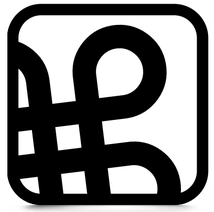


Macoun

Swiftifizieren

Nikolaj Schumacher

Swiftifizieren

Nikolaj Schumacher

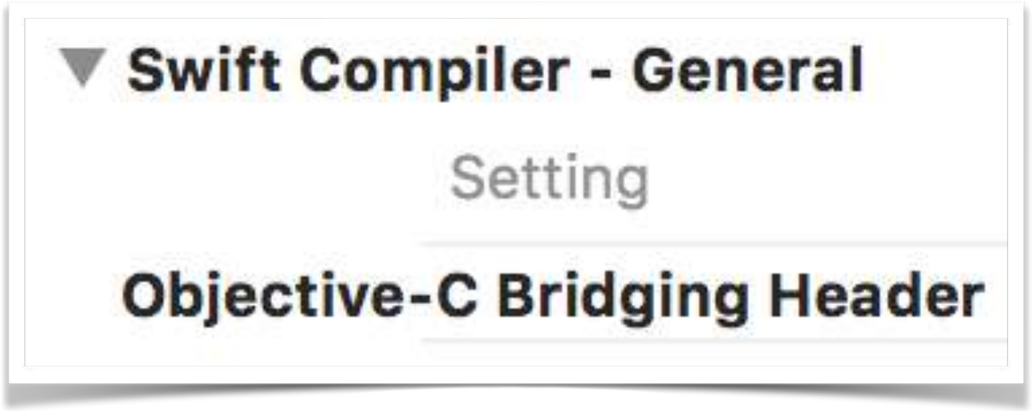
Ablauf

- Grundlagen
- Fortgeschrittene Techniken
- Strategien

Grundlagen

Bridging-Header

- <Module Name>-Bridging-Header.h
- automatisch von Xcode erstellt
- alle aus Swift sichtbaren ObjC-Header

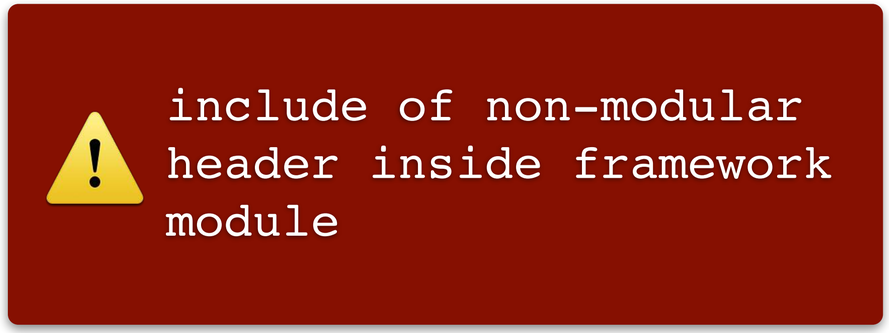


*-Swift.h

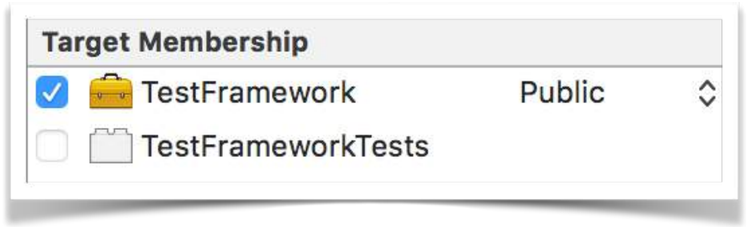
- <Module Name>-Swift.h
- automatisch vom Compiler erstellt
- Header für alle aus Objective-C sichtbaren Swift-Symbole
- in .m importieren, nicht in .h! (stattdessen Forward-Deklaration)
- erscheint erst wenn keine Compile-Fehler

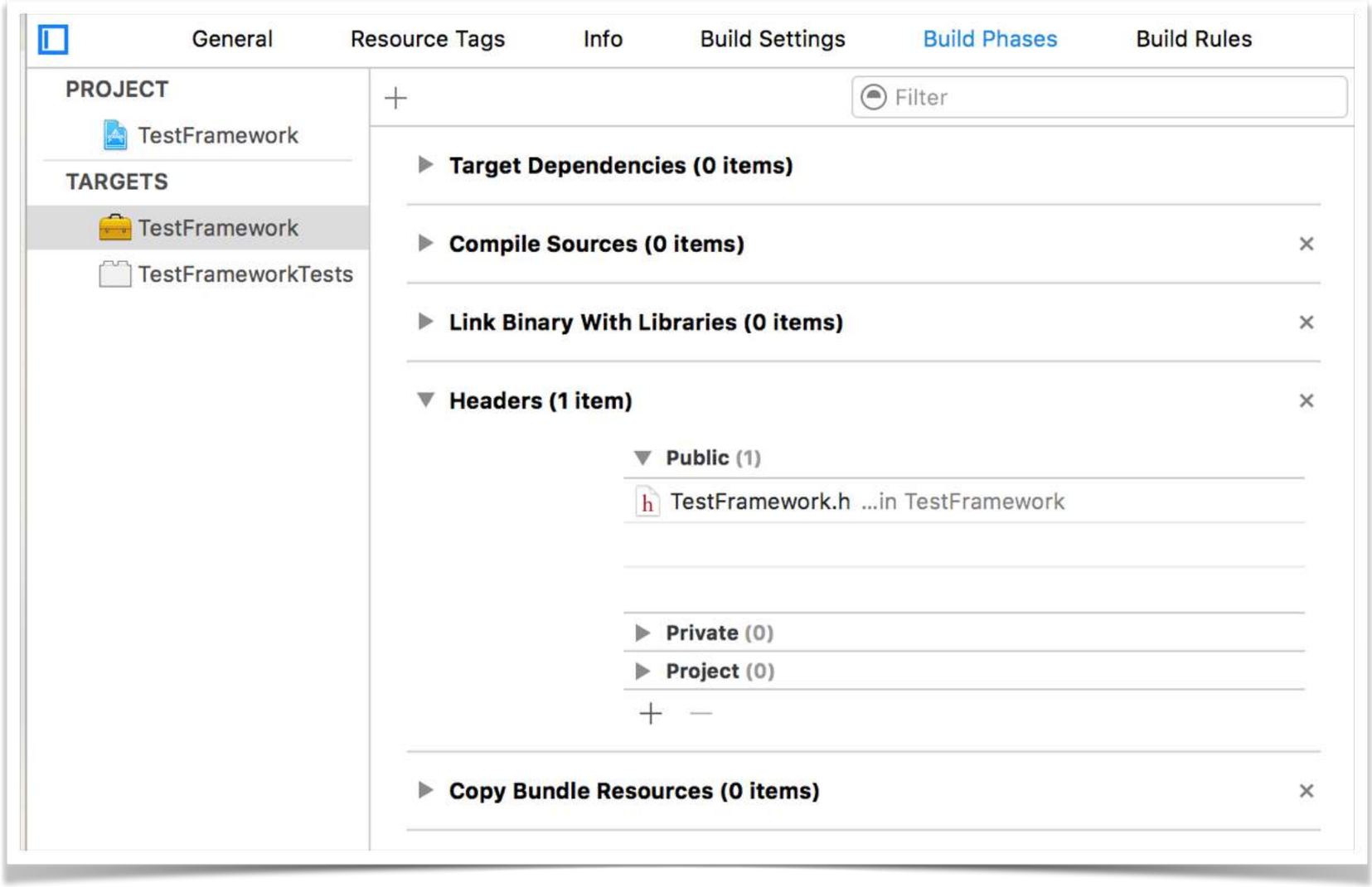
Framework

- Objective-C-Code
  - MeinFramework.h „umbrella header“
  - alle Header public [screenshot]
- Swift-Code
  - public



Framework


Target Membership
TestFramework
Public
TestFrameworkTests


General
Resource Tags
Info
Build Settings
Build Phases
Build Rules
PROJECT
TestFramework
TARGETS
TestFramework
TestFrameworkTests
Filter
Target Dependencies (0 items)
Compile Sources (0 items)
Link Binary With Libraries (0 items)
Headers (1 item)
Public (1)
TestFramework.h ...in TestFramework
Private (0)
Project (0)
Copy Bundle Resources (0 items)

Framework

- import MeinFramework / @import MeinFramework;
- kein Bridging-Header!

Einschränkungen (ง'̀-'́)ง

Swift-Features, die nicht aus Objective-C funktionieren

- Structs
- globale Funktionen & Variablen
- verschachtelte Typen
- von Swift-Klassen erben
- Tupel
- Generics
- Enums (außer Int-basiert)
- Typealias
- …

@objc-Attribut

- Swift-Klassen sind aus Objective-C sichtbar, wenn
  - public bzw. internal
  - @objc
- Klassen, Protokolle, Methoden, Errors

NSObject

- Swift-Klassen können von NSObject erben
- Swift-Protokolle können von NSObjectProtokol erben

NSObject

Vorteile

- dynamische Features
- optionale Methoden in Protokollen

Nachteile

- keine Generics

Namen

```
[[UIView alloc] initWithFrame: CGRectZero];
```



```
UIView(frame: .zero)
```

Namen

```
[[UITableView alloc] initWithFrame: frame style: UITableViewStyleGrouped];
```

```
UITableView(frame: frame, style: .grouped)
```

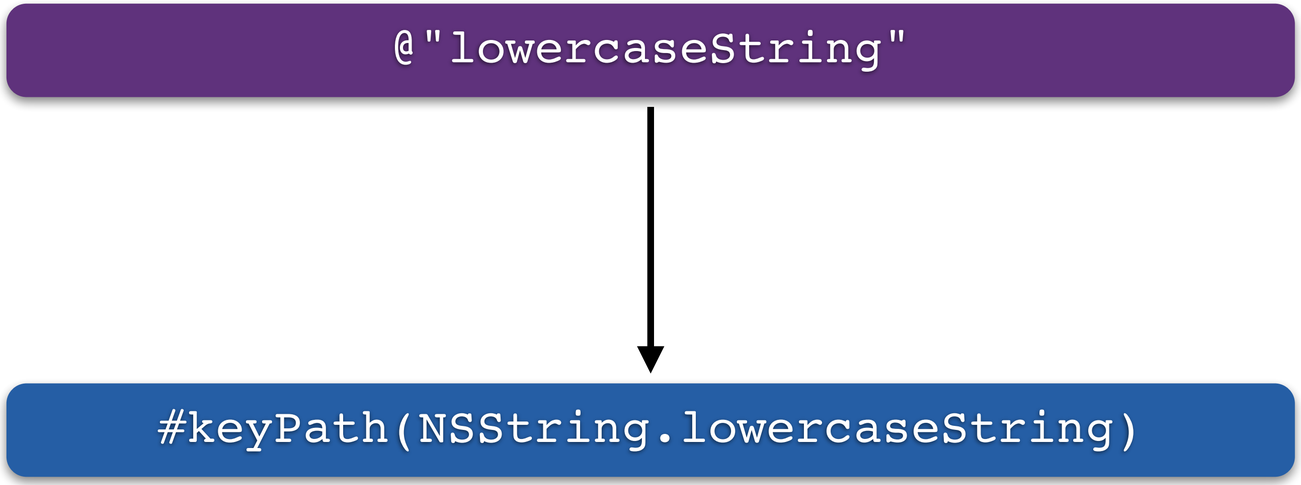
Foundation

- Foundation enthält Swift-Overlay für viele Typen
- NSDate, NSIndexPath, NSURL, etc.
- kein NS-Prefix
- Value-Typen statt Mutable/Immutable
- in Cocoa-API direkt verwendbar

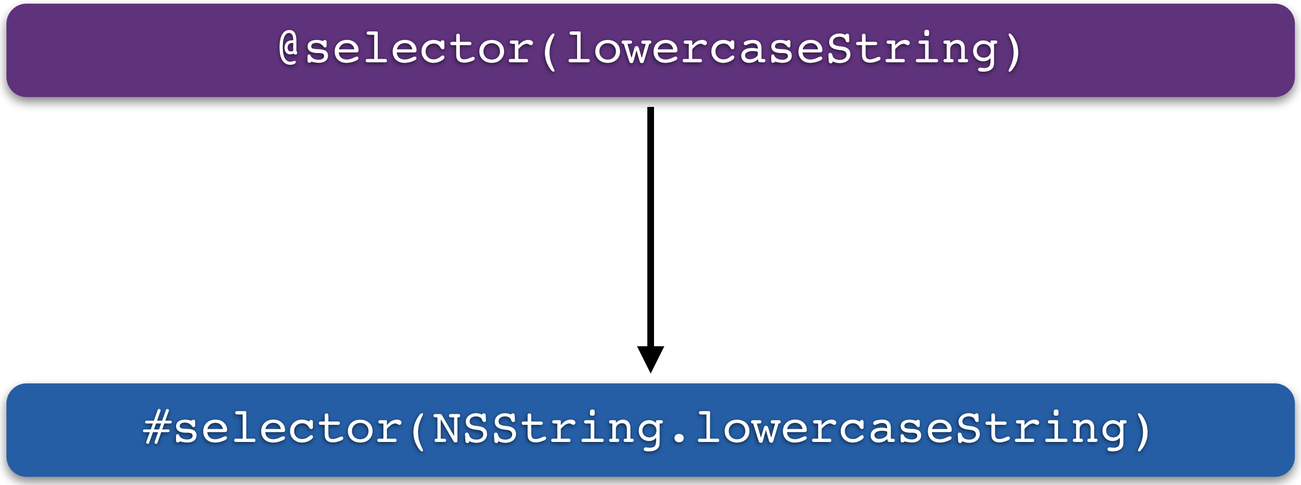
#selector und #keyPath

- Neu in Swift 3
- Namen vom Compiler geprüft

#keyPath



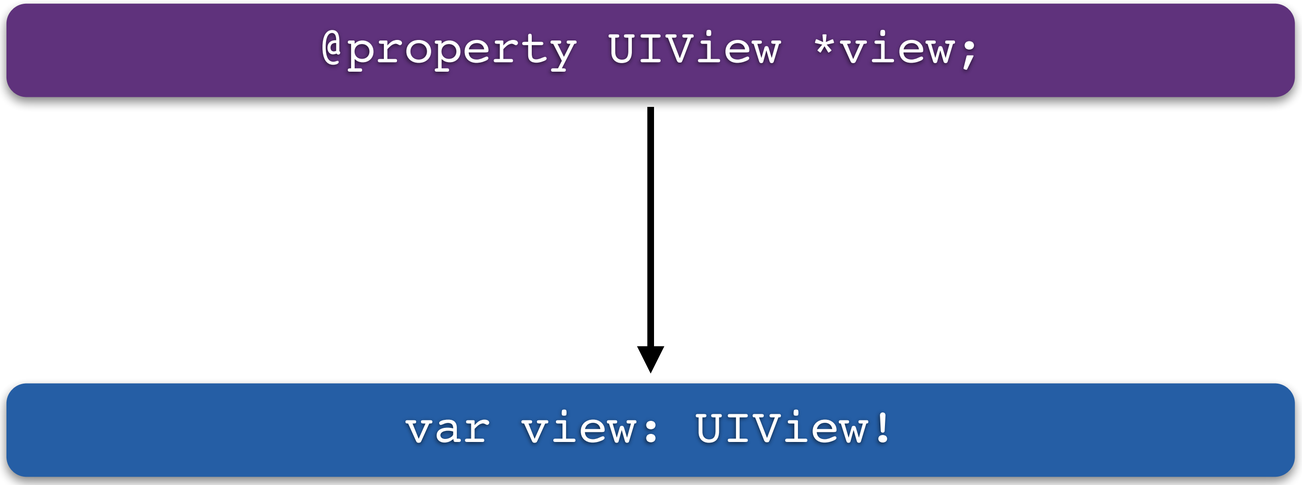
#selector



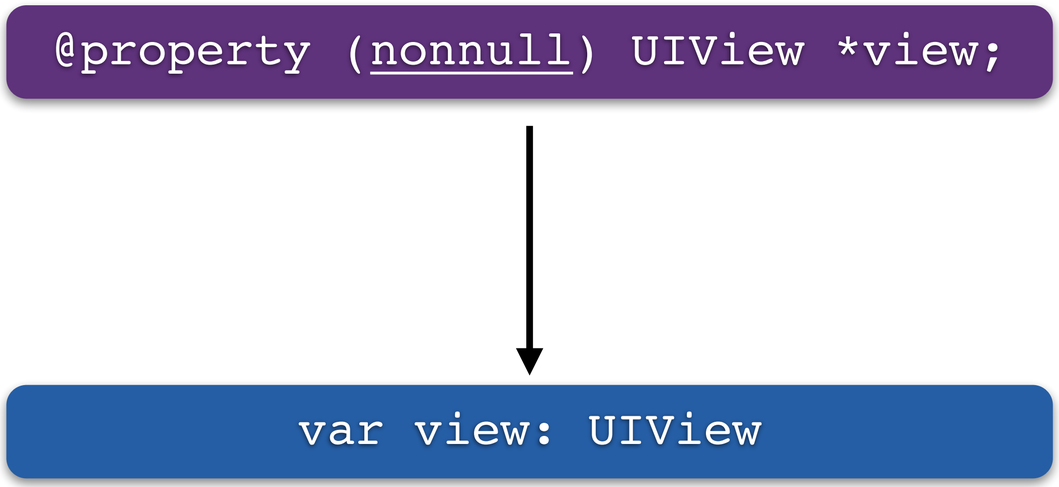
[TODO Bsp. mit Parameter]

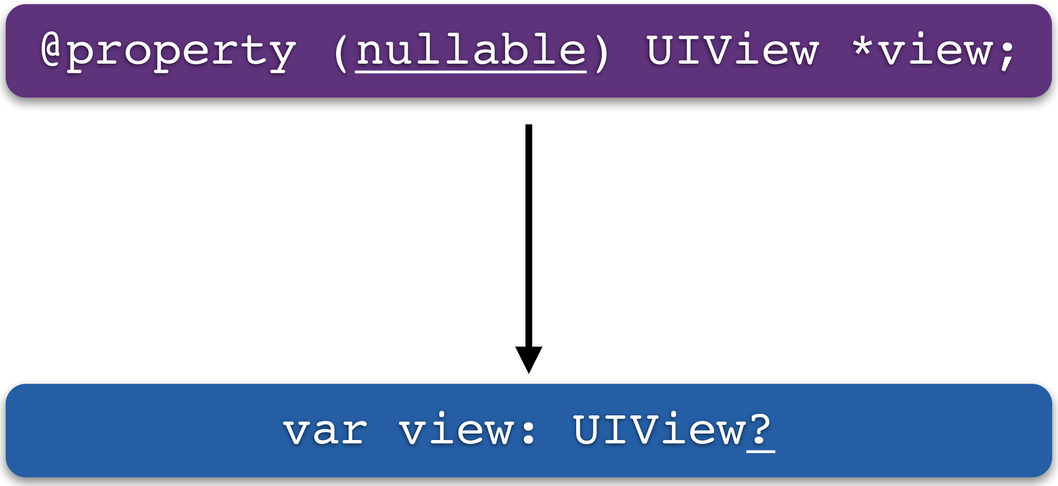
Fortgeschrittene Techniken

Nullable



Nullable


@property (nonnull) UIView *view;
var view: UIView


@property (nullable) UIView *view;
var view: UIView?

Nullable

```
NS_ASSUME_NONNULL_BEGIN

@property UIView *view;
- (void)methodWithArgument(UIView *)view;

NS_ASSUME_NONNULL_END
```

```
var view: UIView
func method(withArgument: UIView)
```

@objc()

```
NS_ASSUME_NONNULL_BEGIN

@property UIView *view;
- (void)methodWithArgument(UIView *)view;

NS_ASSUME_NONNULL_END
```

```
var view: UIView
func method(withArgument: UIView)
```

@objc()

```
@objc(methodWithArgument:)
func method(argument: UIView)
```

```
- (void)methodWithArgument(UIView *)view;
```

NS_SWIFT_NAME

```
- (void)methodWithArgument(UIView *)view
NS_SWIFT_NAME(method(argument:))
```

```
func method(argument: UIView)
```

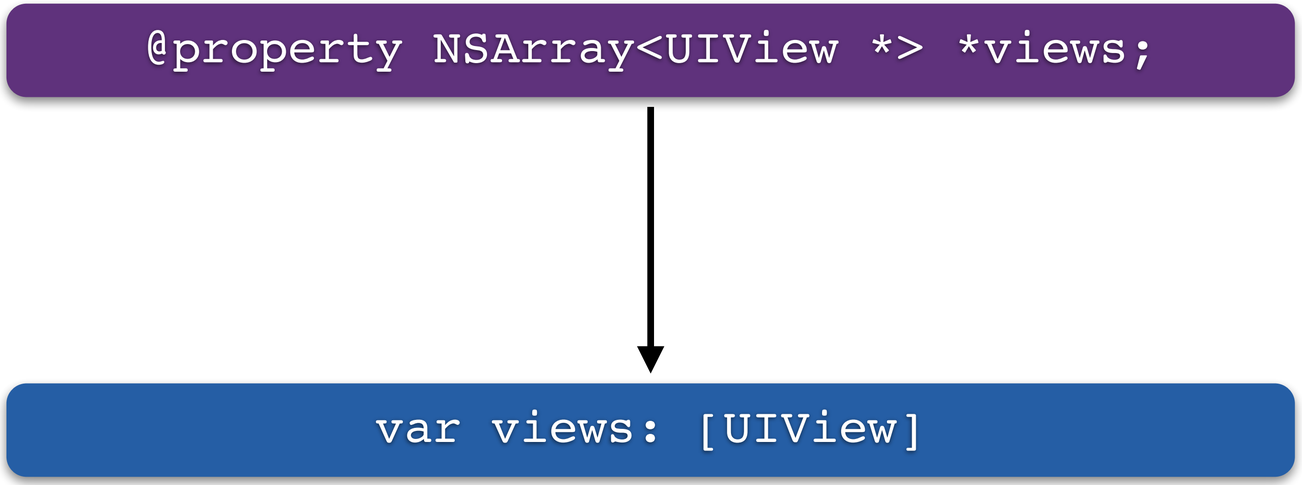
NS_REFINED_FOR_SWIFT

- benennt methode in __methode um
- Schönen Wrapper in Swift-Extension schreiben

@objc()

- Klassen
- Methoden
- Properties
- Enums

Lightweight Generics



Lightweight Generics

```
@interface MyClass<T: UIView *>: NSObject

@property T *view;
@property MyClass *parent;

@end
```

```
class MyClass<T: UIView>: NSObject {
    var view: T
    var parent: MyClass<UIView>
}
```

Flags und Enums

- NS_ENUM

Flags und Enums

- NS_OPTION

Extensions

- Extensions funktionieren auch auf Objective-C-Klassen

Equatable

```
@interface MyClass: NSObject

- (BOOL)isEqual;

@end
```

```
MyClass() == MyClass()
```

✓

Equatable

```
class MyClass: Equatable {}

func operator==(l: MyClass, r: MyClass) {
  …
}
```

```
[left isEqual: right]
```

NSCopying

```
class Kopierbar: NSObject, NSCopying {

  required override init() {
  }

  func copy(with zone: NSZone? = nil) -> Any {
    return type(of: self).init()
  }
}
```

Ausblenden

- NS_SWIFT_UNAVAILABLE
- @nonobjc

Strategien

Strategie 1
Neu schreiben

Vorteile

- keine API-Kompromisse
- weniger Bridging
- eine Baustelle weniger

Nachteile

- enormer Aufwand

Strategie 1
Neu schreiben

Wann sinnvoll?

- kleine Projekte
- gute Testabdeckung
- große Änderungen geplant

Strategie II
Von unten nach oben

Vorteile

- wenige API-Kompromisse
- Kompromisse verschwinden
- Value-Typen

Nachteile

- Schöne Swift-Konstrukte nicht aus Objective-C verwendbar

Strategie II
Von unten nach oben

Wann sinnvoll?

- gekapselte Logik mit wenig API

Strategie III
Von oben nach unten

Vorteile

- Model-API kann ohne Neu-Schreiben swiftifiziert werden

Nachteile

- API-Kompromisse werden verewigt

Strategie III
Von oben nach unten

Wann sinnvoll?

- Model-Schicht ausgereift

Strategie IV
Eine neue Hoffnung

Strategie IV

~~Eine neue Hoffnung~~

Strategie IV
Bunt gemischt

Vorteile

- langsamer Einstieg

Nachteile

- Überall API-Kompromisse
- Überall Durcheinander

Strategie IV
Bunt gemischt

Wann sinnvoll?

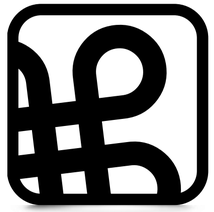
- Experimentieren

Strategie IV
Bunt gemischt

- Kein Fließtext
- Stichpunkte
- Fakten, etc.

Fragen?

Vielen Dank



Macoun