Values
Dictionaries sind nicht-geordnete Collections von Key-Value Beziehungen







Collection Types

Arrays, Sets und Dictionaries sind als Generic Collections implementiert. Per-Default sind diese Mutable, es können also Elemente hinzugefügt oder gelöscht werden.

Ist sichergestellt, dass die Collection konstant ist, sollte diese als Immutable implementiert werden, so dass der Compiler die Performance optimieren kann.

Beispiele:

Leeres Array

```
var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
// Prints "someInts is of type [Int] with 0 items."
```

```
someInts.append(3)
// someInts now contains 1 value of type Int
someInts = []
// someInts is now an empty array, but is still of type [Int]
```

Collection Types

Default Value

```
var threeDoubles = [Double](count: 3, repeatedValue: 0.0)
// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

Verknüpfen von Arrays durch Additionsoperator +

```
var anotherThreeDoubles = [Double](count: 3, repeatedValue: 2.5)
// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5,
2.5]
```

Erstellen eines Arrays durch Array-Literal

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

Da der Typ der Elemente gleich ist, kann der Typ des Arrays daraus geschlossen werden.

```
var shoppingList = ["Eggs", "Milk"]
```

Collection Types

Formal wird der Typ eines Arrays als `Array<Element>` geschrieben, kann jedoch in der Kurzform `[Element]` geschrieben werden. Funktionell ist beides identisch.

Sollen in einem Array Elemente unterschiedlicher Typen gespeichert werden, so kann dies mittels [AnyObject] realisiert werden.

Zugriff und Modifikation von Arrays

```
print("The shopping list contains \(shoppingList.count) items.")
// Prints "The shopping list contains 2 items."
```

Oder über Shortcut

```
if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
// Prints "The shopping list is not empty."
```

Collection Types

Hinzufügen von Elementen

```
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is making pancakes
```

Alternativ über den Addition-Assignment Operator += (auch für Arrays)

```
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 7 items
```

Zugriff via Index über Subscript Syntax

```
var firstItem = shoppingList[0]
// firstItem is equal to "Eggs"
```

```
shoppingList[0] = "Six eggs"
// the first item in the list is now equal to "Six eggs" rather than "Eggs"
```

Collection Types

Zugriff auf Range

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items
```

Einfügen bzw. Entfernen eines Elementes an einer Index-Stelle

```
shoppingList.insert("Maple Syrup", atIndex: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list
```

```
let mapleSyrup = shoppingList.removeAtIndex(0)
// the item that was at index 0 has just been removed
// shoppingList now contains 6 items, and no Maple Syrup
// the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

Collection Types

Iterieren über Arrays

```
for item in shoppingList {
    print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

```
for (index, value) in shoppingList.enumerate() {
    print("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking Powder
// Item 5: Bananas
```

Iteration mittels der Methode `enumerate()` liefert zusätzlich die Index Position des Elementes.

Collection Types

Die Funktionsweise von Sets und Dictionaries verhält sich ähnlich wie die von Arrays. Da dies aus PAD/ENA schon bekannt sein sollte, verzichten wir auf eine detaillierte Besprechung.

Die entsprechende Dokumentation inkl. Beispielen finden Sie hier:
Swift Programming Language: Collection Types