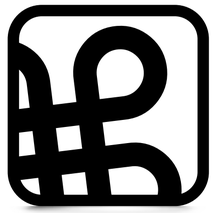


Macoun

Swift fällt aus

Amin Negm-Awad & Christian Kienle

Ablauf

- Orga
- Vorzüge von Swift
- Konzept & Unterschiede
- Fallbeispiele

Orga

- Vorstellung
- Fußballergebnisse

Vorstellung

- Amin
- Chris
- Objective-Cloud

Fußballergebnisse

- Mainz - SAP
- S04 - BVB
- Freiburg - IG Farben
- VfB - Hanoi
- SCP - Bauern

Fußballergebnisse

ERSTER FUSSBALLCLUB KÖLN

fcb

1 - 0

Vorzüge von Swift

- Intentionally left blank

Vorzüge von Swift

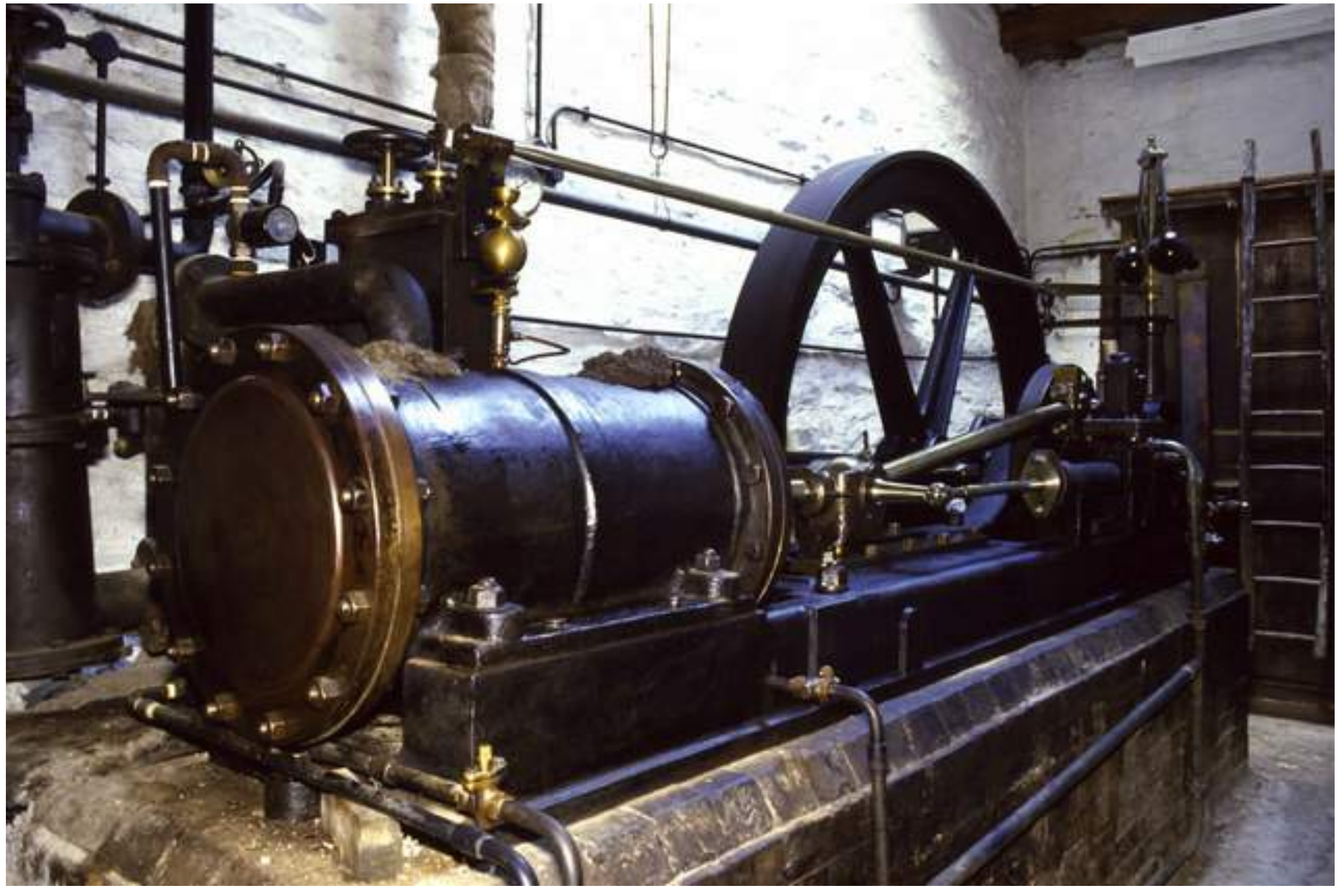
- “safe programming patterns”
- “modern features to make programming…”
  - easier,
  - more flexible,
  - and more fun

Safe

Quelle: Wikipedia / Edward unter Public Domain ohne Einschränkung

- Templates
- Statische Typisierung

Modern



Quelle: Wikipedia / Chris Allen unter CC-BY-SA 2.0

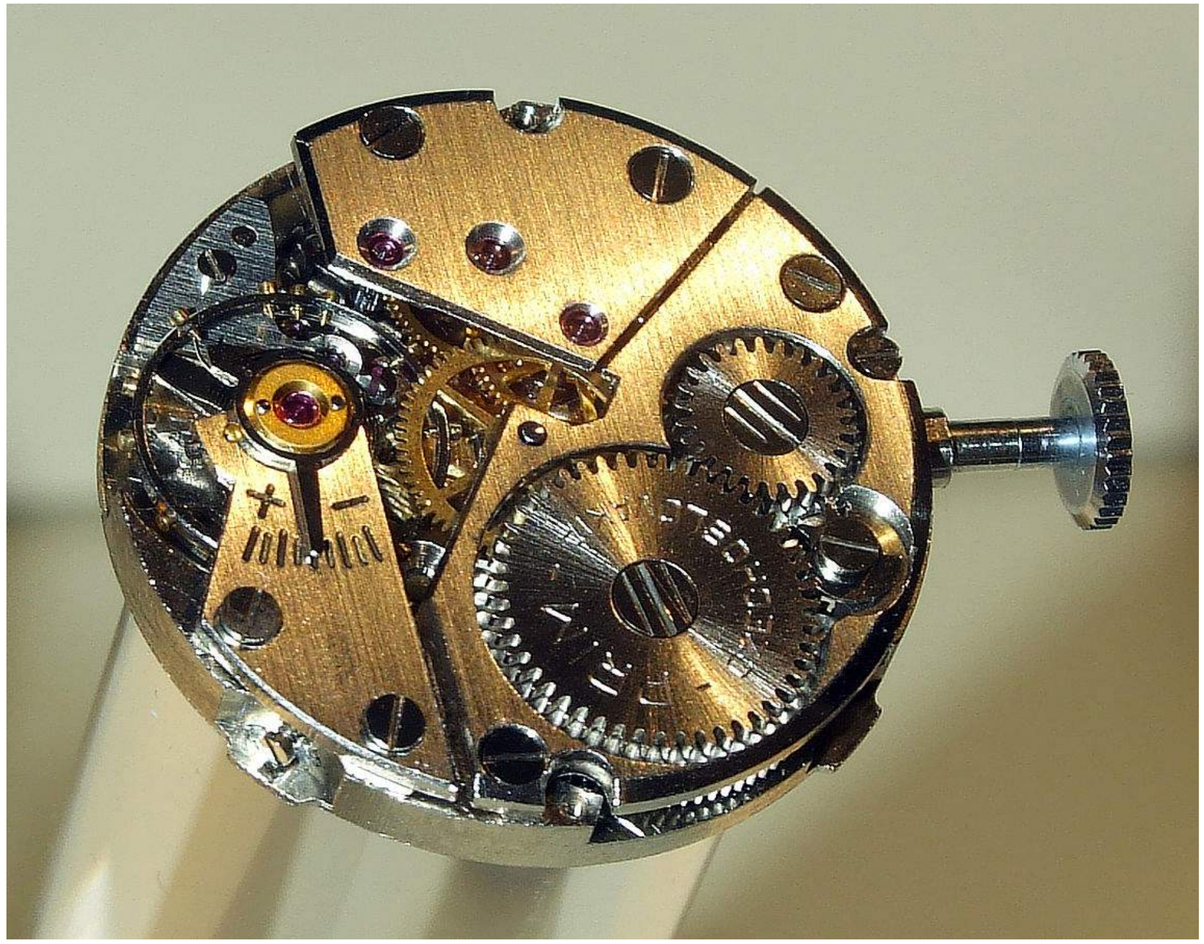
- Operatoren
- Overloading
- Templates
- Typinferenz

More flexible

Quelle: Wikipedia / Eduard Gerlach GmbH unter CC-BY-SA 3.0

- Eigene Operatoren
- Templates
- Overloading

Easier



Quelle: Wikipedia / Kozuch unter CC-BY-SA 3.0

- Gewöhnung
- Operatoren
- Statische Typisierung

More Fun

let 🙀 = "😱"

- ja

Konzepte

- Swift: Statische Typisierung + Templates
- Objective-C: Dynamische Typisierung

Typisierung

- Stark - schwach
- Dynamisch - statisch
- *“OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.” (Alan Kay)*

Statische Typisierung

- Typ steht zur Übersetzungszeit fest
- Versprechen der Sicherheit
- Generischer Code? (Typunabhängige Lösung)
  Lösung I: Templates
  Lösung II: Dynamisierung

Templates

- Compiler schreibt Code für jeden Typ
- Dümmste Lösung

Templates

```
func max<T : Comparable>(x: T, y: T) -> T
{
   return a>b?a:b
}
```

```
// max( 5, 3 )
func max(x: Int, y: Int) -> Int
{
   return a>b?a:b
}
```

Dynamische Typisierung

- Angebliche Unsicherheit
- Laufzeitsystem bindet

Vergleich

- Statische Typisierung + Templates sicherer?
- Dynamische Typisierung mächtiger
  Core Data
  Responder-Chain
  Core Animation
  Undo-Manager
  Proxies
  Grundsätzlich: Mehr als Typentscheidung

Fallbeispiele

- Overloading
- Typinferenz & komplexe Ausdrücke
- Typinferenz & Overloading
- Generische Programmierung
- Optionals
- Optional Chaining
- Operatoren
- Refactoring

Overloading

- Methodenauswahl hägt vom Typen ab
- Auch vom Returntyp

Overloading

```
class Person {
  func log() {
    println("Person::log")
  }
}

class SpecialPerson : Person {
  override func log() {
    println("SpecialPerson::log")
  }
}

let sp = SpecialPerson()
let p:Person = sp
p.log()
```

Overloading

```
func log(p:Person) {
  println("Person")
}

func log(p:SpecialPerson) {
  println("Special Person")
}



let sp = SpecialPerson()
let p:Person = sp
log(p)
```

Typinferenz & komplexe Ausdrücke

- Collections
- Collections von Collections
- Collections von heterogenen Collections

Homogene Collections

```
var myArray = [
                ["name" : "Amin"],
                ["name" : "Chris"]
              ]
```

- Kann ich hinzufügen:
  - [“name” : “Lattner”]
  - [“age” : 5]
  - 5

Heterogene Collection

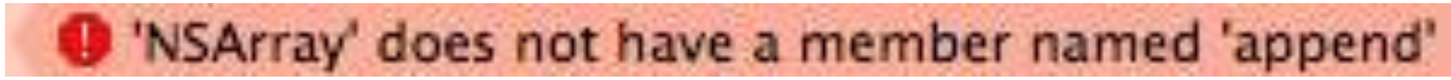
```
var myArray = [
                  ["name" : 1],
                  ["name" : "Chris"]
              ]
```

- Kann ich hinzufügen:
  - [“name” : “Lattner”]
  - [“age” : 5]
  - 5

Lösungen

```
var myArray = [
                ["name" : 1],
                ["name" : "Chris"]
              ]
```

- Kann ich hinzufügen:
  - [“name” : “Lattner”]: Nein, Compilerbug



  - [“age” : 5]: Ja, falls kein Compilerbug
  - 5: Ja, fall kein Compilerbug

Übersicht

```
let myarray                    = [1,2,3]                          // [(Int)]
let myarray2                   = [1,"2",3]                        // NSArray
let myarray3:[Any]             = [1,"2",3]                        // [Any]
let myarray4                   = [["name" : "Amin"],              // [[String:String]]
                                  ["name" : "Chris"]]
let myarray5                   = [["name" : 1],                   // NSArray
                                  ["name" : "Chris"]]
let myarray6:[[String:Any]] = [["name" : 1],                      // [String:Any]
                                  ["name" : "Chris"]]
let myarray7                   = [[2 : 1],                        // NSArray
                                  ["name" : "Chris"]]
let mydict8                    = ["name" : 1.5, "age" : 4]        // NSDictionary
```

Special Person; Person

Typinferenz & Overloading

- Typinferenz: Herleitung des Typen
- Kombination mit anderen Konstrukten

```
// Swift
var date = NSDate()
```

```
// Objective-C
NSDate *date = [NSDate new];
```

Typinferenz

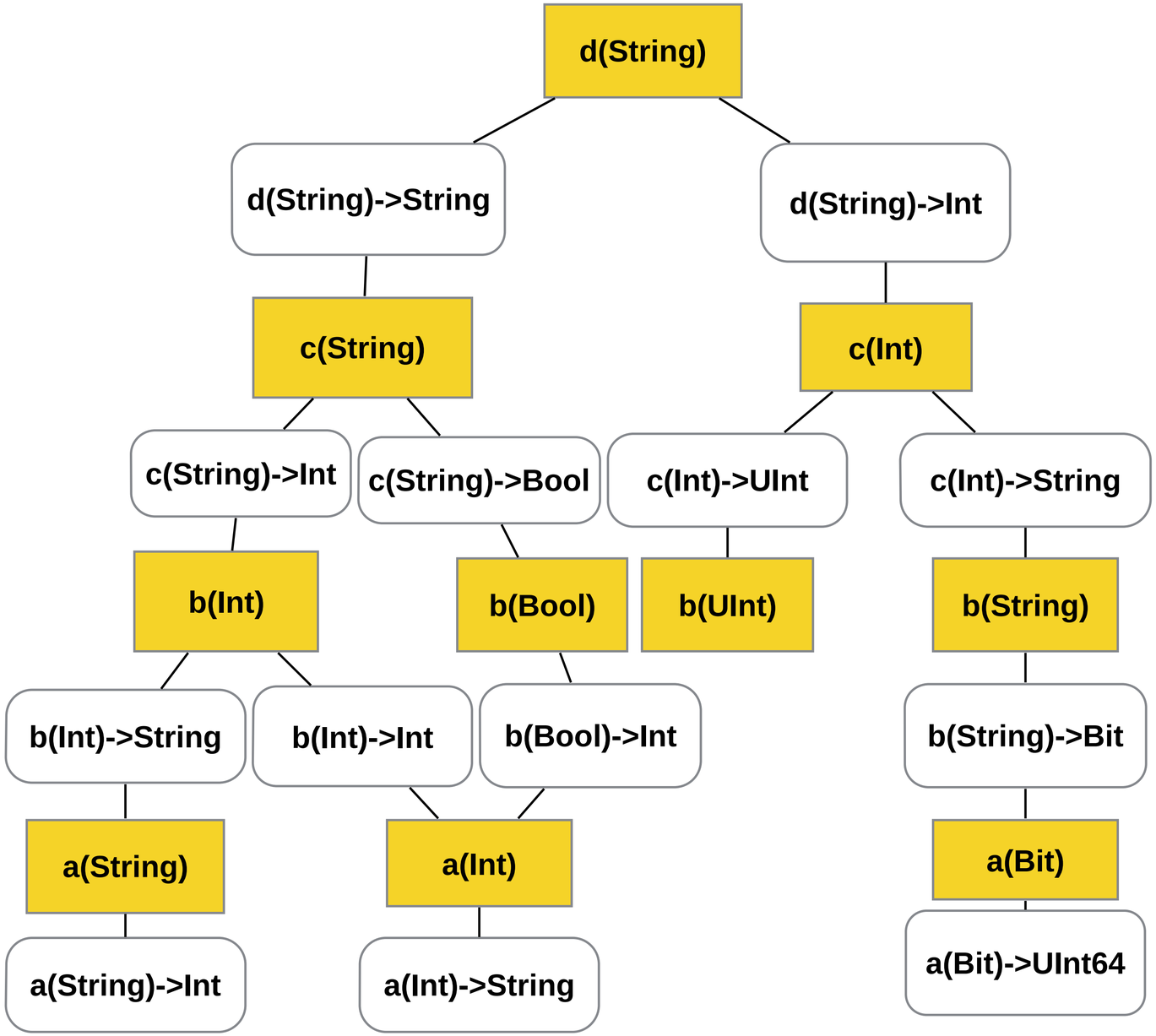
- *"Type inference is particularly useful when you declare a constant or variable with an initial value." Quelle: The Swift Programming Language*
- Für einfache Fälle

```
func d(input : String) -> Int { return 1 }
func d(input : String) -> String { return "Hello" }

func c(input : String) -> Int { return 2 }
func c(input : String) -> Bool { return false }
func c(input : Int) -> String { return "c" }
func c(input : Int) -> UInt { return 1 }

func b(input : Int) -> String { return "c" }
func b(input : Int) -> Int { return 1 }
func b(input : Bool) -> Int { return 1 }
func b(input: String) -> Bit { return Bit.One }

func a(input : String) -> Int { return 2 }
func a(input : Int) -> String { return "c" }
func a(input : Bit) -> UInt64 { return 4 }
```

Typinferenz

```
var ergebnis:Int = a(b(c(d("Hello"))))
```

Welche Funktionen werden aufgerufen?


d(String)
d(String)->String
d(String)->Int
c(String)
c(Int)
c(String)->Int
c(String)->Bool
c(Int)->UInt
c(Int)->String
b(Int)
b(Bool)
b(UInt)
b(String)
b(Int)->String
b(Int)->Int
b(Bool)->Int
b(String)->Bit
a(String)
a(Int)
a(Bit)
a(String)->Int
a(Int)->String
a(Bit)->UInt64

Typinferenz

```
var ergebnis:String = a(b(c(d("Hello"))))
```

Welche Funktionen werden aufgerufen?

Typinferenz

```
var ergebnis:String = a(b(c(d("Hello"))))
```

Could not find an overload for 'a' that accepts the supplied arguments

Generische Programmierung

- Swift: Templates
- Objective-C: Typdynamik

Generische Programmierung: Swift

```
 func memoize<T: Hashable, U>(body : ((T)->U, T)->U) -> (T)->U {
  var memo = Dictionary<T, U>()
  var result: ((T)->U)!
  result = { x in
    if let q = memo[x] { return q }
    let r = body(result, x)
    memo[x] = r
    return r
  }
  return result
}
```

Quelle: WWDC

Generische Programmierung: Objective-C

```
typedef id(^memoizeFunction)(id val);
typedef id(^function)(id val);
memoizeFunction memoize(function func) {
  __block NSMutableDictionary *memo = [NSMutableDictionary new];
  memoizeFunction result = (id)^(id input) {
    if(memo[input] != nil) {
      return memo[input];
    }
    id output = func(input);
    memo[input] = output;
    return output;
  };
  return result;
}
```

Generische Programmierung

- Nützlich für die alltägliche Entwicklung?
- Anwendungen sind konkret
- Komplexität

Optionals

- Wert oder nil
- Kein NSNotFound, NSIntegerMax, NSIntegerMin, NULL, 0, ...
- Formalisierung von "Policies"

Optionals

addObject:

Inserts a given object at the end of the array.

```
- (void)addObject:(id)anObject
```

Parameters

anObject

The object to add to the end of the array's content. This value must not be `nil`.

> **Important:** Raises an `NSInvalidArgumentException` if *anObject* is `nil`.

Optionals

```
- (void)addObject:(id)object {
  if(object == nil) {
    @throw [NSException exceptionWithName:NSInvalidArgumentException
                                   reason:@"object cannot be nil."
                                 userInfo:nil];
  }
  [self.storage addObject:object];
}
```

```
func addObject(object:T) {
    storage.append(object)
}
```

Optionals

- Objective-C via Dokumentation
- Swift via Optionals
- Wünschenswert: "Lightweight Optionals"
  - z.B.: Annoationen für den Compiler

Optional Chaining: Swift

- Mehrere Instruktionen zusammenfassen
- Void-Methoden

Optional Chaining

```
 class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// prints "Unable to retrieve the number of rooms.”
```

Optional Chaining: Objective-C

- Nachricht an nil schicken: OK
- Nur letztes Ergebnis prüfen
- Letzte Prüfung oft überflüssig

Optional Chaining: Objective-C

```
@interface Residence : NSObject
@property NSUInteger numberOfRooms;
@end

@interface Person : NSObject
@property (strong) Residence *residence;
@end

int main(int argc, const char * argv[]) {
  Person *john = [Person new];
  NSUInteger numberOfRooms = john.residence.numberOfRooms;

  NSLog(@"number of rooms: %lu", numberOfRooms);
  // $ number of rooms: 0
  return 0;
}
```

Optional Chaining: Methodenaufruf

```
if john.residence?.printNumberOfRooms() != nil {
    println("It was possible to print the number of rooms.")
} else {
    println("It was not possible to print the number of rooms.")
}
// prints "It was not possible to print the number of rooms.”
```

Quelle: The Swift Programming Language

Optional Chaining: Methodenaufruf

```
Person *john = [Person new];
if(john.residence != nil) {
   [john.residence printNumberOfRooms];
} else {
   NSLog(@"error");
}
```

- Auch nur ein “if”
- Redundanz

Operatoren

- Operatoren als Nachrichten
- Eigene Operatorten

Operatoren als Nachrichten

- Operatoren sind gewöhnter für numerische Typen
- Operatoren sind ungewöhnter für nicht-numerische Typen

Operatornachrichten

```
let a:Vector = Vector(x:5, y:3)
let b:Vector = Vector(x:98, y:11)
let c = a + b
```

```
Vector *a = [Vector newVectorWithX:5 y:3]
Vector *a = [Vector newVectorWithX:5 y:3]
Vector *c = [a add:b];
```

Operatornachrichten

```
let a:Vector = Vector(x:5, y:3)
let b:Vector = Vector(x:98, y:11)
let c = a * b // ?
```

```
Vector *a = [Vector newVectorWithX:5 y:3]
Vector *b = [Vector newVectorWithX:98 y:11]
Vector *c = [a dotMultiply:b];
Vector *d = [a crossMultiply:b];
```

Operatornachrichten

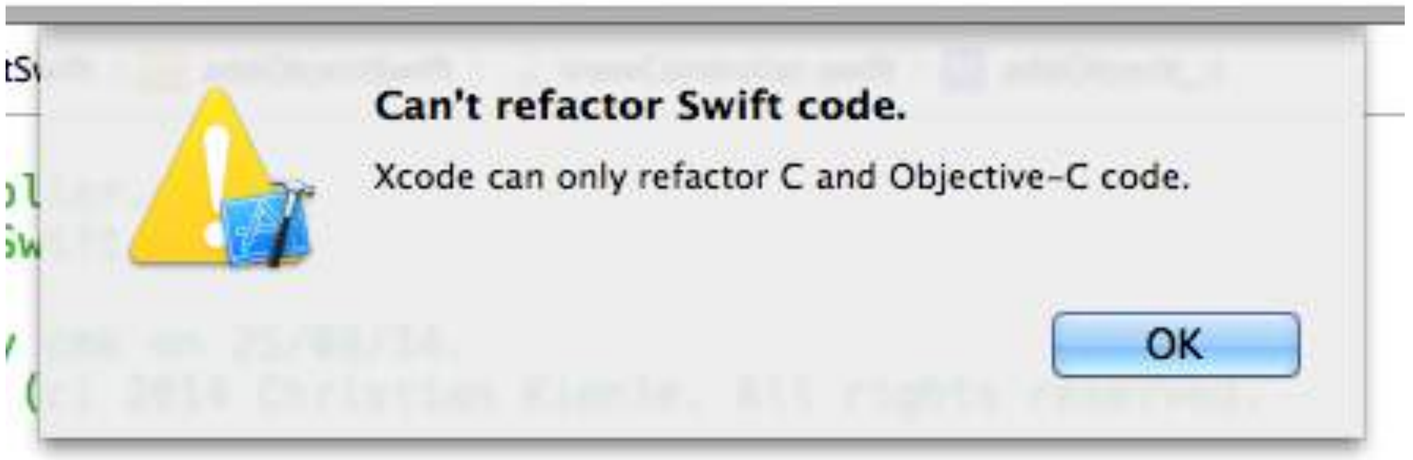
```
func * (left:Vector, right:Vector)->Vector {…}
func * (left:Vector, right:Vector)->Float {…}
…
let a:Vector = Vector(x:5, y:3)
let b:Vector = Vector(x:98, y:11)
let p = a * b        // Fehler
let d:Float = a * b // Dot
let c:Vector = a * b // Cross
let r:Float = a * b * a * b

// Ausdruck als Parameter für Overloading …
```

Operatornachrichten

```
let a = "Amin"
let b = “Negm"
let c = a + b
let d = a - b
let e = a + b - b
let f = a + (b - b)
```

Refactoring

- Objective-C: problematisch
  - -performSelector:
  - Late Binding
  - NSClassFromString(...)
- Swift: viel besser
  - Early Binding
  - Compiler hat mehr Wissen
  - Dadurch bessere Möglichkeiten


Can't refactor Swift code.
Xcode can only refactor C and Objective-C code.
OK

Swift	Geschichte	Tiere
$ 100		
$ 200		
$ 1000		

Type 'UInt8' does not conform to protocol 'ExtendedGraphemeClusterLiteralConvertible'

```
var a = "1" - "1"
```

Swift	Geschichte	Tiere
$ 200		
$ 1000		

'Int' is not identical to 'UInt8'

```
var a = 1
var b = "1"
var d = (a--)*b
```

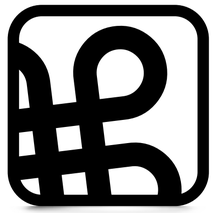
Swift	Geschichte	Tiere
$ 1000		

@lvalue $T3' is not identical to 'String'

```
protocol Named {
  var name: String {get}
}

protocol Aged {
  var name: Int {get}
}

var a : protocol<Named, Aged>
a.name = 1
```

Fragen?

Vielen Dank



Macoun