Schwere Typen

Nikolaj Schumacher

~~Objective~~ C++ without the C

jetzt mit etwas Python

Typen

grundlegende Typen

- Klassen
- Structs
- Enums
- Protokolle

- Extensions?
- Tupel
- Optionals
- Closures
- Void
- …

Klassen und Structs

Klassen

- Referenzen
- ARC (nicht GC)
- (Einfach-)Vererbung
- Polymorphie
  (auch statisch)

Structs

- Value-Typen
- Copy-on-Write
- Immutable
- keine Vererbung

Klasse oder Struct?

- Daumenregel:
- Struct genau dann wenn Equatable
- Referenz braucht kein Equatable (===)

Enums

Raw

```
enum E : Integer {
    case first
    case second
}
```

Tagged

```
enum E {
    case first(Int)
    case second(String)
}
```

Enums

Raw

- Konstanten
- beliebiger Typ für Konstanten
- Hashable/Equatable

Tagged

- Instanzen
- ersetzt simple Typ-Hierarchien
- switch statt Polymorphie

Protokolle

- Schnittstelle ohne Implementierung
- generisch/nicht-generisch
- für Klassen, Structs und Enums
- „Klassen-Protkolle“ nur für Klassen

Klassen-Protokolle

```
protocol P: class {
    …
}
```

Extensions

Erweiterung anderer Typen

- zusätzliche Methoden
- zusätzliche Protokolle
- statischer Dispatch
- nicht überschreibbar

Default-Implementierung Protokoll

- überschreibbar durch:
  - tatsächliche Implementierung
  - Kind-Protokolle

Erweiterung

```
protocol P {}
class C : P {}

extension C { func foo() { print("C") } }
extension P { func foo() { print("P") } }

let c: C = C()
c.foo() // C

let p: P = c
p.foo() // P
```

Default Implementierung

```
protocol P { func foo() }
class C : P {}

extension C { func foo() { print("C") } }
extension P { func foo() { print("P") } }

let c: C = C()
c.foo() // C

let p: P = c
p.foo() // C
```

Typ-Aliase

- alternative Namen
- austauschbar
- keine zusätzliche Typsicherheit

```
typealias A = Int

var a: A
var int: Int

a = int ✔
int = a ✔
```

Tupel

- Structs
  - ohne Namen
  - ohne Methoden
  - ohne Protokolle

```
let t = (foo: 0, bar: 1)

print(t.foo)
print(t.bar)
```

Property-Namen

Optionals

`nil` ist standardmäßig nicht erlaubt

```
Int?
UIView?
((Int?, Double?) -> Void)?
```

Optionals

```
enum Optional<T> {
    case none // nil
    case some(T)
}
```

Void

„kein“ Rückgabewert

```
func foo() -> Void {}
let void: Void
let optional: Void?
```

Void

```
typealias Void = ()
```

`Void?` enthält also entweder `()` oder `nil`

Closures

- Code als Variable
- `@escaping` speichert Kontext
- `@nonescaping` nur als Parameter (Macro)

AnyObject

- implizites Basis-Protokoll für Klassen

native Typen

```
struct Int {
    …
}
```

Generics

Generics

- Generische Parameter
- Associated Types
- (Self)

Generische Parameter

```
class Collection<Element> {
    func foo<T>(t: T) -> Element {…}
}

let c = Collection<String>
c.foo(t: Int)

let c2: Collection<String> = Collection()
```

Associated Types

```
protocol Collection {

    associatedtype Element

    func foo() -> Element

}
```

Associated Types

```
protocol Collection {
    associatedtype Element
    func foo<T>(t: T) -> Element
}

class MyCollection {
    typealias Element = Int
    func foo<T>(t: T) -> Element
}
```

Associated Types

```
protocol Collection {
    associatedtype Element
    func foo<T>(t: T) -> Element
}

class MyCollection {
    struct Element {}
    func foo<T>(t: T) -> Element
}
```

Self

`Self` ist ein Platzhalter für implementierenden Typ

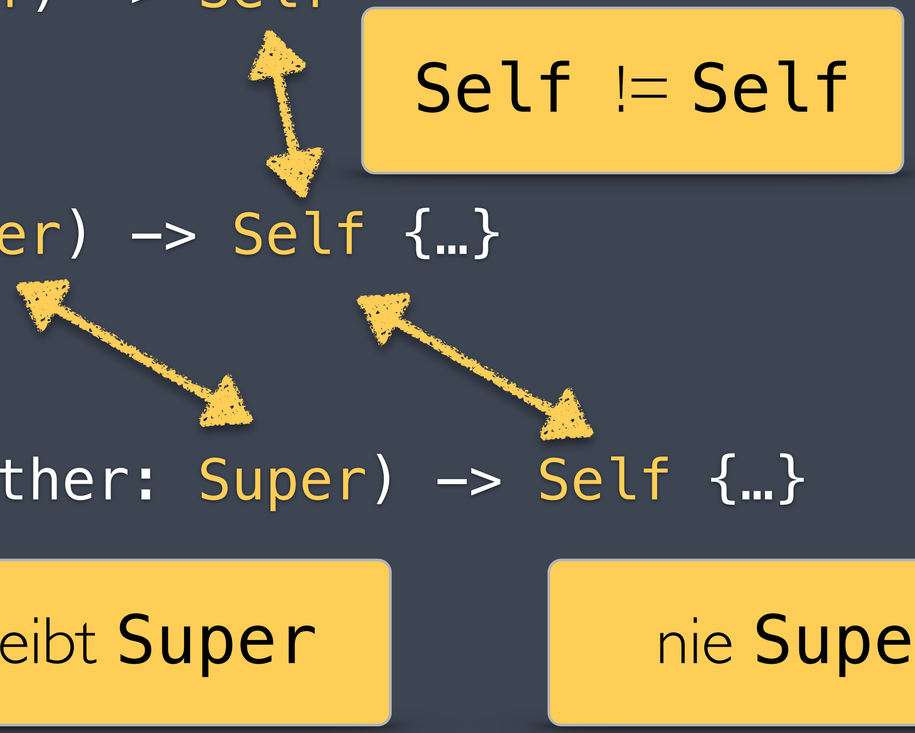
```
protocol Collection {
    func append(other: Self) -> Self
}
```

Self

```
protocol Collection {
    func append(other: Self) -> Self
}

final class MyCollection : Collection {
    func append(other: MyCollection) -> MyCollection {…}
}
```

Kindklasse muss Protokoll erfüllen

```
protocol Collection {
    func append(other: Self) -> Self
}
class Super : Collection {
    func append(other: Super) -> Self {…}
}
class Child : Super {
    override func append(other: Super) -> Self {…}
}
```



Constraints

```
class Collection<Element> where Element: Hashable {
    func foo<T>(t: T) -> Element where T: Equatable {…}
}

extension P where Self: Equatable {…}
```

Verschachtelung

```
protocol Collection {
    associatedtype Element
}

func sort<C: Collection>(c: C)
    where C.Element: Equatable {…}
```

Conditional Conformance

```
protocol Collection {
    associatedtype Element
}

extension Collection: Equatable where Element: Equatable {…}
```

Conditional Conformance

- `Optional: Equatable`
- Tupel?

Tupel haben keine Protokolle …

```
@inlinable public func == <A, B, C, D, E, F>(
    lhs: (A, B, C, D, E, F),
    rhs: (A, B, C, D, E, F))
    -> Bool

    where A : Equatable, B : Equatable, C : Equatable,
          D : Equatable, E : Equatable, F : Equatable {…}
```

Standardbibliothek

Warum Associated Types

```
protocol Sequence<Element>
    where Element: Equatable {…}

protocol Collection<Element> : Sequence<Element>
    where Element: Equatable {…}

protocol IndexedCollection<Index, Element> : Collection<Element>
    where Element: Equatable,
          Index: SignedInteger {…}
```

kein echtes Swift!

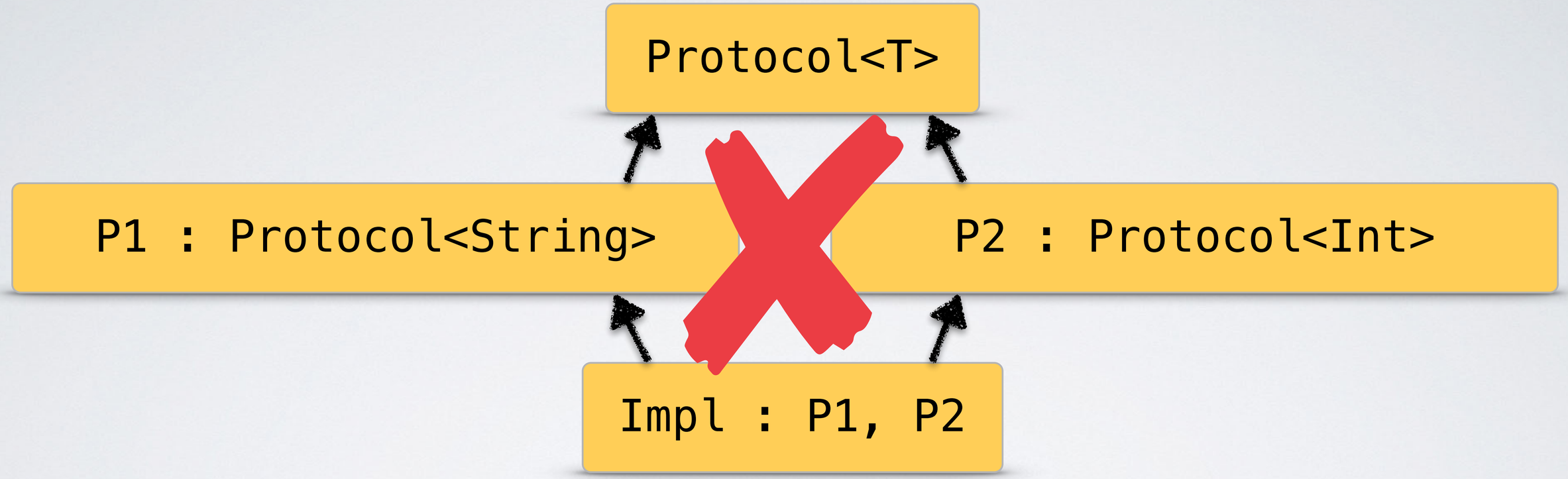
Warum Associated Types

```
protocol Sequence {
    associatedtype Element: Equatable
    …
}

protocol Collection : Sequence {…}

protocol IndexedCollection : Collection {
    associatedtype Index: SignedInteger
    …
}
```

Implizite Ersetzung

```
protocol Collection {
    associatedtype Element
    func foo<T>(t: T) -> Element
}

class MyCollection {
    func foo<T>(t: T) -> Int
}
```

Warum Associated Types



Nachteile von Associated Types?

Protokolle

Protokoll-orientierte Programmierung (POP)

Protokolle statt Klassen

POP

```
func countUsages(of x: Int, in elements: [Int]) -> Int {
    var n = 0
    for element in elements {
        if element == x {
            n = n + 1
        }
    }
    return n
}
```

POP

```
func countUsages(of x: Equatable, in elements: [Equatable]) -> Int {
    var n = 0
    for element in elements {
        if element == x {
            n = n + 1
        }
    }
    return n
}
```

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

Warum nicht?

POP

```
func countUsages(of x: Equatable, in elements: [Equatable]) -> Int {
    var n = 0
    for element in elements {
        if element == x {
            n = n + 1
        }
    }
    return n
}
```

Type Existential

Type Existential

- Platzhalter für einen echten Typ

Type Existential

```
protocol Equatable {

    static func ==(left: Self, right: Self)

}
```

muss ersetzt werden durch konkreten Typ

Type Existential

- Platzhalter für einen echten Typ
- Equatable ist aber nicht ein Typ
- unterschiedliche Methodensignatur für jede Implementierung

Type Existential

- `[Equatable]` könnte `Int` und `String` enthalten
- `func ==(left: Int, right: Int)` ✓
- `func ==(left: String, right: String)` ✓
- `func ==(left: Int, right: String)` ✗
- `func ==(left: String, right: Int)` ✗

Lösung 1: Generics

```
func countUsages<T: Equatable>(of x: T, in elements: [T]) -> Int {
    var n = 0
    for element in elements {
        if element == x {
            n = n + 1
        }
    }
    return n
}
```

Lösung 1: Generics

```
func countUsages<T: Equatable, S: Sequence>(
    of x: T, in elements: S) -> Int
    where S.Element == T {

    var n = 0
    for element in elements {
        if element == x {
            n = n + 1
        }
    }
    return n
```

Lösung 1: Generics

```
func countUsages<T: Equatable, S: Sequence, R: SignedInteger>(
    of x: T, in elements: S) -> R
    where S.Element == T {

    var n: R = 0
    for element in elements {
        if element == x {
            n = n + 1
        }
    }
    return n
```

Aufruf

```
let result: Int8 = countUsages(0, in: [0, 0, 1, 1, 2])
```

R = Int8

T = Int

S = [Int]

(nächster Schritt)

```
extension Sequence where Element: Equatable {
    func countUsages<R: SignedInteger>(of x: Element) -> R {
        var n: R = 0
        for element in self {
            if element == x {
                n = n + 1
            }
        }
        return n
    }
}
```

Lösung 2: Type Erasure

Lösung 2: Type Erasure

```
struct AnyEquatable: Equatable {
    let value: Any
    let equatable: (Any) -> Bool

    public init<T: Equatable>(_ value: T) {
        self.value = value
        equatable = {
            guard let arg = $0 as? T else { return false }
            return true
        }
    }
```

Lösung 2: Type Erasure

```
func ==(left: AnyEquatable, right: AnyEquatable) -> Bool {
    left.equatable(right.value)
}
```

Lösung 2: Type Erasure

```
func countUsages(of x: AnyEquatable,
                 in elements: [AnyEquatable]) -> Int {
    var n = 0
    for element in elements {
        if element == x {
            n = n + 1
        }
    }
    return n
}
```

Lösung 2: Type Erasure

```
let array = [AnyEquatable(0), AnyEquatable("")]
countUsages(of: AnyEquatable(0), in: array)
```

„Lösung“ 3: Opaque Typen

- Swift 5.1
- nur Rückgabewerte/Variablen

Opaque Typen

```
struct View {
    var innerView: some View {…}
}
```

Opaque Typen

- ein Typ, der Protokoll adaptiert
- konkreter Typ unbekannt
- aber: immer derselbe Typ (pro Variable/Methode)
- `any` ebenfalls geplant (siehe Type Erasure)

Covariance

Covariance

```
class Super {}
class Child : Super {}

var super: Super
var child: Child

super = child ✓
child = super ✗
```

Covariance beschreibt das
Verhalten generischer Parameter

Invariance

```
class Super {}
class Child : Super {}
class X<T> {}

var superX: X<Super>
var childX: X<Child>

superX = childX ✗
childX = superX ✗
```

Covariance

```
class Super {}
class Child : Super {}

var superArray: [Super] // Array<Super>
var childArray: [Child] // Array<Child>

superArray = childArray ✔
childArray = superArray ✘
```

Covariance

```
class Super {}
class Child : Super {}

var superOptional: Super? // Optional<Super>
var childOptional: Child? // Optional<Child>

superOptional = childOptional ✔
childOptional = superOptional ✘
```

Covariance

```
class Super {}
class Child : Super {}

var superGetter: () -> Super
var childGetter: () -> Child

superGetter = childGetter ✓
childGetter = superGetter ✗
```

Contravariance

```
class Super {}
class Child : Super {}

var superSetter: (Super) -> Void
var childSetter: (Child) -> Void

superSetter = childSetter ✘
childSetter = superSetter ✔
```

Funktionen

- Closures/Funktions-Referenzen/Methoden-Referenzen
- Rückgabetyp ist *covariant*
- Parameter sind *contravariant*

magische Typen

Covariance

- Array, Set, Dictionary
- Optionals
- Closures
- keine eigenen Typen! (derzeit)

Optionals

```
var int: Int
var intOptional: Int?

int = intOptional ✗
intOptional = int ✓
```

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
```

Optionals

```
func orElse<T>(left: T?, _ right: T) -> T {
    …
}

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

orElse(i, j) // → nil
```

T = Int?, T? = Int??

AnyHashable

```
class C : Hashable {
    func hash(into hasher: inout Hasher) {…}

    static func ==(left: C, right: C) -> Bool {…}
}

let c: C
let anyHashable: AnyHashable = c ✔
```

Never

```
func criticalFailure() -> Never {…}

func test(arg: Int) -> Int {
    guard arg > 0 else { criticalFailure() }
    return 0
}
```

Funktion beendet nie

kein `return` nötig

Literale

```
class C : ExpressibleByIntegerLiteral {
    required init(integerLiteral value: IntegerLiteralType) {
        …
    }
}

let int: Int = 0
let c1: C = int
let c2: C = 0
```

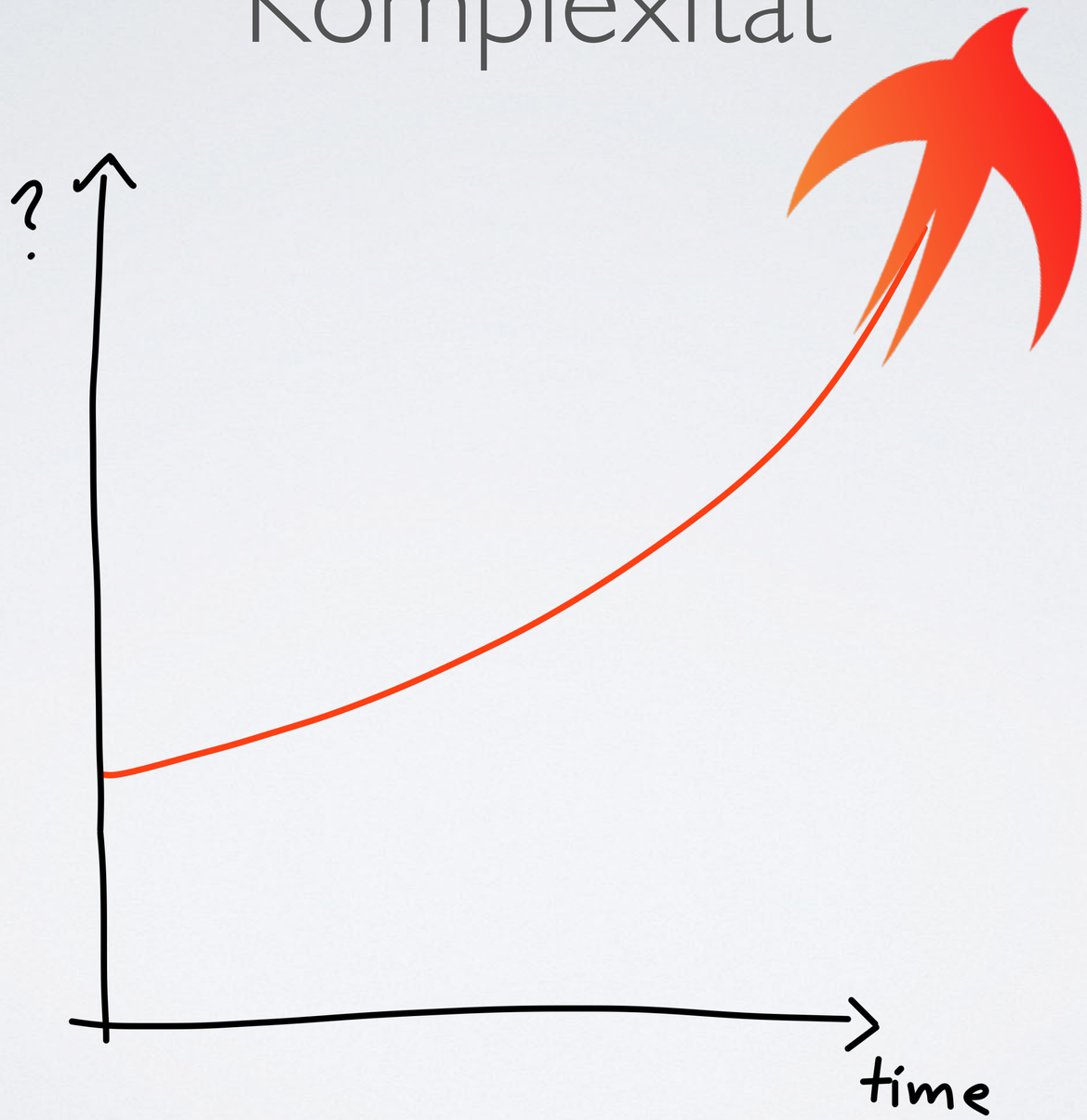
magische Typen
sind oder werden
Sprach-Features

Zusammenfassung

```
// I wish this was JavaScript
var foo: AnyObject!
```

–Nick Lockwood

Komplexität



Fragen?

https://github.com/nschum/talks

https://commons.wikimedia.org/wiki/File:Swift_logo.svg
Public Domain, https://commons.wikimedia.org/wiki/User_talk:Totie