BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

MAD4

Enumerations,Klassen, Strukturen

Prof. Dr. Dragan Macos

Aufzählungstyp, Enumerations

- Typ für eine Menge aufgezählter Werte

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

Werte des Typen CompasPoint

Oder so

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

Verwendung

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

Wenn der Typ bekannt ist

```
directionToHead = .East
```

Beispiel

```
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"
```

Assoziierte Werte

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

Bis 2.953 Zeichen langer String

```
var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```

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

Beispiel

Konstantendefinition. Wert der Konstante: erste Komponente des Wertes UPCA.

Anstatt „let“ kann man auch mit „var“ Variable definieren.

```
switch productBarcode {
case .UPCA(let numberSystem, let identifier, let check):
      print ("UPC-A with value of \(numberSystem), \
        (identifier), \(check).")
case .QRCode(let productCode):
      print ("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

```
case let .UPCA(numberSystem, identifier, check):
```

Wenn alle Assoziierten Werte als Konstanten extrahiert werden → nur ein „let“ ist schicker.

Rohwerte

- Wenn alle assoziierte Werte den selben Typ haben...

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

..reicht eine Typdeklaration vor den Klammern.

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

Kürzere Form für Zuweisung der Werte von 1-8

```
1  let possiblePlanet = Planet(rawValue: 7)
```

Versucht, das Element mit dem Wert 7 zu finden. Optional. Wieso optional?

Beispiel

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

```
1  let positionToFind = 11
2  if let somePlanet = Planet(rawValue: positionToFind) {
3      switch somePlanet {
4      case .earth:
5          print("Mostly harmless")
6      default:
7          print("Not a safe place for humans")
8      }
9  } else {
10     print("There isn't a planet at position \
           (positionToFind)")
11 }
```

Ausgabe??

Beispiel

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

```
1   let positionToFind = 11
2   if let somePlanet = Planet(rawValue: positionToFind) {
3       switch somePlanet {
4       case .earth:
5           print("Mostly harmless")
6       default:
7           print("Not a safe place for humans")
8       }
9   } else {
10      print("There isn't a planet at position \
              (positionToFind)")
11  }
```

```
// Prints "There isn't a planet at position 11"
```

Rekursvie Enumerations

- Werden durch das Schlüsselwort „indirect“ gekennzeichnet

```
1  enum ArithmeticExpression {
2      case number(Int)
3      indirect case addition(ArithmeticExpression,
           ArithmeticExpression)
4      indirect case multiplication(ArithmeticExpression,
           ArithmeticExpression)
5  }
```

Oder

```
1  indirect enum ArithmeticExpression {
2      case number(Int)
3      case addition(ArithmeticExpression,
           ArithmeticExpression)
4      case multiplication(ArithmeticExpression,
           ArithmeticExpression)
5  }
```

Aufgabe

- Eine Variable als ArithmeticExpression mit dem Wert

  (5 + 4) * 2

  initialisieren.

Ein kleines Beispiel:

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let oneplustwo = ArithmeticExpression.addition(five, four)
```

```
1  indirect enum ArithmeticExpression {
2      case number(Int)
3      case addition(ArithmeticExpression,
           ArithmeticExpression)
4      case multiplication(ArithmeticExpression,
           ArithmeticExpression)
5  }
```

Bonusaufgabe (5 Punkte bei der Klausur)

- Schreiben Sie eine (rekursive) Funktion, die ein *ArithmeticExpression* auswertet.

- Anfang:

```
1  func evaluate(_ expression: ArithmeticExpression) -> Int {
2      switch expression {
3      case let .number(value):
4          return value
```

.....

Klassen, Strukturen

- Sprachkonstrukte für die Definition neuer Benutzer-Typen.
- Generell wird das Wort „Objekt“ als „Instanz einer Klasse“ verwendet.
- Da wir sowohl Strukturen als auch Klassen instanzieren können, werden wir das Wort „Instanz“ benutzen. Für beide Konstrukte

- Ein paar Unterschiede zwischen Klassen und Strukturen:
  - Klassen haben Vererbung
  - Klassen haben „casting“
  - Klassen haben „Deinitialisierer“
  - Klassen haben „Reference“ Countig – ein Mechanismus für das Zählen der Referenzen von Klasseninstanzen. Wenn eine Instanz nicht referenziert wird, wird diese gelöscht (der Speicher wird freigegeben.

Definition von Klassen und Strukturen

```
1   class SomeClass {
2       // class definition goes here
3   }
4   struct SomeStructure {
5       // structure definition goes here
6   }
```

Neue Klasse/Struktur → neuer Typ

Klassen, Strukturen

- Definition

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

Deklaration und Initialisierung

Neuer Typ. Großgeschrieben.

Deklaration und Initialisierung. Aufruf des struct-Initialisierers.

Deklaration und Initialisierung

- Verwendung/Erzeugung von Instanzen

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

- Gezielte Member-Initilaisierung bei Structs

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

Strukturen und Enumerations sind Wertetypen (Value Types)

- Sie werden immer kopiert.

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
cinema.width = 2048
  print("cinema is now \(cinema.width) pixels wide")
```

Ausgabe??

```
   print("hd is       \(hd.width) pixels wide")
```

Ausgabe??

- Genauso werden auch die Enumerations immer kopiert. Immer!

- Sie werden immer kopiert.

```
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       \(hd.width) pixels wide")
// prints "hd is       1920 pixels wide"
```

- Genauso werden auch die Enumerations immer kopiert. Immer!

Klassen sind Referenztypen (Reference Types)

- Objekte von Klassen werden nicht kopiert. Es wird immer der Zeiger auf das Objekt einer Klasse referenziert.

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

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

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

Ausgabe??

- Mit „===“ und „!===“ fragen wir ob es sich um identische Objekte handelt.

Klassen sind Referenztypen (Reference Types)

- Klassen werden nicht kopiert. Es wird immer der Zeiger auf das Objekt einer Klasse referenziert.

```
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"
```

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

- Mit „===“ und „!===“ fragen wir ob es sich um identische Objekte handelt.

Beispiel

```
1  enum CompassPoint {
2      case north, south, east, west
3  }
4  var currentDirection = CompassPoint.west
5  let rememberedDirection = currentDirection
6  currentDirection = .east
7  if rememberedDirection == .west {
8      print("The remembered direction is still .west")
9  }
```

Ausgabe?

```swift
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"
```

Properties

- Variablen und Konstanten, die zu Klassen und Strukturen gehören (Attribute)
- Stored Properties
  - Properties, die bestimmte Werte haben
  - Einfache Konstanten und Variablen innerhalb der Klasse
- Computed Properties
  - Properties, die auf andere Properties zugreifen und bestimmte Berechnungen durchführen.

```
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

Lazy stored Properties

- Sie sollen nur dann initialisiert werden, wenn man diese wirklich braucht. Ihre Initialisierung ist zu teuer.

```
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"
    // und hier kommt ein sehr aufwendiger Importprozess

}

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

Lazy stored property

Importer wurde nicht gesetzt.

```
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

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

..Erst hier.

Computed Properties

- Klassen, Strukturen und Enumerations können Properties generieren, die keine einfachen Werte besitzen. Ihre Werte werden berechnet.
- Sie stellen getter- und optional setter-Funktionalitäten zur Verfügung.
- Properties ohne setter sind „read only“

Beispiel

Punkt mit zwei Koordinaten

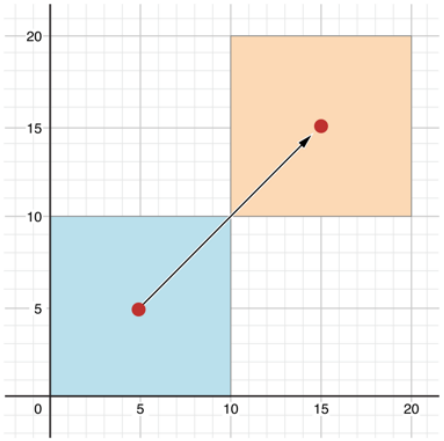
Größe (Höhe, Breite)

Rechteck.
Anfang (unten links),
Größe,
Zentrum – Computed Property.

getter, setter

```
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 ohne setter sind „read only“



Read-Only Computed Proterties

- Properties ohne Setter.
- Wenn es nur Getter gibt, kann dieses Schlüsselwort ausgelassen werden.

```
1  struct Cuboid {
2      var width = 0.0, height = 0.0, depth = 0.0
3      var volume: Double {
4          return width * height * depth
5      }
6  }
7  let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0,
               depth: 2.0)
8  print("the volume of fourByFiveByTwo is \
               (fourByFiveByTwo.volume)")
9  // Prints "the volume of fourByFiveByTwo is 40.0"
```

Property Observer

- *willSet*
- *didSet*

```
1  class StepCounter {
2      var totalSteps: Int = 0 {
3      willSet(newTotalSteps) {
4          print("About to set totalSteps to \
           (newTotalSteps)")
5      }
6      didSet {
7          if totalSteps > oldValue  {
8              print("Added \(totalSteps - oldValue) steps")
9          }
10     }
11     }
12 }
13 let stepCounter = StepCounter()
14 stepCounter.totalSteps = 200
15 // About to set totalSteps to 200
16 // Added 200 steps
17 stepCounter.totalSteps = 360
18 // About to set totalSteps to 360
19 // Added 160 steps
20 stepCounter.totalSteps = 896
21 // About to set totalSteps to 896
22 // Added 536 steps
```

Property Observer

- *willSet*
- *didSet*

Der Benutzer möchte den Wert des Property auf einen neuen Wert setzten. Diesen Wert haben wir „newTotalSteps“ genannt. Wenn wir da nichts geschrieben hätten, würde es „newValue“ heißen (default).

Hier ist der alte Wert nicht umdefiniert worden und er heißt „oldValue“

```
1  class StepCounter {
2      var totalSteps: Int = 0 {
3          willSet(newTotalSteps) {
4              print("About to set totalSteps to \
                 (newTotalSteps)")
5          }
6          didSet {
7              if totalSteps > oldValue  {
8                  print("Added \(totalSteps - oldValue)
                 steps")
9              }
10         }
11     }
12 }
13 let stepCounter = StepCounter()
14 stepCounter.totalSteps = 200
15 // About to set totalSteps to 200
16 // Added 200 steps
17 stepCounter.totalSteps = 360
18 // About to set totalSteps to 360
19 // Added 160 steps
20 stepCounter.totalSteps = 896
21 // About to set totalSteps to 896
22 // Added 536 steps
```

```swift
import UIKit
var str = "Hello, playground"
struct Point {
    var x = 0.0, y = 0.0

}
var somePoint = Point()
somePoint.x = 7.0
print("somePoint.x=\(somePoint.x)")
```

OK?

```swift
import UIKit
var str = "Hello, playground"
struct Point {
    var x = 0.0, y = 0.0

}
var somePoint = Point()
somePoint.x = 7.0
print("somePoint.x=\(somePoint.x)")
```

JA, das ist ok

```
import UIKit
var str = "Hello, playground"
struct Point {
    var x = 0.0, y = 0.0

}
let somePoint = Point()
somePoint.x = 7.0
print("somePoint.x=\(somePoint.x)")
```

Wieso ist das nicht ok?

Type properties

- Ähnlich wie Klassenvariablen in Java
- Variablen, die zu einem Typ gehören.
- Sie Sind über den Typ zugreifbar.

Type property einer Struktur. „static“

Type property eines Aufzählungstyps. „static“

Type property einer Klasse. „class “

```
1  struct SomeStructure {
2      static var storedTypeProperty = "Some value."
3      static var computedTypeProperty: Int {
4          return 1
5      }
6  }
7  enum SomeEnumeration {
8      static var storedTypeProperty = "Some value."
9      static var computedTypeProperty: Int {
10         return 6
11     }
12 }
13 class SomeClass {
14     static var storedTypeProperty = "Some value."
15     static var computedTypeProperty: Int {
16         return 27
17     }
18     class var overrideableComputedTypeProperty: Int {
19         return 107
20     }
21 }
```

Type Properties

```
struct SomeStructure {
    static var storedTypeProperty = "Some
        value."
    static var computedTypeProperty: Int {
    // return an Int value here
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some
        value."
    static var computedTypeProperty: Int {
    // return an Int value here
    }
}
class SomeClass {
    class var computedTypeProperty: Int {
    // return an Int value here
    }
}
```

Die Properties können dann gesetzt und gelesen werden.

```
println(SomeStructure.storedTypeProperty)
// prints "Some value."
SomeStructure.storedTypeProperty = "Another
        value."
println(SomeStructure.storedTypeProperty)
// prints "Another value."
```

Methoden

- Instanzmethoden
- Typmethoden

Instanzmethoden


```swift
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

```swift
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.increment(by: 5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

Self

```
1   func increment() {
2       self.count += 1
3   }
```

Modifikation von „Value Types“ durch Methoden

- Mit „mutating“-Methoden

```
1   struct Point {
2       var x = 0.0, y = 0.0
3       mutating func moveBy(x deltaX: Double, y deltaY:
            Double) {
4           x += deltaX
5           y += deltaY
6       }
7   }
8   var somePoint = Point(x: 1.0, y: 1.0)
9   somePoint.moveBy(x: 2.0, y: 3.0)
10  print("The point is now at (\(somePoint.x), \
            (somePoint.y))")
11  // Prints "The point is now at (3.0, 4.0)"
```

Subscript

- Shortcuts für den Zugriff auf bestimmte Elemente von Collections, Listen und Sequenzen.
- Syntax

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript
        value here
    }
    set(newValue) {
        // perform a suitable setting action
        here
    }
}
```

Es können beliebige Parameter- und Rückgabetypen definiert werden.

.. Oder ohne set

```
subscript(index: Int) -> Int {



}
```

Beispiel

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)

print("six times three is \
      (threeTimesTable[6])")
// prints "six times three is 18"
```

Vererbung

- Klassen in Swift können Methoden, Properties und andere Eigenschaften erben.

Beispiel

```
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
    }
}
```

```
let someVehicle = Vehicle()
println("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

Unterklasse

„..ist Unterklasse von..“

```
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
}
```

```
let bicycle = Bicycle()

bicycle.hasBasket = true
bicycle.currentSpeed = 15.0

println("Bicycle: \(bicycle.description)")

// Bicycle: traveling at 15.0 miles per hour
```

Unterklasse

```
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
}
```

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```

```
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
println("Tandem: \(tandem.description)")
// Tandem: traveling at 22.0 miles per hour
```

Überschreiben (Overriding)

- Neudefinition von geerbten Methoden und Properties
- muss mit „override“ gekennzeichnet werden.
- die neue Methode *someMethod* der Unterklasse kann auf die Originalmethode der Superklasse mit *super.someMethod* zugreifen.

Beispiel, Methode überschreiben

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

```
let train = Train()
train.makeNoise()
// prints "Choo Choo"
```

Beispiel, Property überschreiben

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

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
println("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

Property Observer überschreiben

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

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

```
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
    print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

Schutz vor dem Überschreiben

- *final* in unterschiedlichen Varianten
- final var, final func, final class func, und final subscript
- final class
  - heißt, dass die Klasse keine Unterklasse haben darf.
  - Keine Vererbung möglich.
  - Führt zum Compiler-Fehler

Quellen

Die meisten Sourcecode-Beispiele und die Sprachdefinition der Sprache Swift wurden aus:

Aple Inc. „The Swift Programming Language.“ iBooks. https://itun.es/de/jEUH0.l

genommen.

Eventuelle andere Quellen bzw. eigene Beispiele werden an den entsprechenden Stellen direkt angegeben bzw. gekennzeichnet.