

Macoun

Swiftifizieren

Nikolaj Schumacher

Swiftifizieren

Nikolaj Schumacher

Swiftifizieren

Wie rette ich meinen Objective-C-Code in die neue Swift-Welt

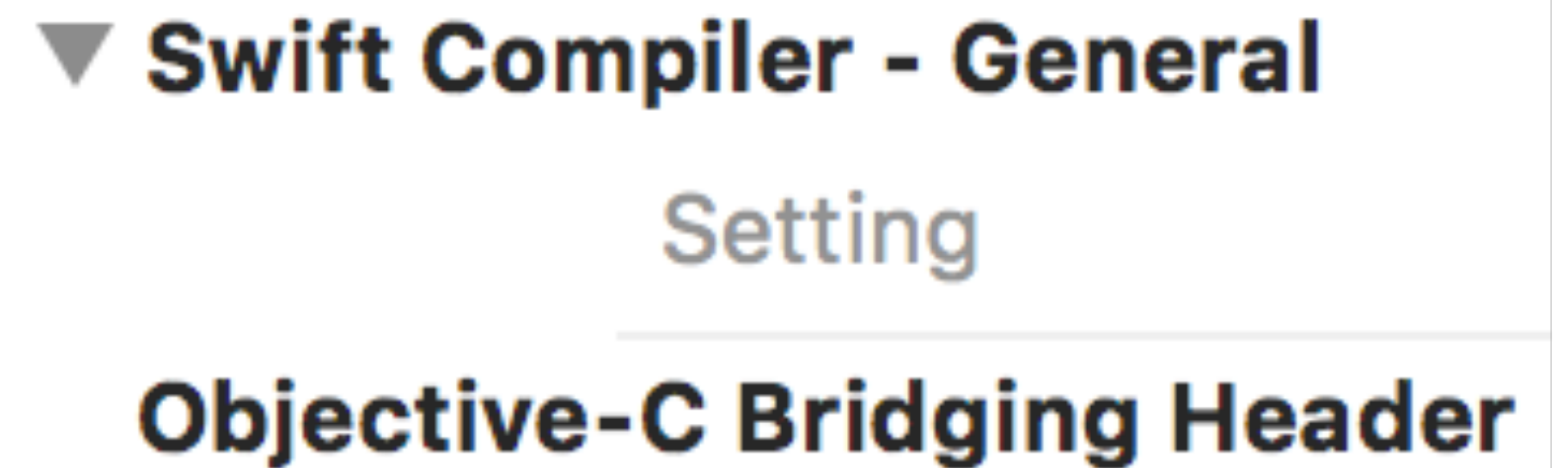
Ablauf

- Grundlagen
- Verfeinerung
- Strategien

Grundlagen

Bridging-Header

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



*-Swift.h

- <Module Name>-Swift.h
- automatisch generiert
- 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

- Swift-Code
 - (mindestens) public
- Objective-C-Code
 - MeinFramework.h „umbrella header“
 - alle Header public



include of non-modular
header inside framework
module

Framework

Target Membership			
<input checked="" type="checkbox"/>		TestFramework	Public
<input type="checkbox"/>		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)

h

TestFramework.h ...in TestFramework

▶ Private (0)

▶ Project (0)

+

—

▶ Copy Bundle Resources (0 items)

×

innerhalb des Frameworks

- Objective-C
- `#import <MeinFramework/MeinFramework-Swift.h>`
- Swift
- kein Bridging-Header!

Einschränkungen (ಇಿ-')ಇ

Swift-Features, die nicht aus Objective-C funktionieren

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

Extensions

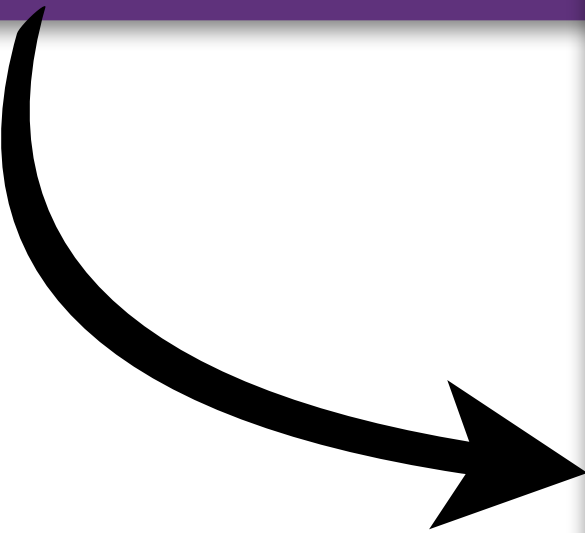
- Extensions funktionieren auch auf Objective-C-Klassen
- auch aus ObjC aufrufbar

@objc

- Swift-Klassen sind aus Objective-C sichtbar, wenn
 - mindestens internal
 - @objc
 - NSObject
- Klassen, Protokolle, Methoden, Errors

@objc

```
@objc  
public class MyClass: NSObject {  
  
}
```



```
__attribute__((objc_subclassing_restricted))  
@interface MyClass: NSObject  
  
@end
```

NSObject

- In Objective-C gibt es NSObject doppelt
- In Swift

Klassen	NSObject
Protokolle	NSObjectProtocol

von NSObject erben

Vorteile

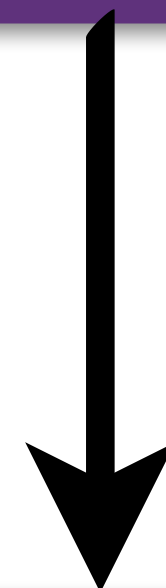
- dynamische Features (KVC, KVO, Swizzling, ...)
- optionale Methoden in Protokollen

Einschränkungen

- mit Generics nicht aus ObjC sichtbar
- nur compatible Properties aus ObjC sichtbar

Namen

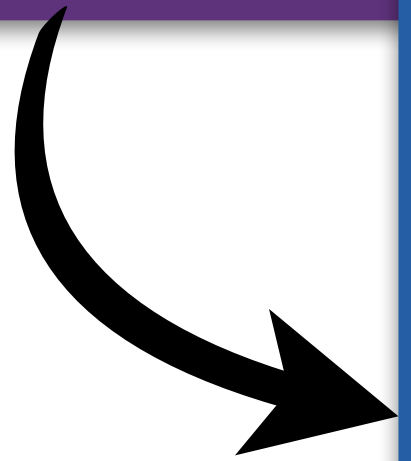
```
[[UIView alloc] initWithFrame: CGRectZero];
```



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

Namen

```
typedef NS_ENUM(NSInteger, MeinEnum) {  
    MeinEnumWert1,  
    MeinEnumWert2,  
    MeinEnumWert3,  
    MeinKomischerEnumWert  
};
```



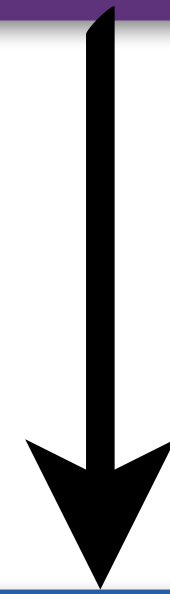
```
enum MeinEnum: Int {  
    case wert1  
    case wert2  
    case wert3  
    case komischerEnumWert  
}
```

#selector und #keyPath

- Neu in Swift 3
- Namen vom Compiler geprüft

#keyPath

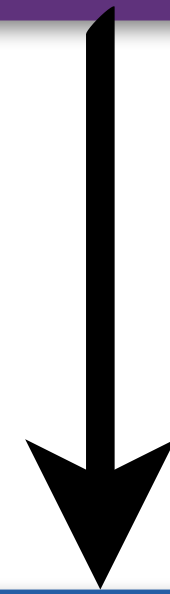
```
@ "lowercaseString"
```



```
#keyPath(NSString.lowercaseString)
```

#selector

```
@selector(lowercaseString)
```



```
#selector(NSString.lowercaseString)
```

Standard-Types

ObjC	Swift
id	Any
NSInteger	Int
char	Int8
int	Int32
long	Int
char *	UnsafeMutablePointer<Int8>
const char *	UnsafePointer<Int8>
bool, BOOL	Bool
unsigned int	UInt32
NSUInteger	UInt
bool *	UnsafeMutablePointer<Bool>
BOOL *	UnsafeMutablePointer<ObjCBool>

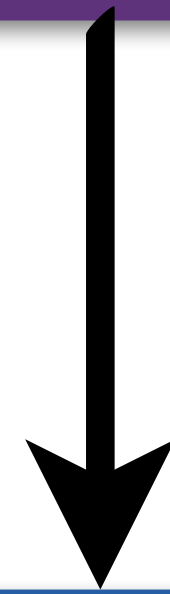
Foundation-Typen

- NSDate, NSIndexPath, NSURL, etc.
- Swift-Overlay für viele Typen
- kein NS-Prefix
- Value-Typen statt Mutable/Immutable (keine Referenz-Semantik!)
- automatisches Mapping

Verfeinerung

Nullable

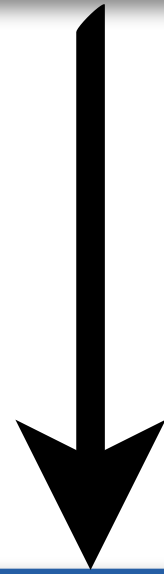
```
@property UIView *view;
```



```
var view: UIView!
```

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()

```
func methode(parameter: UIView)
```



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

@objc()

```
@objc(method:MitParameter:)  
func methode(parameter: UIView)
```



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

NS_SWIFT_NAME

CF_SWIFT_NAME für C

```
- (void)methodeMitParameter:(UIView *)view  
NS_SWIFT_NAME(methode(parameter:))
```



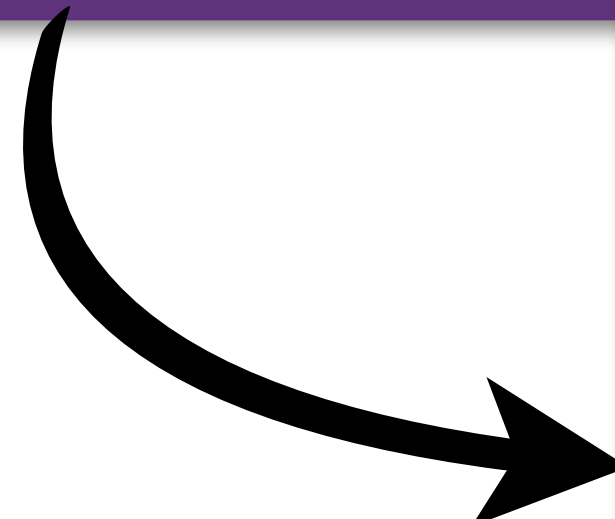
```
func methode(parameter: UIView)
```

NS_REFINED_FOR_SWIFT

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

NS_REFINED_FOR_SWIFT

```
- (void)methodeMitParameter:(UIView *)view  
NS_REFINED_FOR_SWIFT;
```



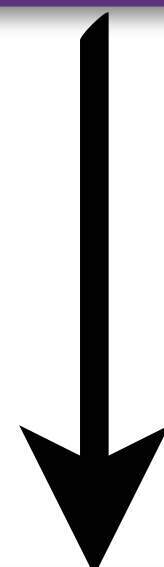
```
extension AppDelegate {  
    @nonobjc  
    func methode(parameter: UIView) {  
        __methodeMitParameter(parameter)  
    }  
}
```

Ausblenden

- NS_SWIFT_UNAVAILABLE
- @nonobjc

Lightweight Generics

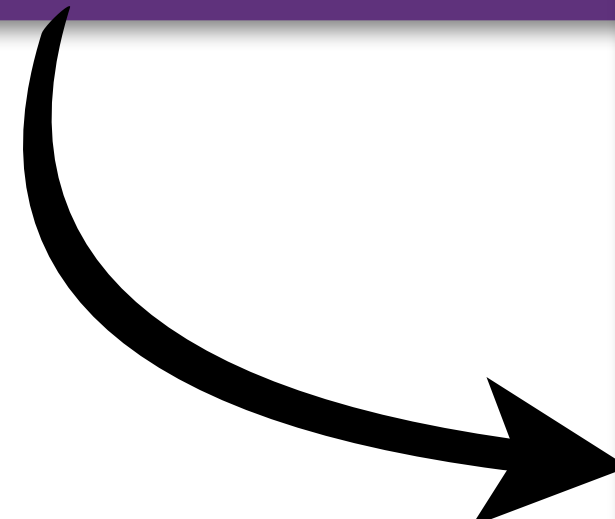
```
@property NSArray<UIView *> *views;
```



```
var views: [UIView]
```

Lightweight Generics

```
@interface MyClass<T: UIView *>: NSObject  
  
@property T view;  
@property MyClass<T> *parent;  
  
@end
```



```
let c1 = MyClass()  
let view1: UIView = c2.view  
  
let c2 = MyClass<UITableView>()  
let view2: UITableView = c2.view
```

Equatable

```
@interface MyClass: NSObject  
- (BOOL)isEqual;  
@end
```



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



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()  
    }  
}
```


NS_NOESCAPE

```
@interface MyClass: NSObject  
- (void)iterate:(BOOL (^)(NSInteger i))block;  
@end
```



```
x.iterate({x in x < self.arg})
```

NS_NOESCAPE

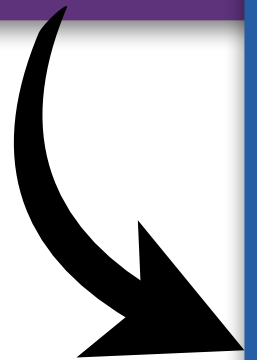
```
@interface MyClass: NSObject  
- (void)iterate:(BOOL (NS_NOESCAPE ^)(NSInteger i))block;  
@end
```



```
x.iterate({x in x < arg})
```

Stringly-typed Enums

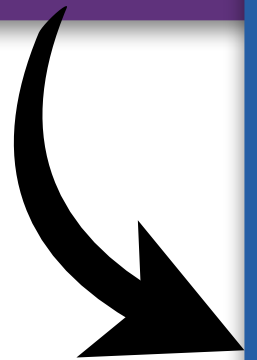
```
typedef NSString *MeinEnum NS_STRING_ENUM;  
MeinEnum const MeinEnumWert = @"MeinEnumWert";
```



```
enum MeinEnum: String {  
    case wert  
}
```

Stringly-typed Enums

```
typedef NSString *MeinEnum NS_EXTENSIBLE_STRING_ENUM;  
  
MeinEnum const MeinEnumWert = @"MeinEnumWert";
```



```
struct MeinEnum: RawRepresentable {  
    typealias RawValue = String  
  
    init(rawValue: RawValue)  
    var rawValue: RawValue { get }  
  
    static var wert: MeinEnum { get }  
}
```

Strategien

App

UI

Persistenz

Model

Logik

Strategie I: Neu schreiben

Vorteile

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

Nachteile

- enormer Aufwand
- enormes Risiko

Strategie I: 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 schmaler Schnittstelle

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 or schneller Einstieg

Nachteile

- Überall API-Kompromisse
- Überall Durcheinander

Strategie IV: Bunt gemischt

Wann sinnvoll?

- Experimentieren
- keine Lust mehr auf Objective-C

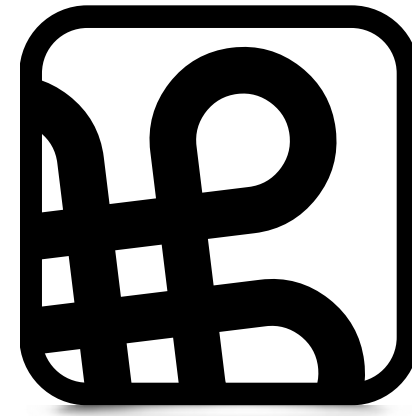
Ärger gibt's immer

Fragen?

Folien:

<https://github.com/nschum/talks/>

Vielen Dank



Macoun