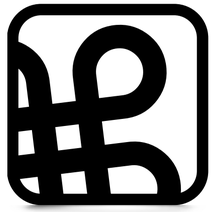


Macoun

Schwere Typen

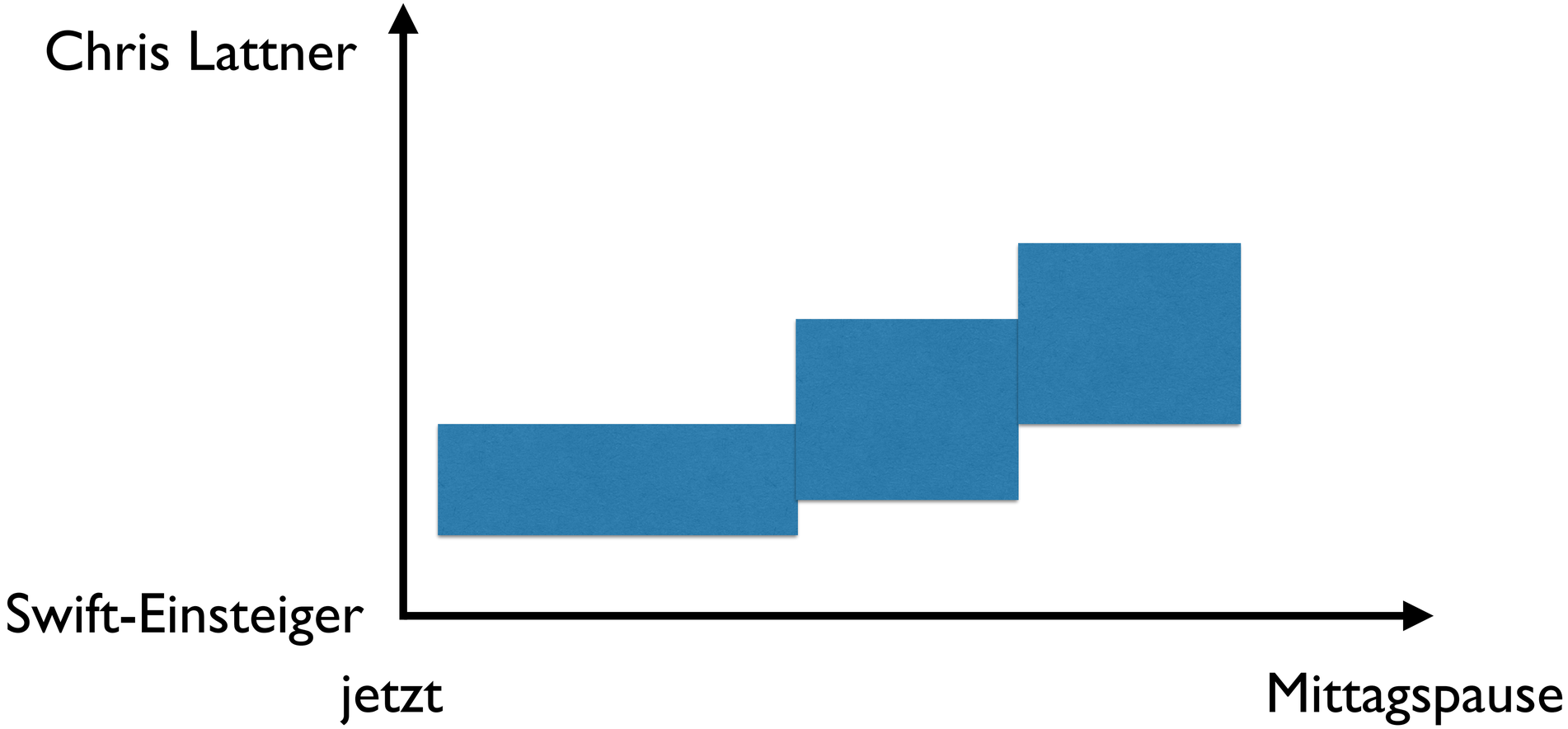
Nikolaj Schumacher

Zielpublikum

Einsteiger bis Fortgeschrittene

Mindestvoraussetzung: Swift-Syntax lesen

Zielpublikum



Ablauf

- statisch & dynamisch
- Swift-Typen
- Generics
- Bonus Features

Code

https://github.com/nschum/macoun2015

- Folien
- Playgrounds

(URL wird am Ende wiederholt)

Prolog

dynamisch & statisch

statisch

- C++
- Java
- C#
- Pascal
- Swift

dynamisch

- Ruby
- Python
- Javascript
- Objective-C

```
x.length()
```

statische Sprachen

Compiler kennt:

- Typ der Variable*
- aufgerufene Funktion*

```
x.length()
```

(bei Polymorphie: Menge von Typen/Funktionen)

dynamische Sprachen

Compiler kennt:

- Namen der Funktion
- (sonst nichts)

```
x.length()
```

Was macht Obj-C dynamisch?

- respondsToSelector
- NSInvocation
- Method Swizzling

statisch …

- mehr Überprüfung durch Compiler
- Typ-Herleitung
- Optimierung (Inlining, final, whole module optimization)

- kein Runtime-Hacking
- Komplexität (Hallo Generics!)

Grundlagen

Swift-Typen

Class

- Referenz-Typen
- ARC
- „auf dem Heap”
- Polymorphie

Beispiele Class

- Cocoa: `CALayer`
- Swift-Stdlib: ? (fast nichts)

Swift & Obj-C Class

- Standard: statischer Dispatch (vtable)
- keine Root-Klasse
- kein automatisches `isEqual`, `hash`

Struct

- Value-Typen
- Kopien statt ARC
- „auf dem Stack”
- keine Vererbung (aber Protokolle)

Beispiele Struct

- Cocoa: `CGRect`
- Swift-Stdlib: `Int`, `String`, `Array`, `Dictionary` (… fast alles)

Besonderheiten Struct

- kein „Copy-Constructor“
- kein deinit
- immutable

Swift & Obj-C Structs

- Methoden
- Protokolle
- passen in Arrays und Dictionaries
- direktes Manipulieren von Properties sicher

Klasse oder Struct

Goldene Regel:

Wenn == sinnvoll, dann Struct

Tupel

- namenlose Structs

```
(name: "x", wert: "y")

struct MeinTupel {
    var name = "x"
    var wert = "y"
}
```

Enums

- „raw value” Enums → Konstanten
- „associated value“ Enums → Tupel „mit Polymorphie“

Swift & Obj-C Enums

- „raw value” Enums → `NS_ENUM`
- „associated value“ Enums ersetzen Hierarchien trivialer Klassen

Optionals

- associated Value Enum

```
public enum Optional<T> {
    case None
    case Some(T)
}
```

Closures

- siehe Objective-C
- `__block → var`

Playgrounds

- Properties
- Immutable
- Klassen in Structs

Generics

(dramatische Musik entfällt wegen GEMA)

Generics

```
struct Array {
    func objectAtIndex(index: Int) -> Any
}
```

```
struct Array<T> {
    func objectAtIndex(index: Int) -> T
}
```

```
let myArray: Array<Int>
```

Generics

```
func max(a: Any, b: Any) -> Any {
    …
}
```

```
func max<T: Comparable>(a: T, b: T) -> T {
    …
}
```

Warum?

- Typ-Herleitung sonst nicht möglich
  - mehr explizite Typen
  - mehr explizite Casts
- mehr Optimierungen

Associated Types

andere Schreibweise bei Protokollen

```
protocol Sequence {
    func objectAtIndex(index: Int) -> Any
}
```

```
protocol Sequence {
    typealias T
    func objectAtIndex(index: Int) -> T
}
```

Associated Types

Implementierung des Protokolls

```
struct MyDictionary<Key: Hashable, Value>
    : CollectionType, Indexable, SequenceType, DictionaryLiteralConvertible {

    typealias Element = (Key, Value)
    typealias Index = DictionaryIndex<Key, Value>
}
```

Associated Types

Zum Vergleich: „normale“ Schreibweise

kein echtes Swift!

```
public struct MyDictionary<Key: Hashable, Value>
    : CollectionType<Index: (Key, Value)>,
      Indexable<Index: DictionaryIndex<Key, Value>,
      SequenceType<Generator: IndexingGenerator<Self>,
                   SubSequence: Slice<Self>>,
      DictionaryLiteralConvertible<Key: Key, Value: Value> {

}
```

Associated Types

Sonderfall `Self`

```
public protocol Equatable {
    public func ==(lhs: Self, rhs: Self) -> Bool
}
```

automatischer Alias für implementierenden Typ

Protokolle als Parameter

2 Möglichkeiten

```
func g<T: CustomStringConvertible>(x: T) {
    print(x)
}

func f(x: CustomStringConvertible) {
    print(x)
}
```

Protokolle als Parameter

Möglichkeit 1 erlaubt Einschränkungen

```
func sort<S: SequenceType where S.Generator.Element: Comparable>(t: S) -> S {
    …
}
```

Playgrounds

- Generics
- (Protokoll)
- (Protokoll-Extensions)

Protokolle als Parameter

2 Möglichkeiten

```
func g<T: CustomStringConvertible>(x: T) {
    print(x)
}

func f(x: CustomStringConvertible) {
    print(x)
}
```

Protokolle als Variablen

```
let x: Equatable
```

```
protocol 'Equatable' can only be used as a generic
constraint because it has Self or associated type
requirements
```

Playgrounds

- Problem
- Lösung 1
- Lösung 2

Vererbung

Superklasse	Kindklasse	?
nicht generisch	nicht generisch	✓
nicht generisch	generisch	✓
generisch	nicht generisch	✗
generisch	generisch	✓

Vererbung

Protokoll	Klasse	?
nicht generisch	nicht generisch	✓
nicht generisch	generisch	✓
generisch	nicht generisch	✓
generisch	generisch	✓

Kompatibilität

Zuweisung	?
`let x: Klasse<Parent> = Klasse<Child>()`	✗
`let x: Klasse<Child> = Klasse<Parent>()`	✗
`let x: Struct<Parent> = Struct<Child>()`	✗
`let x: Struct<Child> = Struct<Parent>()`	✗

Kompatibilität

Zuweisung	?
`let x: Array<Parent> = Array<Child>()`	✓

- Arrays sind magisch!
- zum Glück
- Only Apple …

Kompatibilität

Zuweisung	?
`let f: Child -> () = { (x: Super) in }`	✓
`let f: () -> Super = { return Child() }`	✓
`let f: Super -> () = { (x: Child) in }`	✗
`let f: () -> Child = { return Super() }`	✗

Kompatibilität

Zuweisung	?
`let f: Int -> () = { (x: Int?) in }`	✓
`let f: () -> Int? = { return 0 }`	✓

Bonus Features

Optionals & Comparable

Optionals sind (magischerweise) Comparable,
wenn ihre inneren Typen es sind

Achtung!

```
nil < 5 // → true
```

`Void`? WTF?

- `Void` ist nichts.
- `Void?` ist vielleicht nichts?
- Warum existiert das …?

Was ist eigentlich …?

Schlüsselwort	Bedeutung
Void	()
AnyObject	(implizite Root-Klasse)
Any	protocol<>

Property-Spezialisierung

```
@interface MyViewController: NSViewController {
}

@implementation MyViewController {
    - (MyView *)view {
        return (MyView *)super.view
    }
}
```

Property-Spezialisierung

```
class MyViewController: NSViewController {
    override var view: MyView {
        get {
            super.view as! MyView
        }
        set {
            super.view = newValue
        }
    }
}
```

```
Cannot override mutable property
'view' of type 'NSView' with
covariant type 'MyView'
```

Property-Spezialisierung

- Unmöglich nur Getter zu überschreiben
- Setter können nicht spezialisiert werden

```
let vc: NSViewController = MyViewController()

vc.view = NSView()
```

```
NSViewController erlaubt NSView
MyViewController erwartet nur MyView.
NSView ist kein MyView.
```

automatische Optionals

```
func orElse<T>(left: T?, _ right: T) -> T {
    if let left = left {
        return left
    } else {
        return right
    }
}

orElse(1, 2) // → 1
orElse(nil, 2) // → 2
```

automatische Optionals

```
func orElse<T>(left: T?, _ right: T) -> T {
    if let left = left {
        return left
    } else {
        return right
    }
}

let i: Int? = nil
let j: Int? = 5

orElse(i, j) // → nil
```

T ← Int?

automatische Optionals

```
func orElse(left: Int??, _ right: Int?) -> Int? {
    if let left = left {
        return left
    } else {
        return right
    }
}

let i: Int? = nil
let j: Int? = 5

orElse(.Some(nil), .Some(j)) // → .Some(nil)
```

T ← Int?

Playgrounds

- (Void)
- (automatische Optionals)

Swift-Tipps

- unverständlicher Compile-Fehler?
  explizite Typen ausprobieren
- markante Farbe für „Project Instance Variables and Globals”
  (`self.` ist optional)

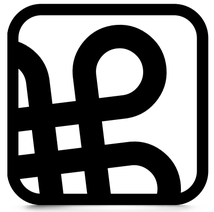
Fragen?

Werbung

- https://github.com/nschum/macoun2015
- http://swiftsandbox.io

- CocoaHeads Frankfurt
  - erster Montag im Monat
  - http://www.meetup.com/CocoaHeadsFFM/

Vielen Dank



Macoun