App-Entwicklung für iOS und OS X

SS 2016

Stephan Gimbel

Datentypen

Ein weiterer Typ ist AnyObject, welches ein Objekt eines beliebigen Typs sein kann.

```
var someValues = [AnyObject]()
```

Typen können mittels Type casting umgewandelt werden. Schauen wir uns später an, zuerst noch ein paar Grundlagen.

Funktionen

Mit Parameter und Rückgabewert

```
func sayHelloAgain(personName: String) -> String {
    return "Hello again, " + personName + "!"
}
print(sayHelloAgain("Anna"))
// Prints "Hello again, Anna!"
```

Mit mehreren Parametern

```
func sayHello(personName: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return sayHelloAgain(personName)
    } else {
        return sayHello(personName)
    }
}
print(sayHello("Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

Funktionen

Ohne Rückgabewert

```
func sayGoodbye(personName: String) {
    print("Goodbye, \(personName)!")
}
sayGoodbye("Dave")
// Prints "Goodbye, Dave!"
```

Ignorieren von Rückgabewerten

```
func printAndCount(stringToPrint: String) -> Int {
    print(stringToPrint)
    return stringToPrint.characters.count
}
func printWithoutCounting(stringToPrint: String) {
    printAndCount(stringToPrint)
}
printAndCount("hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting("hello, world")
// prints "hello, world" but does not return a value
```

Funktionen

Mehrere Rückgabewerte

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

```
let bounds = minMax([8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```

Funktionen

Optionals als Rückgabewerte

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

```
if let bounds = minMax([8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
// Prints "min is -6 and max is 109"
```

Der Rückgabewert kann mittels Optional Binding geprüft werden

`(Int, Int)?` ist nicht identisch mit `(Int?, Int?)`

Funktionen

Parameter für Funktionen können externe und lokale Parameternamen haben.

Per Default wird der externe Name des ersten Parameters unterdrückt, der zweite und weitere Parameter verwenden ihren lokalen Namen als externen Namen. Alle Parameter müssen eindeutige lokale Namen haben. Die externen Namen können identisch sein (nicht empfohlen!)

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // function body goes here
    // firstParameterName and secondParameterName refer to
    // the argument values for the first and second parameters
}
someFunction(1, secondParameterName: 2)
```

Funktionen

Externe Parameternamen können spezifiziert werden.

```
func someFunction(externalParameterName localParameterName: Int) {
    // function body goes here, and can use localParameterName
    // to refer to the argument value for that parameter
}
```

Beispiel

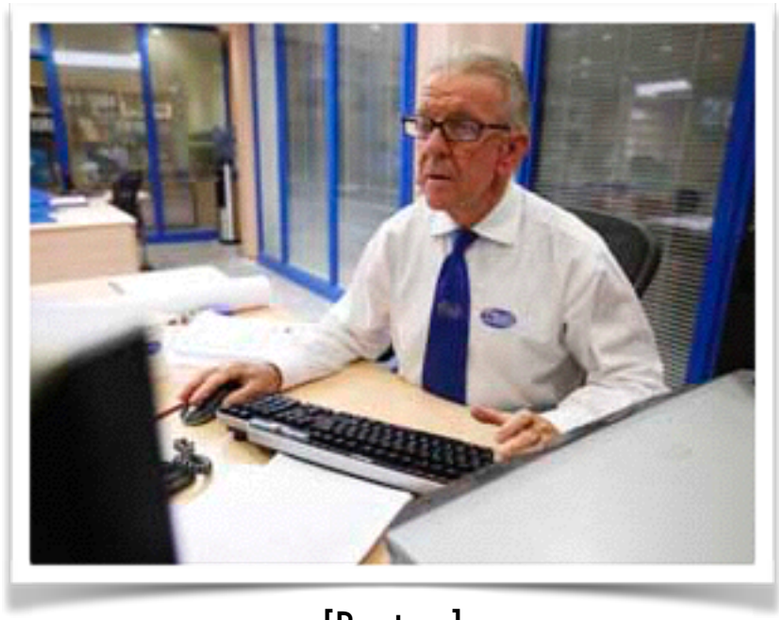
```
func sayHello(to person: String, and anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}
print(sayHello(to: "Bill", and: "Ted"))
// Prints "Hello Bill and Ted!"
```

Wird ein externer Name für den ersten Parameter spezifiziert, so muss dieser auch verwendet werden.

Funktionen

Unterdrückung von externen Parameternahmen (the old way)

```
func someFunction(firstParameterName: Int, _ secondParameterName: Int) {
    // function body goes here
    // firstParameterName and secondParameterName refer to
    // the argument values for the first and second parameters
}
someFunction(1, 2)
```



[Reuters]

Funktionen

Default Parameter

```
func someFunction(parameterWithDefault: Int = 12) {
    // function body goes here
    // if no arguments are passed to the function call,
    // value of parameterWithDefault is 12
}
someFunction(6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12
```

Variadic Parameter zur Übergabe von keinen oder mehrerer Werte.

```
func arithmeticMean(numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

Funktionen

In-Out Parameter zur Übergabe von Referenzen

```
func swapTwoInts(inout a: Int, inout _ b: Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

In-Out Parameter können keinen Default-Value haben. Ebenso können Variadic Parameter nicht als In-Out Parameter verwendet werden.

Funktionen

Jede Funktion hat einen Funktionstypen (Function Type), der sich aus der Signatur der Funktion ergibt.

```
func addTwoInts(a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(a: Int, _ b: Int) -> Int {
    return a * b
}
```

Die beiden Funktionen sind vom gleichen Typ: `(Int, Int) -> Int`

```
func printHelloWorld() {
    print("hello, world")
}
```

Der Type dieser Funktion ist: `() -> Void`

Funktionen

```
func addTwoInts(a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(a: Int, _ b: Int) -> Int {
    return a * b
}
```

Verwendung von Funktionstypen

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

Liest sich: *"Definiere die Variable* `mathFunction`*, welche den Typ einer Funktion hat, die zwei* `Int` *Werte annimmt und einen* `Int` *Wert zurück gibt. Weise dieser neuen Variable die Funktion* `addTwoInts` *zu"*.

```
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
```

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

Funktionen

Funktionstypen als Parameter Typ

```
func printMathResult(mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8"
```

Funktionen

Funktionstypen als Rückgabe Typ

```
func stepForward(input: Int) -> Int {
    return input + 1
}
```

```
func stepBackward(input: Int) -> Int {
    return input - 1
}
```

Beide Funktionen sind vom Typ `(Int) -> Int`

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
```

Der Rückgabewert von `chooseStepFunction` **ist vom Typ "Funktion vom Typ** `(Int) -> Int`**"**

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

Basierend auf `backwards`**, gibt** `chooseStepFunction` **die Funktion** `stepForward` **oder** `stepBackward` **zurück.**

Funktionen

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

Unabhängig vom Vorzeichen von Value zählt nähert sich der Algorithmus an Null an.

Funktionen

Nested Functions sind nur im Scope der einbettenden Funktion sichtbar und können nur von dort aufgerufen werden

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

Enumeration

Enumerationen dienen zum gruppieren von verwandten Werten.

Im Gegensatz zu C und Objective-C wird in Swift kein Default Value von 0, 1, 2, usw. zugewiesen. Die Cases sind eigenständige Werte!

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

```
enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

```
enum Planet {
    case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

```
var directionToHead = CompassPoint.West
directionToHead = .East
```

Enumeration

Matching Enumerations

```
directionToHead = .South
switch directionToHead {
case .North:
    print("Lots of planets have a north")
case .South:
    print("Watch out for penguins")
case .East:
    print("Where the sun rises")
case .West:
    print("Where the skies are blue")
}
// Prints "Watch out for penguins"
```

```
let somePlanet = Planet.Earth
switch somePlanet {
case .Earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// Prints "Mostly harmless"
```

Wenn nicht alle Fälle explizit berücksichtigt werden, muss ein Default-Case vorhanden sein.

Enumeration

In manchen Fällen ist es nützlich zusätzlich Associated Values zu den einzelnen Fällen zu speichern.

```
enum Barcode {
    case UPCA(Int, Int, Int, Int)
    case QRCode(String)
}
```

```
var productBarcode = Barcode.UPCA(8, 85909, 51226, 3)
```

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

Enumeration

Associated Values können für jeden Fall als Variable bzw. Konstante extrahiert werden

```
switch productBarcode {
case .UPCA(let numberSystem, let manufacturer, let product, let check):
    print("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).")
case .QRCode(let productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

```
switch productBarcode {
case let .UPCA(numberSystem, manufacturer, product, check):
    print("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).")
case let .QRCode(productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

Enumeration

Enumeration Values können vorinitialisiert werden (Raw Values).

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

Werte können ebenfalls implizit zugewiesen werden.

```
num Planet: Int {
    case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

```
enum CompassPoint: String {
    case North, South, East, West
}
```

```
let earthsOrder = Planet.Earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.West.rawValue
// sunsetDirection is "West"
```

Enumeration

Recursive Enumeration

```
enum ArithmeticExpression {
    case Number(Int)
    indirect case Addition(ArithmeticExpression, ArithmeticExpression)
    indirect case Multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

```
indirect enum ArithmeticExpression {
    case Number(Int)
    case Addition(ArithmeticExpression, ArithmeticExpression)
    case Multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

Enumeration

```
let five = ArithmeticExpression.Number(5)
let four = ArithmeticExpression.Number(4)
let sum = ArithmeticExpression.Addition(five, four)
let product = ArithmeticExpression.Multiplication(sum,
ArithmeticExpression.Number(2))
```

```
func evaluate(expression: ArithmeticExpression) -> Int {
    switch expression {
    case let .Number(value):
        return value
    case let .Addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .Multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// Prints "18"
```

Klassen und Strukturen

Hinweis:
Traditionell ist eine Instanz einer Klasse als Objekt bekannt. Klassen und Strukturen in Swift sind sich in ihrer Funktion ähnlicher als in anderen Sprachen. Die vorgestellte Funktionalität kann für Klassen- oder Strukturtypen verwendet werden, daher beschränken wir uns auf den generelleren Ausdruck Instanz.

Classes vs. Structs

Klassen und Strukturen haben folgende Gemeinsamkeiten:

- Definieren Properties um Werte zu speichern
- Definieren Methoden für Funktionalität
- Definieren Subscripts um Zugriff auf ihre Werte mittels Subscript Syntax zur Verfügung zu stellen
- Definieren Initializer um initiale Werte zuzuweisen
- Können erweitert werden um zusätzliche Funktionalität zur Verfügung zu stellen
- Sind konform zu Protokollen, um Standard Funktionalität zur Verfügung zu stellen

Klassen und Strukturen

Drüber hinaus haben Klassen folgende Eigenschaften, die Strukturen nicht haben

- Vererbung
- Type Casting ermöglicht Prüfung und Interpretation des Typs einer Klasseninstanz zur Laufzeit
- Deinitializers ermöglichen einer Klasseninstanz Resources freizugeben, die sie vergeben hat
- Reference Counting erlaubt mehr als eine Referenz auf eine Klasseninstanz

Syntax

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

Klassen und Strukturen

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

Erstellen einer Instanz

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Klassen und Strukturen

Zugriff auf Properties

```
print("The width of someResolution is \(someResolution.width)")
// Prints "The width of someResolution is 0"
```

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is 0"
```

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is now 1280"
```

Structure Types haben automatisch generierte Memberwise Initializer

```
let vga = Resolution(width: 640, height: 480)
```

Klassen und Strukturen

Structs und Enums sind Value Types, die bei Zuweisung kopiert werden

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

```
cinema.width = 2048
```

```
print("cinema is now \(cinema.width) pixels wide")
// Prints "cinema is now 2048 pixels wide"

print("hd is still \(hd.width) pixels wide")
// Prints "hd is still 1920 pixels wide"
```

```
enum CompassPoint {
    case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
currentDirection = .East
if rememberedDirection == .West {
    print("The remembered direction is still .West")
}
// Prints "The remembered direction is still .West"
```

Klassen und Strukturen

Klassen sind Referenztypen

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

```
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// Prints "The frameRate property of tenEighty is now 30.0"
```

Klassen und Strukturen

Klassen sind Referenztypen

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

```
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// Prints "The frameRate property of tenEighty is now 30.0"
```

Warum kann hier eine Zuweisung erfolgen, wo `alsoTenEighty` doch eine Konstante ist?

Klassen und Strukturen

Identity Operator

Da Klassen Referenztypen sind können mehrere Konstanten oder Variablen auf die selbe Instanz einer Klasse verweisen. Manchmal ist es sinnvoll herauszufinden ob dies der Fall ist

- Identisch zu (===)
- Nicht identisch zu (!==)

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode
instance.")
}
// Prints "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

Identisch zu (===) ist nicht das selbe wie Gleichheit (==)

- Identisch zu bedeutet, zwei Variablen oder Konstanten eines Klassentyps referenzieren die exakt selbe Klasseninstanz
- Gleichheit bedeutet, dass zwei Instanzen gleich sind, wenn deren Werte gleich sind.

Klassen und Strukturen

Klasse oder Struktur?

Strukturen werden als Value übergeben, Klassen dagegen als Referenz, daher eignen sich beide für unterschiedliche Anwendungsfälle. Die Auswahl ob eine Klasse oder Struktur verwendet werden soll, hängt von Ihrer Implementierung ab.

Grobe Richtlinie, wann Sie eine Struktur verwenden sollten:

- Primär zum kapseln von wenigen einfachen Daten/Werten
- Es wird erwartet dass die gekapselten Werte der Struktur kopiert und nicht referenziert werden
- Jedes Property welches von einer Struktur gespeichert wird, sind selbst Value Types die kopiert werden
- Die Struktur muss keine Eigenschaften/Properties und Funktionalität von einem existierendem Typen aufweisen

Klassen und Strukturen

Gute Kandidaten für Strukturen sind u.a.

- Größe von Geometrie, z.B. Rechteck mit `width` und `height` vom Typ `Double`
- Zugriff auf einen Bereich innerhalb einer Reihe mit `start` und `length` vom Typ `Int`
- Punkt in einem 3D Koordinatensystem, spezifiziert durch `x`, `y` und `z` Koordinate vom Typ `Double`

Klassen und Strukturen

Initializers erlauben die Erstellung von Instanzen. Dazu gehört auch das Zuweisen von (Default) Werten für Properties.

```
init() {
    // perform some initialization here
}
```

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// Prints "The default temperature is 32.0° Fahrenheit"
```

```
struct Fahrenheit {
    var temperature = 32.0
}
```

Ebenso existieren Deinitializer (`deinit`), die aufgerufen werden bevor eine Instanz deallokiert wird.

Properties

Properties assoziieren Werte mit bestimmten Klassen, Strukturen (Struct) oder Enumerationen (Enum).

Properties unterschieden sich nach

- Stored Properties
- Computed Properties

Stored Properties speichern einen konstanten oder variablen Wert als Teil einer Instanz und werden von Klassen und Structs zur Verfügung gestellt.

Computed Properties berechnen einen Wert und werden von Klassen, Structs und Enums zur Verfügung gestellt.

Normalerweise werden stored und computed Properties mit Instanzen eines bestimmten Typs assoziiert, jedoch können Properties auch mit einem Typ selbst assoziiert werden. Solche Properties sind als Type Properties bekannt.

Eine weitere Möglichkeit besteht darin Property Observers zu definieren, die Properties bzgl. Änderungen überwachen.

Properties

Stored Properties

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

Stored Properties von konstanten Strukturinstanzen

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable property
```

Properties

Wenn die Initialisierung von äußeren Faktoren abhängt oder extrem "teuer" ist, können Lazy Stored Properties verwendet werden. Dies stellt sicher, dass Properties erst dann erstellt werden, wenn diese auch benötigt werden.

```
class DataImporter {
    /*
     DataImporter is a class to import data from an external file.
     The class is assumed to take a non-trivial amount of time to initialize.
     */
    var fileName = "data.txt"
    // the DataImporter class would provide data importing functionality here
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    // the DataManager class would provide data management functionality here
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
// the DataImporter instance for the importer property has not yet been
created
```

Properties

Erst wenn der erste Zugriff auf DataImporter erfolgt, wird die Instanz erstellt.

```
print(manager.importer.fileName)
// the DataImporter instance for the importer property has now been created
// Prints "data.txt"
```

Lazy Properties müssen immer Variablen (var) sein, da der initiale Wert möglicherweise nicht gesetzt ist nachdem die Initialisierung abgeschlossen ist.

Wenn der Zugriff auf ein Lazy Property von mehreren Threads gleichzeitig erfolgt und dieses noch nicht initialisiert wurde, gibt es keine Garantie dass die Initialisierung nur einmal erfolgt.

Properties

Für Klassen, Structs und Enums existieren ebenfalls Computed Properties, die keinen Wert speichern, aber einen Getter und Setter (optional) für andere Properties zur Verfügung stellen.

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
                  size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// Prints "square.origin is now at (10.0, 10.0)"
```

Properties

Es existiert eine kurze Setter Variante, welche den Default Namen `newValue` verwendet, wenn kein Name spezifiziert wird.

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

Properties

Ein Property kann Read-Only sein, wenn auf einen Setter verzichtet wird.

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// Prints "the volume of fourByFiveByTwo is 40.0"
```

Properties

Property Observers erlauben über die Änderung eines Properties informiert zu werden (auch wenn der neue Wert gleich dem alten Wert ist).

Property Observers können jedem Stored Property hinzugefügt werden, mit der Ausnahme von Lazy Properties. Dies trifft auch auf vererbte (stored oder computed) Properties zu, die überschrieben (override) werden müssen.

Ein oder beide dieser Observer können definiert werden:

- `willSet` wird aufgerufen kurz bevor der neue Wert geschrieben wird
- `didSet` wird aufgerufen, direkt nachdem der neue Wert geschrieben wurde

An `willSet` wird der neue Wert als konstanter Parameter übergeben. An `didSet` wird dagegen der alte Wert als konstanter Parameter übergeben.

Properties

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

Access Control

Wie andere Sprachen verfügt auch Swift über einen Access Control Mechanismus um den Zugriff auf Teile des Codes einzuschränken. Dies trifft auf individuelle Typen (Klassen, Strukturen, Enumerationen), aber auch auf Properties, Methoden, Initializers und Subscripts zu.

Swift verfügt über drei verschiedene Access Levels:

- Public
- Internal
- Private

Da sich das grundlegende Prinzip hinter Access Control nicht von dem aus anderen Sprachen unterscheidet, verzichten wir darauf uns diese näher anzuschauen.

Details zu den Access Levels inkl. Beispielen entnehmen Sie bei Bedarf dem Kapitel Access Control aus dem Swift Programming Guide

Vererbung

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing - an arbitrary vehicle doesn't necessarily make a noise
    }
}
```

```
class Bicycle: Vehicle {
    var hasBasket = false
}
```

Vererbung

Überschreiben von Methoden

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
```

Überschreiben von Properties

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}
```

Überschreiben von Property Observers

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

Das Überschreiben kann mittels `final` Modifier verhindert werden.