12.–15.09.2010
in Nürnberg

Wissenstransfer
par excellence

Fallobst

Fallstricke in Objective C

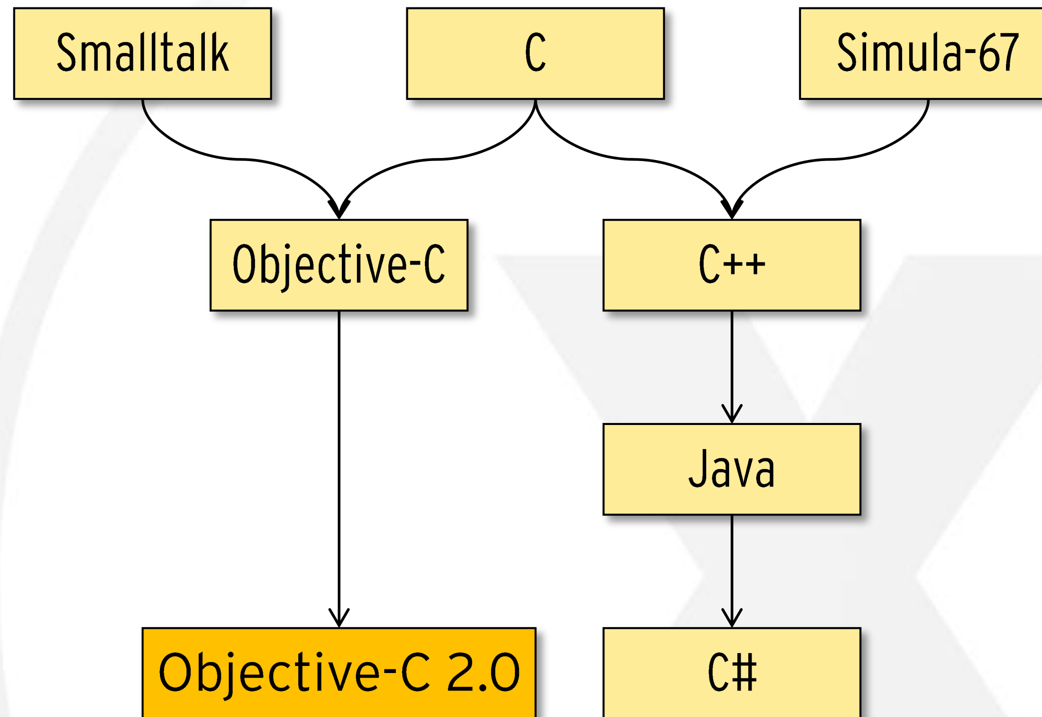
Andreas Oetjen
belox software gmbh

- **Einleitung**
- Objekterzeugung
- Speicherverwaltung
- Zugriffschutz
- Ausnahmen
- Kategorien
- Namensräume
- Liskov'sches Substitutionsprinzip
- Interface Builder
- Fazit

- Motivation
  - Eigene Erwartungen aus Java/C++/C#-Hintergrund
- Zielsetzung
  - Einstieg und Fortschritt erleichtern
- Inhalt
  - Unvollständig
  - Persönliche Gewichtung
- Grundlagen
  - Grundkenntnisse Objective C
  - OO-Erfahrung

- Warum Objective C?
  - Mac und iPhone-Entwicklung
- Warum „Fallobst“?
  - Vortrag soll Objective C nicht madig machen
    - Es gibt viele „schöne Seiten“
  - Fallstricke gibt es überall
    - C: Pointer-Arithmetik, Speicherverwaltung
    - C++: Copy-Konstruktor <-> `operator=()`
    - Java: `equals()` <-> `hashCode()`

- Geschichte



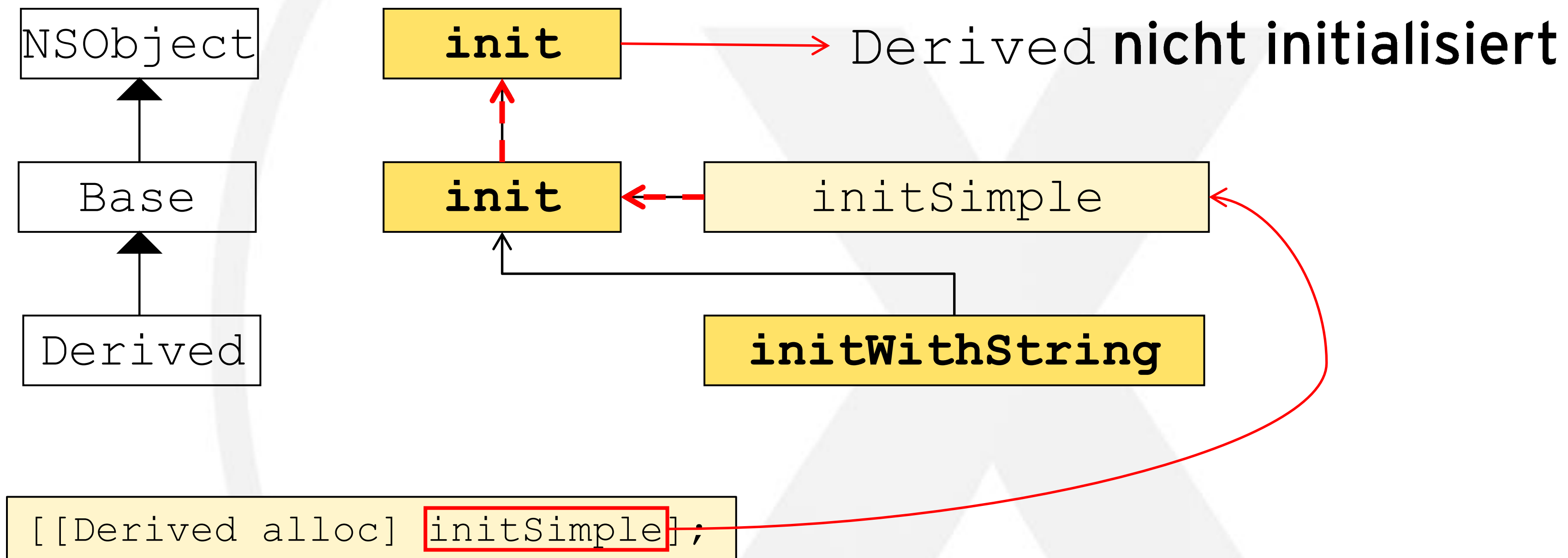
- Zielsetzung
  - Flexibel wie Smalltalk
  - Schnell wie C
- Schmale Sprachdefinition
  - Framework übernimmt viele Funktionen
  - Konventionen

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov'sches Substitutionsprinzip
Interface Builder
Fazit

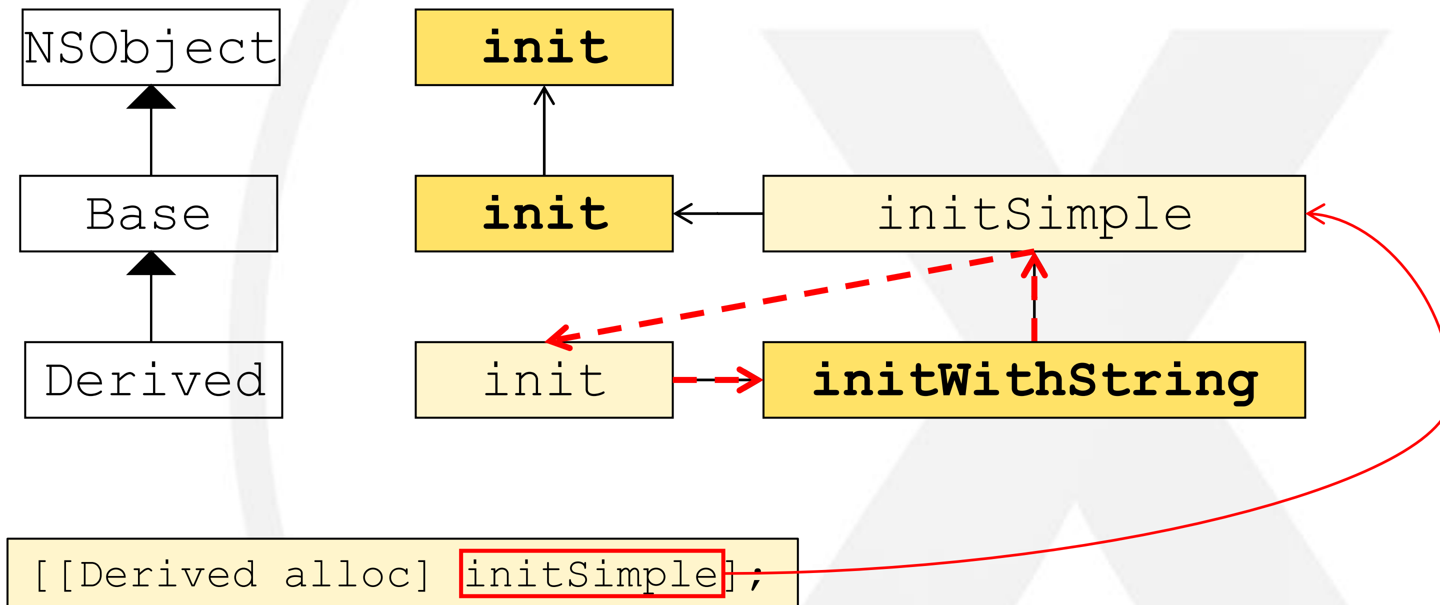
- Objekterzeugung
  - Ähnlich zu C
    - Erst Speicher anfordern, dann initialisieren
    - `[[Klasse alloc] init]` - Semantik
  - Keine erzwungene Initialisierung
    - Expliziter Aufruf notwendig
      - Methoden heißen immer `init*`
  - Keine implizite Basisklassen-Initialisierung
    - Dieser muss explizit aufgerufen werden

- Designierter Initialisierer
  - Muss von allen anderen `init`-Methoden aufgerufen werden
  - Muss von abgeleiteten Klassen aufgerufen werden
  - Muss überschrieben werden
    - Wenn ein neuer Initialisierer eingeführt wird
    - Dies stellt sicher, dass das abgeleitete Objekt korrekt initialisiert wird
  - Muss dokumentiert werden
    - Kein Sprachmittel

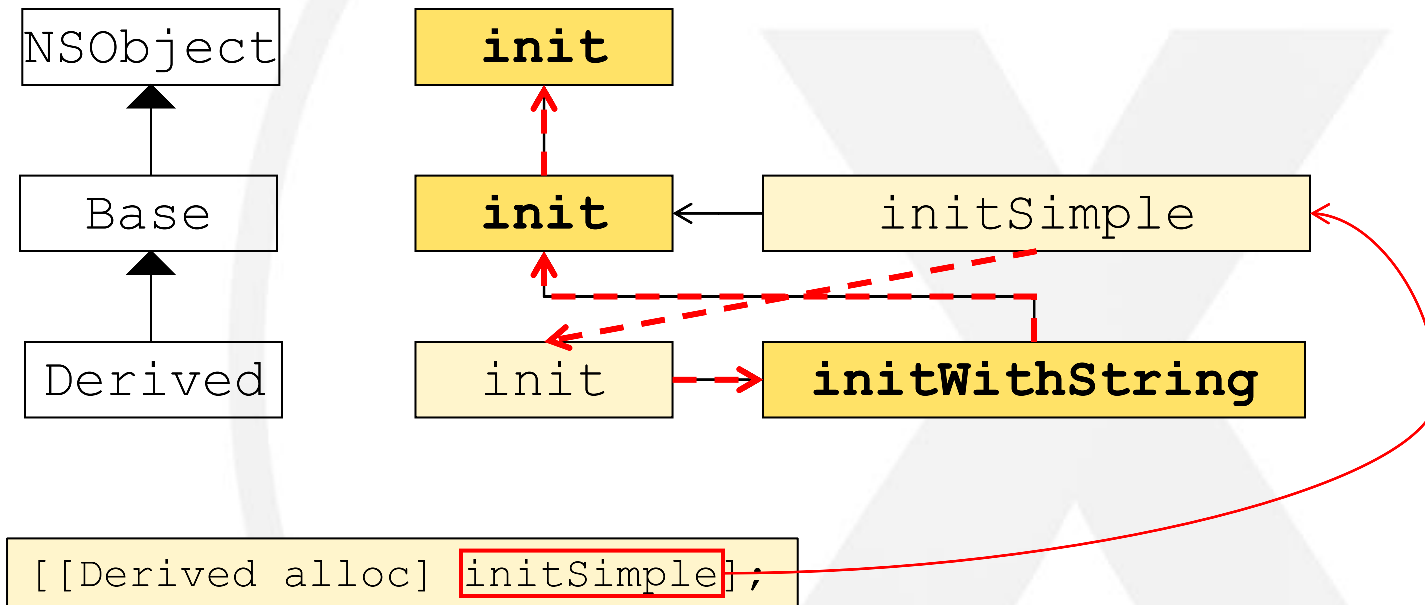
- Fehler: Neuer Designierter Initialisierer eingeführt
  - Aber Basisklassen-Initialisierer nicht überschrieben



- Fehler: Aufruf des falschen Basisklassen-Initialisieres
  - Endlose Rekursion



- Korrekte Implementierung der Initialisierer
  - Beim Aufruf von `init` greift der virtuelle Mechanismus



- Initialisierer kann Implementierung austauschen
  - Sehr häufig anzutreffen bei Class Cluster

```
- (id) init {
  self = [super init];
  if (self == nil)
    return nil;
  // Weiterer Initialisierungscode
}
```

Basisklassenaufruf und Zuweisung an `self`

Prüfung auf `nil`

- Austausch der Implementierung
  - Aus Benutzersicht

Nein:

```
Base obj = [Base alloc];
[obj init];
[obj doSomething];
```

`init` kann ein anders Objekt returnen

`obj` wäre dann nicht initialisiert

Ja:

```
Base obj = [Base alloc];
obj = [obj init];
[obj doSomething];
```

Besser:

```
Base obj = [[Base alloc] init];
[obj doSomething];
```

- Fehlerbehandlung im Initialisierungscode
  - Sobald Fehlerfall festgestellt, wird `self` freigeben und aufgeräumt
  - Wenn `[super init]` den Wert `nil` zurückliefert, ist `dealloc` bereits gelaufen!
    - Auf `self` darf nicht mehr zugegriffen werden – ist bei korrekter Verwendung ja auch `nil`

- Fehlerbehandlung im Initialisierungscode

Derived

```
- (id) init {
    self = [super init];
    if (self == nil)
        return nil;

    // Initialisierungscode
    return self;
}
```

An dieser Stelle ist `dealloc` bereits durchlaufen

Base

```
- (id) init {
    if (fehlerfall) {
        [self release];
        // Cleanup ...
        return nil;
    }
    // ...
}
```

Ruft `dealloc` (polymorph) auf

Nur Aufräumen, was `dealloc` nicht schon bereinigt hat.
Nicht mehr auf `self` zugreifen!

- Convenience-Konstruktoren
  - Statische Methoden
  - Einfaches Erzeugen
  - Objekte werden auto-released
    - Aufrufer muss `retain` aufrufen, wenn das Objekt gehalten werden soll

Ja:

```
+ (id) newObject {
  id x = [[[self alloc] init] autorelease];
  return x;
}
```

- Convenience-Konstruktoren

Vorsicht:

```
+ (id) newObject {
  self = [[[self alloc] init] autorelease];
  return self;
}
```

`self` ist das Klassenobjekt – nach der Zuweisung verweist `self` jedoch auf das erzeugte Objekt!

Nein:

```
+ (id) newObject {
  id x = [[[Klasse alloc] init] autorelease];
  return x;
}
```

Funktioniert nicht mehr, wenn `Klasse` abgeleitet wird, ohne gleichzeitig `newObject` zu überschreiben

- Objekte freigeben
  - `dealloc`-Methode implementieren
  - Apple sagt: Felder freigeben (`release`)
  - `dealloc` der Superklasse aufrufen

Vorsicht:

```
- (void) dealloc {
   [value release];
   // ...
   [super dealloc];
}
```

Auf keinen Fall `dealloc` aufrufen!

Aber nur, wenn `retain`ed wurde!

Nicht mehr auf `value` zugreifen (Dangling Pointer)

- Freigabe mittels setter-Methode
  - Einzige Möglichkeit bei synthetisierten, nicht-deklarierten Instanzvariablen
  - **Achtung:** Evtl. Seiteneffekte in *`set`*-Methode
    - Apple-Dokumentation empfiehlt `release`
    - Ich sage: Setter-Methode benutzen!

Ja:

```
- (void) dealloc {
   [self setValue:nil];
   // ...
   [super dealloc];
}
```

Vorheriger Wert wird freigegeben; entspricht `self.value=nil`

Weitere Zugriffe auf `value` würden ignoriert, da `nil`

- Freigabe mittels setter-Methode
  - Einheitlicher Code

```
@property (nonatomic, retain) NSString *retainValue;
@property (nonatomic, copy)   NSString *copyValue;
@property (nonatomic, assign) NSString *assignValue;
// ...
- (void) dealloc {
    [self setRetainValue:nil];
    [self setAssignValue:nil];
    [self setCopyValue:nil];
    [super dealloc];
}
```

Gleicher Code, unabhängig von der Property-Deklaration

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov‘sches Substitutionsprinzip
Interface Builder
Fazit

- Referenzzählung
  - Notwendig in iOS
    - iPhone, iPad
  - Prinzip der Ownership
    - (Nur) wer Besitzer eines Objektes ist, muss das Objekt auch wieder freigeben
  - Namenskonventionen
    - Methoden mit definiertem Namensschema

- Besitz erwerben
  - Implizit (beim Erzeugen)
    - Aufruf von Methoden mit den Namen `alloc`*, `new`* oder *`copy`*
  - Explizit erwerben
    - Aufruf von `retain`
- Ein Objekt kann mehrere Besitzer haben
- Besitz freigeben
  - Nur erlaubt, wenn man ein Objekt besitzt
    - Aufruf von `release`
    - Aufruf von `autorelease`

- Collections
  - `NSArray`, `NSDictionary`, `NSSet`
    - Und deren Subklassen
  - Erwerben Besitz der enthaltenen Objekte
  - Vorsicht beim Entfernen von Objekten
  - Vorsicht beim Freigeben der Collection

Nein:

```
id obj = [mutableArray objectAtIndex:0];
[mutableArray removeObjectAtIndex:0];
[obj doSomething];
```

Ruft `release` auf
Potentieller *dangling pointer*

Nein:

```
id obj = [mutableArray objectAtIndex:0];
[mutableArray release];
[obj doSomething]; // hier könnte es knallen
```

Gibt auch alle enthaltenen Objekte frei
Potentieller *dangling pointer*

- Collections
  - Aufruf von `retain` verhindert Dangling-Pointer-Problem

Ja:

```
id obj = [mutableArray objectAtIndex:0];
[obj retain];
[mutableArray removeObjectAtIndex:0];
[mutableArray release];
[obj doSomething];
```

Zugriff OK

- Properties
  - Je nach `@property`-Deklaration erfolgt ein implizites `retain` / `release`
    - Mögliche Referenzierung gelöschter Werte

Nein:

```
@property (retain) NSString *title;
//...
NSString *oldTitle = [anObject title];
[anObject setTitle:@"New Title"];
NSLog(@"Old title was: %@", oldTitle);
```

Gibt den alten Wert (also auch `oldTitle`) frei

Potentieller *dangling pointer*

- Garbage Collection
  - Nur in bestimmten Umgebungen
    - Nicht auf iOS
  - Methoden wie `retain`, `release`, `dealloc` etc. sind als No-Op implementiert
  - Statt `dealloc` wird `finalize` aufgerufen
    - Keine Aussage über Aufrufreihenfolge
    - Finalisiertes Objekt nicht „auferstehen“ lassen

- Garbage Collection: CoreFoundation
  - CoreFoundation-Klassen werden nicht vom GC erfasst
  - Aufruf von `CFRetain()`
    - Erhöht Referenzzähler
  - Aufruf von `CFRelease()`
    - Erniedrigt Referenzzähler
    - Besser: `CFMakeCollectable()` / `NSMakeCollectable()`
  - Aufrufe müssen ausbalanciert sein

- Garbage Collection: CoreFoundation
  - `CFRetain()`/`NSMakeCollectable()` nicht mixen mit `-retain`/`-release`
    - Wechselseitige No-Ops, abhängig von der Umgebung
  - Rückgabe von CF-Objekten

```
- (NSString *)getString {
   CFStringRef s = CFStringCreate(…);

   return [NSMakeCollectable(s) autorelease];
}
```

NO-OP in refCount-Umgebung

NO-OP in GC-Umgebung

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov‘sches Substitutionsprinzip
Interface Builder
Fazit

- Nur rudimentärer Zugriffsschutz
  - `@private` nur auf Attribut / Property-Ebene möglich
  - Private Methoden nur möglich in anonymen Kategorien

- Einfaches Umgehen des Zugriffsschutzes
  - Einfach aufrufen
    - Compilerwarnung ignorieren
      - Oder das Objekt gleich auf `id` casten
  - Neue Kategorie mit entsprechenden Methoden deklarieren
    - Aber nicht implementieren
  - Manchmal geht‘s noch einfacher:

```
#define private public
#import "secretClass.h"
```

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov‘sches Substitutionsprinzip
Interface Builder
Fazit

- Ausnahmebehandlung als Sprachmittel
  - `@try`, `@catch`, `@finally`, `@throw`
    - Auf den ersten Blick nichts neues
  - Kostenlose try-Blöcke seit Objective C 2.0
    - In 1.0 wurde dies noch durch teure `setjmp` / `longjmp` Aufrufe gelöst
- Viele Schnittstellen enthalten weiterhin eine `:(NSError **)` - Schnittstelle

- Klasse `NSException`
  - Repräsentiert ein Exception-Objekt
  - Es sollte nur genau diese Klasse verwendet werden
    - Exception-Typ als String-Property
    - Subklassen könnten vom Framework „verschluckt“ werden

- Keine `throws`-Deklaration auf Methodenebene möglich
  - Speicherlecks quasi überall möglich

Vorsicht:

```
Klasse obj = [[Klasse alloc] init];
[obj doSomething];
[obj release];
```

Es ist unspezifiziert, ob hier eine Exception möglich ist
Falls ja, würde `obj` nicht freigegeben

Ja:

```
Klasse obj = [[Klasse alloc] init];
@try {
   [obj doSomething];
} @finally {
   [obj release];
}
```

Speicher wird in jedem Fall freigegeben

- Exception-Objekte werden typischerweise im Autorelease-Pool abgelegt
  - Freigabe erfolgt automatisch
  - Problem bei geschachtelten Pools

- Autorelease-Pools
  - Dangling Exception-Objekte

Nein:

```
NSAutoreleasePool *innerPool = [[NSAutoreleasePool alloc] init];
Klasse obj = [[Klasse alloc] init];
@try {
    [obj doSomething];
} @finally {
    [obj release];
    [innerPool drain];
}
```

Gibt den Speicherplatz für alle `autorelease`d-Objekte frei, also auch für die (nach außen geworfene) Exception

- Autorelease-Pools
  - Schnelle Lösung

```
NSAutoreleasePool *innerPool = [[NSAutoreleasePool alloc] init];
Klasse obj = [[Klasse alloc] init];
@try {
   [obj doSomething];
} @finally {
   [obj release];
}
[innerPool drain];
```

- Wird im Exception-Fall nicht ausgeführt
- Das `autorelease` erfolgt dann vom äußeren Pool
- Aber: Speicher könnte volllaufen
  - Es gibt sicherlich einen Grund für `innerPool`

- Autorelease-Pools
  - Lösung: Weiterreichen der Exception an den äußeren Pool

Ja:

```
NSAutoreleasePool *innerPool = [[NSAutoreleasePool alloc] init];
Klasse obj = [[Klasse alloc] init];
id innerException;
@try {
    [obj doSomething];
} @catch(NSException* exception) {
    innerException = [exception retain];
    @throw;
} @finally {
    [obj release];
    [innerPool drain];
    [innerException autorelease];
}
```

retainCount == 2 (retain)

retainCount == 1 (drain)

Im äußeren Pool (autorelease)

- Viele Ausnahmen können nicht per `@catch` / `@finally` behandelt werden
  - Division durch Null
  - Zugriff auf gelöschten Speicher
- Klasse `NSExceptionHandler`
  - Anmeldung eines Delegates zur Behandlung bestimmter Ausnahmen
    - Nicht gefangene Exceptions
    - System- und Runtime-Exceptions
  - Nicht verfügbar in iOS
  - Keine Auswirkung auf `@catch` / `@finally`

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov‘sches Substitutionsprinzip
Interface Builder
Fazit

Kategorien

- Kategorien erweitern existierende Klassen
  - Alternative zur Ableitung
  - Neue Methoden können hinzugefügt werden
- Bestehende Methoden können (unabsichtlich) überschrieben werden
  - Basisklassenmethoden
  - Eigene Methoden
  - Methoden einer anderen Kategorie

- Überschreiben existierender Methoden
  - Basisklassenmethoden
    - Können dennoch per `super` aufgerufen werden
  - Eigene Methoden
    - Original-Implementierung verschwindet
  - Methoden einer anderen Kategorie
    - Es ist undefiniert, wer gewinnt

- Fragile Base Class Problem
  - Scheinbar neue Methoden einer Kategorie können bei Evolution der Basisklasse zu Konflikten führen
    - Neue Basisklassen-Version definiert „alte“ Kategoriemethode
  - Keine Compiler-Warnung etc.
    - Kein „overrides“-ähnliches Sprachkonzept vorhanden

- Kategorien für NSObject
  - Zugriff auf `self` ist problematisch

Vorsicht:

```
@implementation NSObject (TestCategory)

- (void) doSomething {
    NSLog(@"[%@ doSomething] => %@",[self class], self);
}
```

```
NSString *s = [NSString stringWithFormat:@"hallo welt"];
[s doSomething];

[NSNumber doSomething];
```

[NSCFString doSomething] => hallo welt

[NSNumber doSomething] => NSNumber

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov'sches Substitutionsprinzip
Interface Builder
Fazit

- Objective C kennt keine Namensräume
  - Präfix vor Klassenname
    - NSString, UILabel, MKMapView
- Relevant für Bibliotheksentwickler
- Hilfe durch Compiler / Linker
- Ein ähnliches Problem existiert bei Kategorien
  - Hinzugefügte Methodennamen können bestehende Namen unabsichtlich (und ohne Warnungen) überschreiben

- Umgehung
  - *Per Eintrag in inoffizielle Präfix-Liste*

    http://www.cocoadev.com/index.pl?ChooseYourOwnPrefix
  - Per `#define`

```
#define ClassName DeBeloxProjectClassName
```

  - Verwendung von `@compatibility_alias`

```
@interface DeBeloxProjectClassName : NSObject
@end

@compatibility_alias ClassName DeBeloxProjectClassName

@implementation ClassName
@end

ClassName *myClass;
```

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov‘sches Substitutionsprinzip
Interface Builder
Fazit

- Das LSP besagt:
  - Ein Objekt einer Basisklasse kann immer durch ein Objekt der abgeleiteten Klasse ersetzt werden
  - Sorgt z.B. bei Java Collections für scheinbar merkwürdige „Verrenkungen“
    - Stichwort <? Super T>, <? extends T>

- In Objective C ist ein Selektor nur an den Methodennamen und die Parameter gebunden
  - Rückgabewert spielt keine Rolle
- Entsprechend können abgeleitete Methoden einen anderen Rückgabetypen haben
  - Dies führt dann leicht zum Absturz

- LSP-Verletzung durch Überschreibung

Nein:

```
@implementation Base
 - (NSString *) func {
      return @"Dibadibadu";
 }
@end
@implementation Derived
 - (float) func {
      return 42.0;
 }
@end

Derived  *derivedObj  = [[Derived  alloc] init];
Base *baseObj = derivedObj;
//...
NSString *val = [baseObj func];

NSLog(@"func returns: %@", val);
```

Unterschiedliche Rückgabetypen

Wir wissen es besser, aber so will es der Compiler!

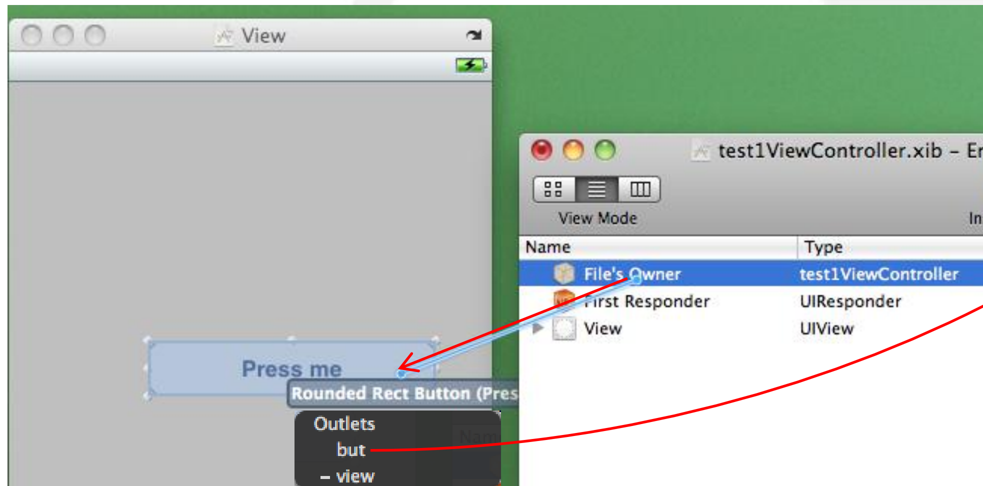
EXC_BAD_ACCESS

Einleitung
Objekterzeugung
Speicherverwaltung
Zugriffschutz
Ausnahmen
Kategorien
Namensräume
Liskov‘sches Substitutionsprinzip
Interface Builder
Fazit

- Verknüpfung von Code und Interface-Builder erfolgt über Markierungen
  - `IBOutlet`
  - `IBAction`
- Interface-Builder verwendet diese Markierungen
  - Bietet nur die passenden Felder bzw. Methoden an

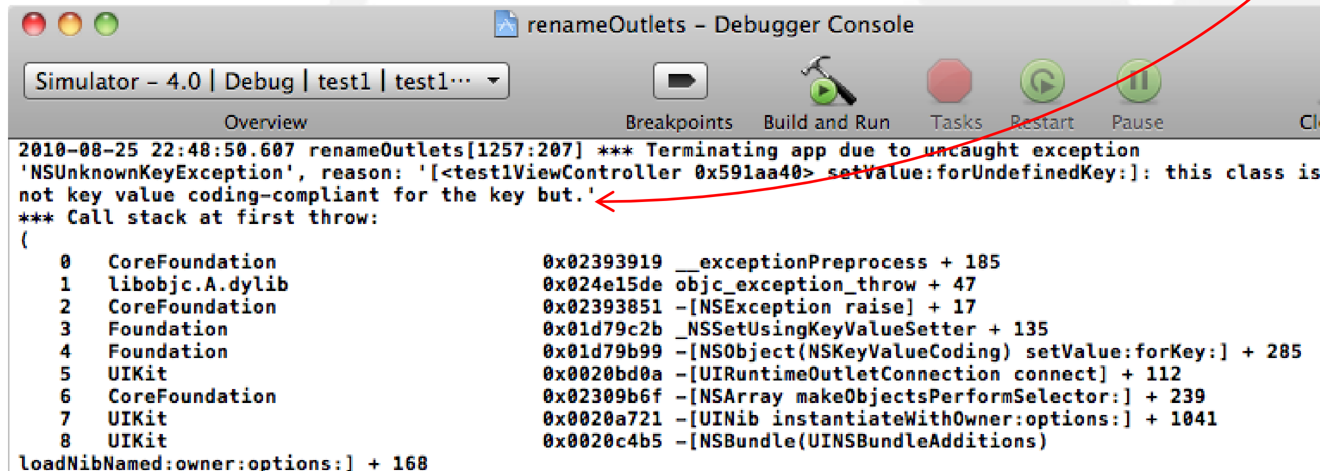
- Herstellen einer Verbindung

```
@interface test1ViewController : UIViewController {
    UIButton *but;
}
@property (nonatomic, retain) IBOutlet UIButton *but;
```



- Laufzeitfehler bei Code-Änderung

```
@interface test1ViewController : UIViewController {
    UIButton *button;
}
@property (nonatomic, retain) IBOutlet UIButton *button;
```



- Einleitung
- Objekterzeugung
- Speicherverwaltung
- Zugriffschutz
- Ausnahmen
- Kategorien
- Namensräume
- Liskov'sches Substitutionsprinzip
- Interface Builder
- **Fazit**

- Sprache schützt nur bedingt vor Fehlern
  - Fehlende Konstruktoren
  - Schwacher Zugriffsschutz
  - Manuelles Speichermanagement
    - Zumindest in bestimmten Umgebungen
  - Ausnahme-Behandlung nicht durchgängig

- Framework übernimmt Sprachfunktionen anderer OO-Sprachen
  - Umgehungsmöglichkeiten existieren
- Verlangt disziplinierte Entwickler
  - Konventionen beachten

- Entwickler mit Java / C++ / C#-Hintergrund
  - Verwunderung / Enttäuschung über fehlende Sprachmittel
  - Sprache erscheint nicht „modern“
    - Viele C-Relikte

- Aber
  - Mac-Rechner und deren Anwendungen gelten als recht stabil
  - Die Sprache zwingt den Entwickler zu sauberer Arbeit
    - Airbag vs. Spitze Speere

- Ausblick
  - Statische Codeanalyse hilft manchmal

```
NSString *o = [NSString stringWithFormat:@"hallo"] ;
[o release];        Incorrect decrement of the reference count of an object that is not owned at this point by the caller
```

```
NSString *o = [arr objectAtIndex:0];
[arr removeObjectAtIndex:0];
NSLog(@"Wert nach release ist: %@", o);        Potential leak of an object allocated on line 30 and stored into 'arr'
```

  - Wird in XCode 4 einiges besser?
    - Unterstützung durch integrierten Interface-Builder?
  - Grundprobleme bleiben
    - Warten auf Objective C 3.0 ?

Vielen Dank!

Andreas Oetjen
belox software gmbh