```
/*: Playground - noun: a place where people can play */
/*: # Swift – aus Sicht eines Java-Entwicklers */
import Foundation
import UIKit

let greetings = "Herzlich Willkommen, JUG Hannover"


UIImage(named: "Apple_Swift_Logo")

print("\(NSDate()) – Jonas Büth - ios@cornr.de")

/*: Na, dann lass uns mal mit einem Hello World anfangen... */

print("hello, world!")

//: [Weiter](@next)
```

```
/*:
## Über Swift
* Swift wurde im Juni 2014 auf der WWDC der Öffentlichkeit vorgestellt
* Seit 2010 unter "geheimer" Entwicklung
* ...durch ein kleines Team rund um Chris Lattner (primärer Author von
    `LLVM, clang`)
* Basiert auf Ideen aus Objective-C, Rust, Haskell, Ruby, Python, C#,
    CLU
* Grundprinzipien: Fast, Safe, Modern, Interactive and Open
* Rasanter Popularitätszuwachs, da Apple es für die iOS und Mac OS
    Entwicklung pushed
* 2 große Meilensteine Swift 1.2 und 2.0 in 2015
*/

//: [Zurück](@previous) | [Weiter](@next)
```

```
/*:
## Warum gibt es Swift?
* Swift Apple dient als Ersatz von Objective-C
* Objective-C wurde in den Anfängen der 80er von NeXT entwickelt
* NeXT wurde 1996 inkl. Steve Jobs von Apple übernommen
* Objective-C bildet zusammen mit dem Cocoa-Framework seit dem die Basis
    für Mac OS X und iOS (sowie watchOS und tvOS)
* Objective-C ist eine Objekt-orientierte, Smalltalk-änliche Syntax-
    Erweiterung von C

    `Person jonas *= [[Person alloc] initWithName:@"Jonas"
        surname:@"Büth"];`

### Etwas neues musste also her...
* "Objective-C without the C"
*/

import Foundation
//: `import` importiert ein Framework. In Foundation befinden sich die
    Cocoa Grundlagen. Diese sind zwar noch größten Teils in Objective-C
    geschrieben. Das macht aber nichts, da die Sprachen kompatibel in
    beide Richtungen sind.

let swift = createAwesomeProgrammingLanguge(buildOn:
    [.ObjectiveC, .Rust, .Haskell, .Ruby, .Python, .CSharp, .CLU, .Java]
    )

//: [Zurück](@previous) | [Weiter](@next)
```

```
import Foundation
//: ## Simple Values
//: Variablen und Konstanten
var myVariable = 42
myVariable = 50
let myConstant = 42

/*:
Die Typen werden automatisch bestimmt. (*type inference*)

Zu jedem Zeitpunkt ist klar welcher Typ eine Variable hat. Eine Variable
    kann ihren Typ nicht mehr ändern. (*strongly typed*)
*/
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70

//: Typen werden nicht automatisch "konvertiert".
//:
let label = "The width is "
let width = 94
let widthLabel = label + String(width)

//: Noch einfach geht es mit String Interpolation
//:
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."

//: ### Arrays und Dictionaries
//: (vgl. List und Array, sowie Maps und Sets)
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[0] = "bottle of water"

var occupations = [
    "Malcolm": [1 : 2],
    "Kaylee": "asd",
 ]
occupations["Jayne"] = "Public Relations"

//: Leere Arrays oder Dictionaries lassen sich über die Konstruktor-
    Syntax erstellen.
let emptyArray = [String]()
let emptyArray2: [String] = []
let emptyDictionary = [String: Float]()

//: Wie fast überall gilt: wenn der Typ klar ist, na dann kann man in
    auch weglassen.
shoppingList = []
occupations = [:]

//: Unsortierte Mengen. Achtung: Typangabe ist hier Pflicht.
//: Zu Generics später mehr...
var numbers: Set<Int> = [1, 2, 3, 5, 5, 5]
```

```
//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Control Flow
//: Wie soll es anders sein: `if`, `switch` für Bedingungen und `for`-`in`, `for`, `while`, `repeat`-`while` für Loops. Runde Klammern sind optional, Geschweifte sind pflicht. 😀
//:
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)

//: ### Exkurs: `Optionals` ❓❗ ByeBye NullPointerException...
//: Variablen die optional (sprich `nil`, aka null) sein können, müssen mit einem ? markiert werden
var optionalString: String? = "ssd"
print(optionalString == nil)

var optionalName: String? = "Jonas"
var greeting = "Hello!"
//if let packt ein "Optional" aus
if let name = optionalName {
    greeting = "Hello, \(name)"
}

//: `if let` von `nil` wertet sich zu `false` und wird damit nicht ausgeführt.

//: Switches funktionieren über alle einfachen Datentypen und müssen "exhaustive" sein, d.h. alle möglichen Werte berücksichtigen. `break` ist optional.
//:
let vegetable = "green pepper"
switch vegetable {
    case "celery":
        print("Add some raisins and make ants on a log.")
    case "cucumber", "watercress":
        print("That would make a good tea sandwich.")
    case let x where x.hasSuffix("pepper"):
        print("Is it a spicy \(x)?")
default:
    print("Everything tastes good in soup.")
}

//:
//: `for`-`in` im Detail.
//: Dictionaries sind im übrigen unsortiert.
//:
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
```

```
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25]
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)

//:
//: Mehr Beispiele:
//:
var n = 2
while n < 100 {
    n = n * 2
}
print(n)

var m = 2
repeat { //repeat hieß mal 'do', dazu später mehr
    m = m * 2
} while m < 100
print(m)

//: `..<` erzeugt Range exclusiv
//: `...` erzeugt Range inclusiv
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
print(firstForLoop)

var secondForLoop = 0
for var i = 0; i < 4; ++i {
    secondForLoop += i
}
print(secondForLoop)

//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Functions and Closures
//: `func` für Funktionen. Die Typen stehen wie in der Variablen-Deklaration hinter dem Namen. Hinter `->` kommt der Rückgabetyp. `->` ist optional, wenn die Funktion keinen Rückgabetyp hat.
//: Parameter können auch Default-Werte haben.
func greet(name name: String, tag day: String = "Wednesday") -> String {
    return "Hello \(name), today is \(day)."
}
greet(name: "Bob", tag: "Tuesday")
greet(name: "Tobi")

/*:
### Exkurs: Tupel
Tupel sind zusammengefasste Werte zu einem anonymen Typen. Die Werte können benannt werden.
Wenn man genau hin sieht, ist die Parameterdefinition einer Funktion eine Tupeldefinition. Der Aufruf einer Funktion ist die Instanz des Tupels.
*/
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}
let statistics = calculateStatistics([5, 3, 100, 3, 9])
print(statistics.sum)
print(statistics.0)

//: Implizite Arrays in Funktionsaufrufen
//:
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}
sumOf()
sumOf(42, 597, 12)

//: Funktionen kann man auch verschachteln (muss man aber nicht).
//:
```

```
func add() -> String {
    return ""
}

func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}


returnFifteen()

//: Funktionen sind first-class types.
//:
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)

//: Dann lass uns mal Funktionen weiterreichen.
func hasAnyMatches(list: [Int], condition: Int -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, condition: lessThanTen)

//: Funktionen sind benannte Closures. Es geht auch ohne Namen...
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})

//: Wenn ein closures-Typ bereits klar ist, kann man die Parameter und
    die Rückgabetyp-Definition weglassen. Bei Einzeilern kann auch das
    `return` weg.
//:
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
```

```
//: Noch kürzer und abgefahrener...
//:

let sortedNumbers = numbers.sort { $0 > $1 }
print(sortedNumbers)



//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Objects and Classes
//:
//: `class` für Java-Entwickler, gibt es hier nicht viel Neues.
//:
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}

//: Instanzen erzeugt man durch den Konstruktor-Aufruf (allerdings ohne
    `new`)
//:
var shape = Shape()
shape.numberOfSides = 7
//var shapeDescription = shape.simpleDescription()

//: Oben fehlte der Konstruktor (ist er leer, kann weggelassen werden).
//:
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}

/*:
Beachte:
* Alle properties müssen nach dem Instanziieren gesetzt sein.
* `this` heißt in Swift `self`
* `deint` als Gegenstück zum Konstruktor, kann genutzt werden um
    Ressourcen freizugeben. A propos: freigeben...


*/

//: ### Exkurs: Automatic Reference Counting (ARC)
//: * Es gibt keinen Garbage-Collector aus (ehemals) Speicher- und CPU-
    Effizienzgründen
//: * Stattdesen zählt das System Referenzen
//: * Zeigt niemand mehr auf ein Objekt, wird es sofort aufgeräumt


//: Vererbung mit `:` und Supertype. Mehrfachvererbung gibt es nicht.
//: Überschrieben von Methoden mit `override`-Schlüsselwort
//:
final class Square: NamedShape {
```

```
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        self.numberOfSides = 4
    }

    final func area() ->  Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()

//: *Calculated Properties* (aka getter / setter)
//:
class EquilateralTriangle: NamedShape {
    private var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        self.numberOfSides = 3
    }

    var perimeter: Double {
        get {
            print("perimeter")
            return Double(self.numberOfSides) * sideLength
        }
        set {
            sideLength = newValue / Double(self.numberOfSides)
        }
    }

    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \
            (sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)
print(triangle.perimeter)
triangle.perimeter = 9.9
triangle.perimeter = 10.9
print(triangle.sideLength)

//: *Property Observer*
//:
```

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength = newValue.sideLength
        }
    }
    var square: Square {
        willSet {
            triangle.sideLength = newValue.sideLength
        }
    }
    init(size: Double, name: String) {
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength: size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test
    shape")
print(triangleAndSquare.square.sideLength)
print(triangleAndSquare.triangle.sideLength)
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
print(triangleAndSquare.triangle.sideLength)

//: Ergänzung zu Optionals: *Optional Chaining*
//:
var optionalSquare: Square? = Square(sideLength: 2.5, name: "optional
    square")
let sideLength = optionalSquare?.sideLength
optionalSquare = nil
optionalSquare?.simpleDescription()



//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Enumerations and Structures
//:
//: Enums können wie jeder benannte Typ Funktionen haben.
//: Über einen "super"-Typ lassen sich Raw-Values festlegen. Im Beispiel
    Integer beginnent bei 1, alternativ kann jeder Wert ein speziellen
    Raw-Value haben.
//:
enum Rank: String {
    case Ace = "Ass"
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace"
        case .Jack:
            return "jack"
        case .Queen:
            return "queen"
        case .King:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
}
let ace = Rank.King
let aceRawValue = ace.rawValue

//: Über den Raw-Value lässt sich der entsprechende Enum-Wert
    instanziieren.
if let convertedRank = Rank(rawValue: "Assjasd") {
    let threeDescription = convertedRank.simpleDescription()
}

//: Wenn der "super"-Typ wegelassen wird, sind die Werte der Raw-Value.
//:
enum Suit: String {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
            case .Spades: //self ist vom Typ Suit, also brauch man das
                auch nicht hinschreiben
                return "spades"
            case .Hearts:
                return "hearts"
            case .Diamonds:
                return "diamonds"
            case .Clubs:
                return "clubs"
        }
    }
}
```

```
let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()


//: `associated values`: Eine Enum-Instanz kann auch zugehörige Werte
    haben.
enum ServerResponse {
    case Result(sunrise: String, sunset: String)
    case Error(String)
}

let success = ServerResponse.Result(sunrise: "6:00 am", sunset: "8:09
    pm")
let failure = ServerResponse.Error("Out of cheese.")

switch success {
    case let .Result(sunrise, sunset):
        let serverResponse = "Sunrise is at \(sunrise) and sunset is at
            \(sunset)."
    case let .Error(error):
        let serverResponse = "Failure...  \(error)"
}

//Das funktioniert soger rerkursiv. whoa...
enum SimpleTree<T> {
    case Leaf(T)
    indirect case Node(SimpleTree, SimpleTree)
}

let leaf = SimpleTree.Leaf(1)
let tree = SimpleTree.Node(leaf, SimpleTree.Node(SimpleTree.Leaf(2),
    SimpleTree.Leaf(3)))


//: `struct` erstellt Structures. Fast wie Classes: mit Methods,
    Initializer und Properties ABER: keiner Vererbung und Call-By-Value-
    Semantik. Das heißt Structures werden beim "herumgeben" immer
    kopiert.
//:
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.
            simpleDescription())"
    }

}
let threeOfSpades = Card(rank: .Three, suit: .Spades) //Konstruktoren
    werden generiert
let threeOfSpadesDescription = threeOfSpades.simpleDescription()


//: Structs sollten immer Equatable implementieren.
//: Denn auf Werten gilt intuitiv immer die Reflexivität, Symmertrie und
```

```
    Transitivität

struct MyInt: Equatable {
    var value: Int
}

func ==(lhs: MyInt, rhs: MyInt) -> Bool {
    return lhs.value == rhs.value
}


MyInt(value: 2).value
MyInt(value: 2) == MyInt(value: 2)

//Beispiel: UndoStack
var undoStack: [MyInt] = []
var model = MyInt(value: 1)
undoStack.append(model)

model.value = 2
undoStack.append(model)

model.value = 3
undoStack.append(model)

model.value = 4

model = undoStack.removeLast()
model.value
undoStack

//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Protocols and Extensions
//:
//: `protocol` die swiftsche Schnittstelle
//:
protocol ExampleProtocol {
     var simpleDescription: String { get }
     mutating func adjust()
}

//: Classes, enumerations und structs können alle Protokolle
    implementieren.
//:
class SimpleClass: ExampleProtocol {
     var simpleDescription: String = "A very simple class."
     var anotherProperty: Int = 69105
     func adjust() {
          simpleDescription += "  Now 100% adjusted."
     }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
     var simpleDescription: String = "A simple structure"
     mutating func adjust() {
          simpleDescription += " (adjusted)"
     }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription

//: Das `mutating` keyword kennzeichnet Methoden, die Structures
    verändern.

//: Mit `extension` kann man vorhandenen Typen Funktionalität
    hinzufügen.
//:
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
 }
print(7.simpleDescription)


//: Refernzen auf Schnittstellen, verbergen den dahinterliegenden Typen,
    ganz wie in Java.
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)
```

```
//print(protocolValue.anotherProperty)  // Fehler

//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Generics
//:
//: Generics sind für Java-Entwickler nichts Neues.
//:
func repeatItem<Item>(item: Item, numberOfTimes: Int) -> [Item] {
    var result = [Item]()
    for _ in 0..<numberOfTimes { //_ wird als Platzhalter für Werte
        verwendet, die gar nicht benötigt werden
         result.append(item)
    }
    return result
}
repeatItem("knock", numberOfTimes:4)

//: Sie funktionieren für Funktionen, Methoden, Klassen, Enumerationen
    und Strukturen.
//:
// Reimplement the Swift standard library's optional type
enum OptionalValue<Wrapped> {
    case None
    case Some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger = .Some(100)

//: Mit der `where`-Klausel lassen sich Typen weiter einschränken.
//:
func anyCommonElements <T: SequenceType, U: SequenceType where T.
    Generator.Element: Equatable, T.Generator.Element == U.Generator.
    Element> (lhs: T, _ rhs: U) -> Bool {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
   return false
}
anyCommonElements([1, 2, 3], [3])


//: [Zurück](@previous) | [Weiter](@next)
```

```
import Foundation
//: ## Pattern Matching
//: Pattern Matching ist eine Erweitung der Conditions die sich auf
    `switch, ifs, for .. in` anwenden lässt.

enum Animal {
    case Dog, Cat, Troll, Dragon
}

for animal: Animal in [.Dog, .Cat, .Troll] where animal == .Dog {
    print(animal)
}

//Union-Typ
enum Either<T1, T2> {
    case First(T1)
    case Second(T2)
}
// Union für Alternative Ergebnisse einer Funktion
func test() -> Either<Animal, String> {
    return Either.First(.Dog)
}

let either = test()

switch either {
    case .First(let animal):
        print(animal)
    case .Second(let string):
        print("nothing heere")
}

if case .First(let animal) = either where animal == .Dog {
    print("test")
}

//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Protocol Extension
//: Neben Klassen, Structs und Enums lassen sich auch Protokolle um
    Implementierungen erweitern (*default implemenation*).

protocol Hello {
    func sayHello() -> String
}

extension Hello {
    func sayHello() -> String {
        return "Hello, stranger"
    }
}

class MyClass: Hello {
    func sayHello() -> String {
        return ""
    }
}


let c = MyClass()
c.sayHello()

//: Damit lassen sich Traits (bwz. Mixins) realisieren.
//: ### Ein Beispiel: Ruby Enumerables
//: Wenn man in Ruby das `Enumerable` Interface implementiert und damit
    `each()` bereitstellt, erhält man einen ganzen Sack voller
    Hilfsoperationen dazu: `map(), group_by(), find_all(), drop_if()`
//: Wie lässt sich das in Swift realisieren?

protocol Enumerable {
    typealias Element // Der typealias übernimmt hier die Rolle eines
        Generics. Generics sind noch nicht bis in die Protocols
        durchgezogen :(
    func each(block: (Self.Element) -> Void) //Das große Self, ist der
        implementierende Typ
}

//: Hilfsmethoden als Extension bereitstellen:
extension Enumerable {
    func dropIf(predicate: (Self.Element) -> Bool) -> [Self.Element] {
        var result = [Element]()
        each { item in if !predicate(item) { result.append(item) } }
        return result
    }

    //    func dropWhile(predicate: (Self.Element) -> Bool) ->
        [Self.Element] {
    //        ...
    //    }
    //
    //    func findAll(predicate: (Self.Element) -> Bool) ->
        [Self.Element] {
```

```
    //        ...
    //    }

    /* many more methods here */
}

struct Family {
    let name = "Smith"
    let father = "Bob"
    let mother = "Alice"
    let child = "Carol"
}

//Protokoll Implementierungen werden oft als Extension realisiert, um
    die Implementierungen zu gruppieren
extension Family : Enumerable {
    func each(block: (String) -> Void) {
        for i in 0...2 {
            switch i {
            case 0: block("\(self.father) \(self.name)")
            case 1: block("\(self.mother) \(self.name)")
            case 2: block("\(self.child) \(self.name)")
            default: break
            }
        }
    }
}

let f = Family()
let withoutBob = f.dropIf { p in p.hasPrefix("Bob") }
withoutBob

//: [Zurück](@previous) | [Weiter](@next)
```

```swift
import Foundation
//: ## Error Handling
//: Bis Swift 2 gab es faktisch kein Error Handling bzw. das von
    Objective-C wurde kopiert:

func loadData(inout error: NSError?) -> Bool {
    //load data and eventually create an error:
    error = NSError(domain: "myDomain", code: 4711, userInfo: nil)
    return false //Konvention: Wenn false dann ist ErrorPointer gesetzt
}

var error: NSError?
if !loadData(&error) {
    if let error = error {
        print(error)
    }
}

//: Alternativ wird auch gerne in Konstruktoren `nil` geliefert.
class Person {
    let name: String
    init?(optionalName: String?) {
        if let name = optionalName {
            if name.characters.count > 0 && name.characters.count < 10 {
                self.name = name
                return
            }
        }
        self.name = "wtf?"
        return nil
    }
}
let person = Person(optionalName: "Test")?.name
let noPerson1 = Person(optionalName: "")?.name
let noPerson2 = Person(optionalName: nil)

//: Gott-Sei-Dank macht Swift 2 einiges besser.
enum PersonError: ErrorType {
    case NoName
    case NameToLong(String)
}

extension Person {
    class func createPerson(name name: String?) throws -> Person {
        if let name = name {
            if let person = Person(optionalName: name) {
                return person
            } else {
                throw PersonError.NameToLong(name)
            }
        } else {
            throw PersonError.NoName
        }
    }
}
```

```
do {
    let name = "namezulaaaaaaang"
    let person2 = try Person.createPerson(name: name)
} catch PersonError.NoName {
    print("kein Name angegeben")
} catch PersonError.NameToLong(let name) {
    print("\(name) hat mehr als 10 Zeichen.")
} catch let error {
    print("some Error \(error)")
}

//: ### Guard Statement - no more Pyramid of Doom
class ProperPerson {
    let name: String

    init(name: String?) throws {
        guard let name = name else {
            self.name = ""
            throw PersonError.NoName
        }
        guard name.characters.count > 0 && name.characters.count < 10
            else {
            self.name = ""
            throw PersonError.NameToLong(name)
        }

        self.name = name
    }
}


//: [Zurück](@previous) | [Weiter](@next)
```

```
/*:
## Wie geht es weiter?
*/
/*:
### Swift wird Open-Source
* OSI-approved license
* Later this year
* Code contributions accepted
* Apple wird Linux Compiler bereitstellen
*/
/*: ### Wird Google Swift für Android SDK adoptieren? */
/*: ### Kommt eine JVM Implementierung? */
/*: ### AppCode - Alternative IDE */

//: [Zurück](@previous) | [Weiter](@next)
```

```
/*:
# Quellen und weitere Links

* Swift Documentation
  * [https://developer.apple.com/library/ios/documentation/Swift/
      Conceptual/Swift_Programming_Language/]()


* WWDC Videos
  * [https://developer.apple.com/videos/wwdc/2015/]()
  * What's New in Swift
  * Protocol-Oriented Programming in Swift
  * Building Better Apps with Value Types in Swift


* Equatable Structs
  * [https://www.andrewcbancroft.com/2015/07/01/every-swift-value-type-
      should-be-equatable/]()


* Mixin and Traits
  * [http://matthijshollemans.com/2015/07/22/mixins-and-traits-in-
      swift-2/]()


* AppCode
  * [http://www.jetbrains.com/objc/]()


* Einstieg in iOS Development
  * [cs193p.stanford.edu/]() [https://itunes.apple.com/us/course/
      developing-ios-8-apps-swift/id961180099]()
  * [https://github.com/matteocrippa/awesome-swift/blob/master/
      README.md]()
  * [https://iosdevweek.ly]()


*/

//: [Zurück](@previous)
```