

iOS Modul

Enumeration & Optional



Enumeration

- Eine weitere Datenstruktur neben class und struct.
- Enums haben ausschliesslich diskrete Werte
- Wie ein struct ist ein enum ein value type.

```
enum FastFoodMenuItem {  
    case hamburger  
    case fries  
    case drink  
    case cookie  
}
```

```
enum FastFoodMenuItem {  
    case hamburger, fries, drink, cookie  
}
```

Enumeration

- Jeder Fall kann zugeordnete Daten (Associated Data) besitzen.

```
enum FastFoodMenuItem {  
    case hamburger(numberOfPatties: Int)  
    case fries(size: FryOrderSize)  
    case drink(String, ounces: Int) //String für Marke: «Coke»  
    case cookie  
}
```

- Fries hat eigenen zugeordneten Enum
- Drink hat zwei zugeordnete Daten, der erste ist unbenannt.

```
enum FryOrderSize {  
    case large, medium, small  
}
```

Enumeration

- Verwendung eines Enums durch Angabe des Enumbezeichners gefolgt von einem Punkt und dann dem entsprechenden Fall:

```
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = FastFoodMenuItem.cookie
var drink: FastFoodMenuItem = .drink("Coke", ounces: 32)
var burger = FastFoodMenuItem.hamburger(patties: 2)
```

Enumeration

- Prüfung des konkreten Falles wird über ein switch-Statement umgesetzt:

```
var menuItem = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem {
    case .hamburger: print("burger")
    case .fries: print("fries")
    case .drink: break
    case .cookie: print("cookie")
}
```

- Im Beispiel werden die zugeordneten Daten ignoriert.
- Im Beispiel wäre die Ausgabe: „burger“.
- Im Fall von .drink würde nichts ausgegeben werden
- Da der Enum klar definiert ist, muss nicht `FastFoodMenuItem.fries:` geschrieben werden.

Enumeration

- Der Zugriff auf die zugeordneten Daten erfolgt über die let-Syntax

```
var menuItem = FastFoodMenuItem.drink("Coke", ounces: 32)
switch menuItem {
case .hamburger(let pattyCount): print("burger w. \$(pattyCount) patties!")
case .fries(let size): print("a \$(size) order of fries!")
case .drink(let brand, let ounces): print("a \$(ounces)oz \$(brand)")
case .cookie: print("a cookie!") }
```

- Der mit let vergebene Variablenname darf von der in der Definition des enums abweichen
 - Vgl.: `numberOfPatties` und `pattyCount`

Enumeration

- Enums können Funktionen und berechnete Variablen besitzen. Aber keine gespeicherten Variablen.

```
enum FastFoodMenuItem {  
    case hamburger(numberOfPatties: Int)  
    case fries(size: FryOrderSize)  
    case drink(String, ounces: Int)  
    case cookie  
  
    func isIncludedInSpecialOrder(number: Int) -> Bool { }  
    var calories: Int { //switch-Statement (self) und Berechnung}  
}
```

Enumeration

- Beispiel zu switch-statement und self:

```
func isIncludedInSpecialOrder(number: Int) -> Bool {  
    switch self {  
        case .hamburger(let pattyCount): return pattyCount == number  
        case .fries, .cookie: return true  
        case .drink(_, let ounces): return ounces == 16  
    }  
}
```

- Falls zugeordnete Daten nicht relevant sind, werden sie mit Unterstrich ignoriert (.drink)

Enumeration

- Zum Iterieren über alle Fälle eines enums muss dieser CaseIterable sein. Dadurch besitzt der Enum die statische Variable allCases:

```
enum TeslaModel: CaseIterable {  
    case X, S, Three, Y  
}
```

```
for model in TeslaModel.allCases {  
    reportSalesNumbers(for: model)  
}  
  
func reportSalesNumbers(for model: TeslaModel) {  
    switch model { ... }  
}
```

Enumeration

- Es ist auch möglich den Fällen einen Rohwert zuzuordnen

```
enum ASCIIControlCharacter: Character {  
    case tab = "\t"  
    case lineFeed = "\n"  
    case carriageReturn = "\r"  
}
```

```
enum Planet: Int {  
    case mercury = 1  
    case venus //implicit 2  
    case earth = 7  
    case mars // implicit 8  
    ...  
}
```

```
enum CompassPoint: String {  
    case north  
    case south //raw value: "south"  
    case east  
    case west  
}
```

Optional

- Ein Optional ist ein sehr weit verbreiteter enum in Swift

```
enum Optional<T> {  
    case none //kein Wert vorhanden  
    case some(<T>) // zugeordnete Daten vorhanden vom Typ T  
}
```

- Optionals drücken aus, dass eine Variable keinen Wert haben muss (nil) und zwingen dazu diesen Fall zu behandeln. Dadurch werden Programme deutlich robuster.
- In Swift wurde zur einfachen Verwendung von Optionals „Syntactic Sugar“ eingeführt

```
var hello: String?          var hello: Optional = .none  
var hello: String? = "hello" var hello: Optional = .some("hello")  
var hello: String? = nil    var hello: Optional = .none
```

Optional

- Um etwas vom Typ `Optional<T>` zu deklarieren wird die Syntax `T?` benutzt
- Diesem kann man den Wert `nil` zuordnen
- Oder auch einen Wert vom Typ `T` (`Optional.some` mit zugeordneten Daten)

```
var hello: String?          var hello: Optional = .none  
var hello: String? = "hello" var hello: Optional = .some("hello")  
var hello: String? = nil    var hello: Optional = .none
```

Optional

- Auf die zugeordneten Daten kann auf verschiedene Arten zugegriffen werden:
 - Wenn man sehr sicher ist, dass es einen zugeordneten Wert gibt mit dem Force Unwrap Operator: !
 - Ist kein Wert vorhanden führt dies zu einem Crash
 - Oder mit einer Prüfung auf einen vorhandenen Wert“

```
let hello: String? = nil
print(hello!) //Crash
```

```
let hello: String? = nil
if let safehello = hello {
    print(safehello)
}
else {
    print("no value set")
}
```

Optional

- Ein optionaler Standardwert kann mit dem ?? Operator vergeben werden:

```
let x: String? = ...  
let y = x ?? "foo"
```