SWIFT 4.2

01 – THE BASICS

DI Thomas Helml

HTL WELS

IT INFORMATIONSTECHNOLOGIE

INHALT

- Constants and Variables
- Comments
- Semicolons
- Integers
- Floating-Point Numbers
- Type Safety and Type Inference
- Numeric Literals
- Numeric Type Conversion
- Type Aliases
- Booleans
- Optionals

CONSTANTS AND VARIABLES

- Variablen / Konstanten müssen vor Verwendung mit `var` bzw. `let` deklariert werden:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0

var x = 0.0, y = 0.0, z = 0.0
```

TYPE ANNOTATIONS

- Sofern kein Initialwert angegeben wurde, kann ein Typ bei der Deklaration definiert werden

```
var welcomeMessage: String
welcomeMessage = "Hello"

var red, green, blue: Double
```

NAMING CONSTANTS AND VARIABLES

- Für Namen sind alle Unicode-Zeichen erlaubt, ausser:
  - Leerzeichen
  - Ziffern am Beginn

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


var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now „Bonjour!"


let languageName = "Swift"
languageName = "Swift++"
// This is a compile-time error: languageName cannot be changed.
```

PRINTING CONSTANTS AND VARIABLES

- Benutze die globale Funktion `print(_:separator:terminator:)` für die Ausgabe

```
print(friendlyWelcome)
// Prints "Bonjour!"
```

```
print(4,5,6,separator:" ... ", terminator: "!")
// Prints "4 … 5 … 6!“
```

COMMENTS

- Kommentare sind zu verwenden wie in C/Java, allerdings sind sie auch schachtelbar:

```
// This is a comment.

/* This is also a comment
but is written over multiple lines. */
```

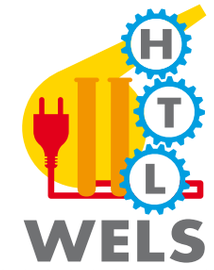
```
/* This is the start of the first multiline comment.
 /* This is the second, nested multiline comment. */
This is the end of the first multiline comment. */
```

SEMICOLONS

- Swift benötigt kein Semikolon am Ende einer Anweisung!
- Semikolons sind notwendig, falls mehrere Statements in einer Zeile stehen

```
let cat = "🐱"; print(cat)
// Prints "🐱"
```

INTEGERS



- In Swift gibt es `signed` und `unsigned Ints` und 8, 16, 32 und 64 Bit Größe
  - `UInt8`: unsigned Integer mit 8 Bit
  - `Int32`: Integer mit 32 Bit

INTEGER BOUNDS

- Die Grenzen von den Integer Werten kann mit `min` bzw. `max` abgefragt werden

```
let minValue = UInt8.min
// minValue is equal to 0, and is of type UInt8
```

```
let maxValue = UInt8.max
// maxValue is equal to 255, and is of type UInt8
```

INT / UINT

- Sofern nicht eine bestimmte Größe benötigt wird, sollte `Int` / `UInt` verwendet werden
  - auf 32-Bit Plattformen hat `Int` / `UInt` die Größe wie `Int32` /`UInt32`
  - auf 64-Bit Plattformen hat `Int` / `UInt` die Größe wie `Int64` /`UInt64`

FLOATING-POINT NUMBERS

- `Double`:
  - 64-bit floating-point Zahl, 15 Stellen Genauigkeit
- `Float`:
  - 32-bit floating-point Zahl, 6 Stellen Genauigkeit

TYPE SAFETY AND TYPE INFERENCE

- Type Safety:
  - Swift ist type-safe, somit muss der Datentyp einer Variablen/Konstante immer feststehen und ist nicht veränderbar

- Type Inference:
  - immer, wenn Literale angegeben werden, folgert Swift auf den Datentyp

TYPE SAFETY AND TYPE INFERENCE

```
let meaningOfLife = 42
// meaningOfLife is inferred to be of type Int


let pi = 3.14159
// pi is inferred to be of type Double
```

```
let anotherPi = 3 + 0.14159
// anotherPi is also inferred to be of type Double
```

NUMERIC LITERALS

- Integer Literale:
  - dezimal: kein Präfix
  - binär: 0b Präfix
  - oktal: 0o Präfix
  - hexadezimal: 0x Präfix

```
let decimalInteger = 17

let binaryInteger = 0b10001       // 17 in binary notation

let octalInteger = 0o21           // 17 in octal notation

let hexadecimalInteger = 0x11     // 17 in hexadecimal notation
```

NUMERIC LITERALS

- Float Literale:
  - dezimal: kein Präfix
  - hexadezimal: `0x` Präfix
  - mit Exponent: `1.25e-2` => 1.25×10^{-2}

```
let decimalDouble = 12.1875

let exponentDouble = 1.21875e1
```

NUMERIC LITERALS

- Um Lesbarkeit zu erhöhen, dürfen vorausgestellte Nullen bzw. Underscores verwendet werden:

```
let paddedDouble = 000123.456

let oneMillion = 1_000_000

let justOverOneMillion = 1_000_000.000_000_1
```

NUMERIC TYPE CONVERSION

- Benutze `Int` als Allzweck Integer für Variablen und Konstanten, selbst wenn sie nicht negativ werden können
- Alle anderen Int-Typen sollten nur verwendet werden, wenn Performance, Speicher oder ander Gründe dafür sprechen
- Wertebereich für `Int` weicht je nach Typ ab, werden diese überschritten kommt es zu einem Compile-Fehler

```
let cannotBeNegative: UInt8 = -1
// UInt8 cannot store negative numbers, and so this will report an error

let tooBig: Int8 = Int8.max + 1
// Int8 cannot store a number larger than its maximum value,
// and so this will also report an error
```

NUMERIC TYPE CONVERSION

- Inkompatible Int-Typen müssen explizit in passende umgewandelt werden:

```
let twoThousand: UInt16 = 2_000

let one: UInt8 = 1

let twoThousandAndOne = twoThousand + UInt16(one)
```

INTEGER AND FLOATING-POINT CONVERSION

- Konvertierungen zwischen Integer und Gleitkommazahlen müssen ebenfalls explizit gemacht werden:

```
let three = 3

let pointOneFourOneFiveNine = 0.14159

let pi = Double(three) + pointOneFourOneFiveNine

// pi equals 3.14159, and is inferred to be of type Double
```

- Floating-Point auf Integer: Komma wird abgeschnitten

```
let integerPi = Int(pi)

// integerPi equals 3, and is inferred to be of type Int
```

TYPE ALIASES

- Ein Type Alias definiert einen alternativen Namen für einen existierenden Type

```
typealias AudioSample = UInt16
```

- Type Aliase können wie der existierende Typ verwendet werden

```
var maxAmplitudeFound = AudioSample.min

// maxAmplitudeFound is now 0
```

BOOLEANS

- Datentyp `bool`

```
let orangesAreOrange = true
let turnipsAreDelicious = false

if turnipsAreDelicious {
    print("Mmm, tasty turnips!")
} else {
    print("Eww, turnips are horrible.")
}
// Prints "Eww, turnips are horrible."
```

TUPLES

- Tupel gruppieren mehrere Werte (auch mit unterschiedlichem Typ) in einen zusammengesetzten Wert
- Verwendung z.B. als Rückgabewert von Funktionen

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found“)
```

- Tupel zerlegen:

```
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"
```

TUPELS

- Werden nur einzelne Werte aus einem Tupel gebraucht:

```
let (justTheStatusCode, _) = http404Error

print("The status code is \(justTheStatusCode)")
// Prints "The status code is 404“
```

- oder:

```
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"
```

TUPLES

- Einzelne Elemente eines Tupels können benannt werden:

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

- Zugriff auf Elemente erfolgt dann über den Namen:

```
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"
```

OPTIONALS

- Options werden verwendet, wenn ein Wert fehlen kann.
- Ein Optional repräsentiert 2 Möglichkeiten:
  - ein Wert ist nicht vorhanden (= `nil`)
  - man kann das Optional "unwrappen" und erhält den Wert

```
let possibleNumber = „123"

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

OPTIONALS – NIL

- `nil`
  - bedeutet Wert fehlt
  - kann nur Optionals zugewiesen werden!

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

- Optional Variable ohne Defaultwert: automatisch `nil`

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

IF STATEMENTS AND FORCED UNWRAPPING

- `if` Statement kann verwendet werden, um ein Optional gegen `nil` zu prüfen

```
if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
}
// Prints "convertedNumber contains some integer value.“
```

- Wenn man sicher ist, dass das Optional einen Wert beinhaltet, dann kann es mit dem `!`-Operator ein *forced unwrapping* durchgeführt werden

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

OPTIONAL BINDING

- *Optional Binding*:
  - wird verwendet um herauszufinden ob ein Optional einen Wert beinhaltet.
  - Ist das der Fall, wird dieser Wert einer temporären Variable zugewiesen.
- Verwending in `while/if`:

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

OPTIONAL BINDING

- Anstatt *forced unwrapping* kann *optional binding* verwendet werden:

```
if let actualNumber = Int(possibleNumber) {
    print("The string \"\(possibleNumber)\"
          has an integer value of \(actualNumber)")
} else {
    print("The string \"\(possibleNumber)\"
          could not be converted to an integer")
}
// Prints "The string "123" has an integer value of 123“
```

OPTIONAL BINDING

- Mehrere *optional bindings* werden durch Komma getrennt

```
if let firstNumber = Int("4"), let secondNumber = Int("42"),
        firstNumber < secondNumber && secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
// Prints "4 < 42 < 100"
```

```
if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("\(firstNumber) < \(secondNumber) < 100")
        }
    }
}
// Prints "4 < 42 < 100“
```

IMPLICITLY UNWRAPPED OPTIONALS

- In manchen Fällen ergibt ich anhand der Programmstruktur, dass ein Optional IMMER einen Wert hat, nachdem er einmal gesetzt wurde
- Es wäre sinnvoll, wenn man nicht jedes Mal vor dem Zugriff auf `nil` prüfen müsste
- Diese Art von Optionals nennt man *implicitly unwrapped optionals*
- Man gibt dem Optional die Erlaubnis, automatisch unwrapped zu werden
- Sollte jedoch irgendwann dieses Optional `nil` und nicht geprüft worden sein, kommt es zu einem *runtime error*

IMPLICITLY UNWRAPPED OPTIONALS

```
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
```

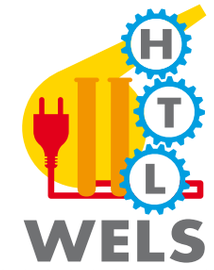
- ein *implicitly unwrapped optional* kann immer noch wie ein normales Optional verwendet werden:

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

- ebenfalls erlaubt ist *optional binding*:

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

QUELLEN



- Foliensatz basiert auf:
  - https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html