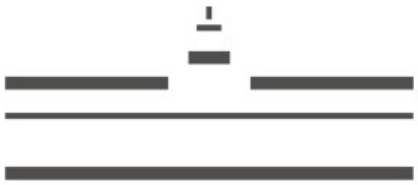


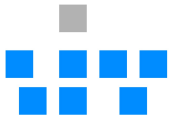
Westfälische
Wilhelms-Universität
Münster

Programmierung für mobile Endgeräte

Objective-C (Speichermanagement)

wissen.leben
WWU Münster

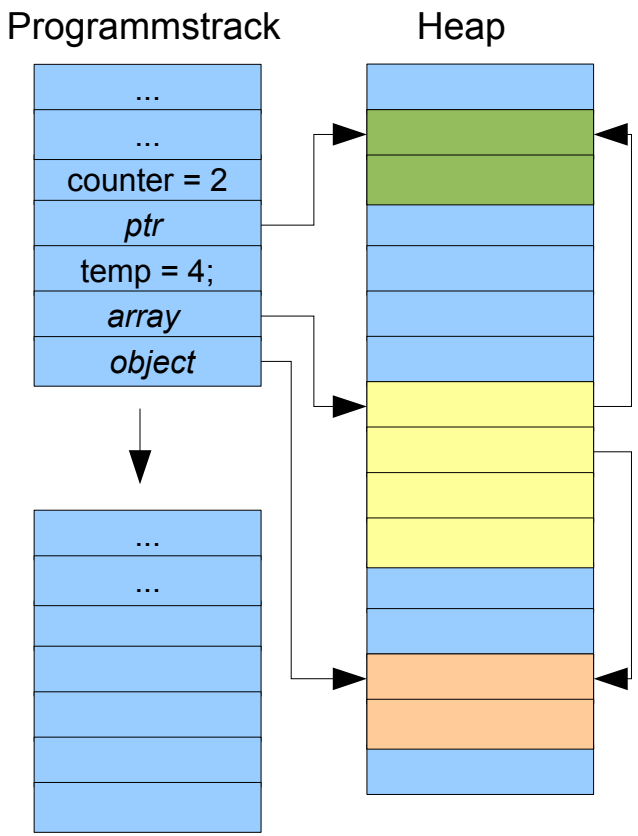
Name: Patrick Förster



ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Objective-C: Speichermanagement (Beispiel)

```
-(void) simpleExamples:
        (int) counter withPointer: (id) ptr {

    int temp = (count * sizeof(int))/2;
    NSMutableArray* array =
         [[NSMutableArray alloc] initWithCapacity:temp];

    [array addObject:ptr];

    MemoryObject* object = [MemoryObject new];

    [array addObject:object];

    ...
}
```

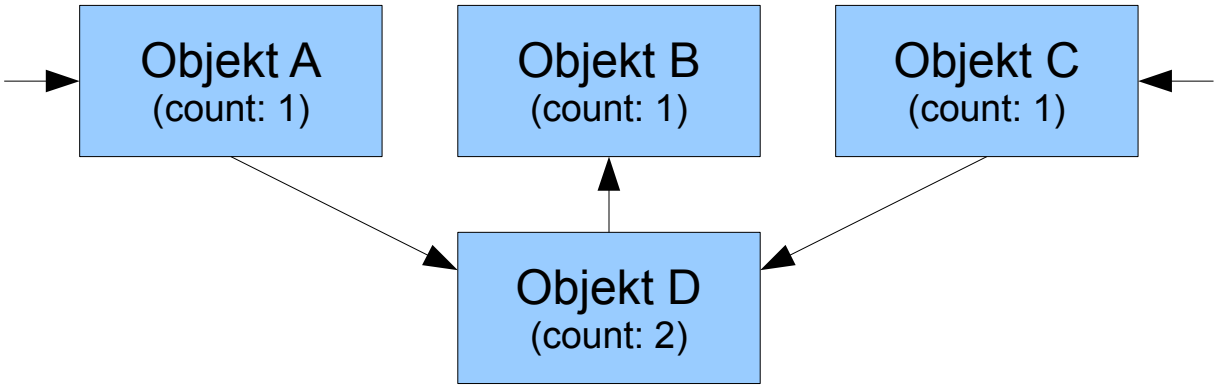


Objective-C: Speichermanagement (Überblick)

- Jedes Objekt, dass mit alloc erzeugt wird, belegt Platz im dynamischen Speicher
- Ist dieser voll kann es zum Absturz der gesamten Applikation kommen
- Der belegte Speicherplatz muss wieder freigegeben werden, sobald ein Objekt nicht mehr benötigt wird
- Allgemein gibt es zwei Möglichkeiten, dieses Problem anzugehen:
  1. Automatische Speicherbereinigung (Garbage Collection): Zur Laufzeit wird überprüft, welche Objekte im Speicher noch benötigt werden, alle anderen werden entfernt
  2. Manuelle Speicherbereinigung: Der Programmierer ist selbst dafür verantwortlich, den von ihm belegten Speicherplatz wieder freizuräumen
- Schlechte Nachricht: Es gibt keine Garbage-Collection für Objektive-C in Kombination mit iOS (Garbage-Collection allgemein wurde von Apple als „deprecated“ erklärt)
- Gute Nachricht: Es gibt durchaus Mittel und Wege das manuelle Speichermanagement zu vereinfachen

Objective-C: Referenzzähler

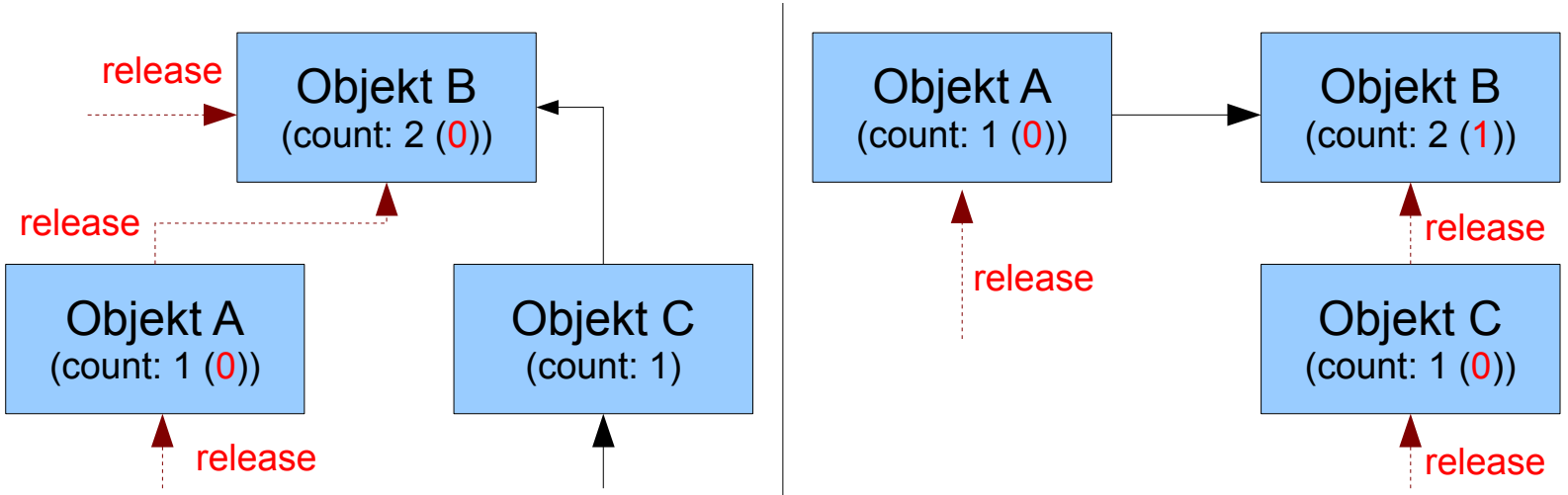
- Das Mittel zur manuellen Speicherbereinigung in Objektive-C ist der Referenzzähler
- Jedes Objekt verfügt über diesen Zähler, der besagt, wie viele andere Objekte das Objekt als Ressource noch benötigen



- Erst wenn der Zähler auf 0 fällt, wird das Objekt bzw. dessen Speicher freigegeben
- Zuvor wird automatisch die dealloc-Methode aufgerufen, damit das Objekt die Möglichkeit erhält seinerseits Ressourcen freizugeben

Objective-C: Referenzzähler (II)

- Im Umgang mit Referenzzählern kann es zu zwei möglichen Problemen kommen:
  1. Wird ein Objekt freigegeben, obwohl dieses noch benutzt wird, ist das Verhalten beim nächsten Zugriff undefiniert (genannt: Overrelease)
  2. Wenn ein Objekt freigegeben wird, dass noch eine Referenz auf ein anderes Objekt hält, diese aber nicht freigibt, wird der referenzierte Speicherplatz niemals freigegeben (genannt: Leak)



Objective-C: Verantwortung (I)

- Aus diesen Gründen ist Disziplin von Nöten: Wird ein Objekt von einem anderen benötigt, so muss es Verantwortung für dieses übernehmen
- Es gibt drei Verhaltensregeln:
  I. Ein Objekt hat automatisch die Verantwortung für jedes andere Objekt, das es per "alloc", "new" "copy" oder "mutableCopy" erzeugt
  II. Wenn die Verantwortung für ein Objekt übernommen wird, muss diese wieder abgegeben werden, wenn das Objekt nicht mehr benötigt wird
  III. Ein Objekt kann keine Verantwortung für ein Objekt abgeben, für das es keine Verantwortung hat
- Programmatisch wird die Verantwortung über retain und release geregelt:

```
MemoryObject* object = [[MemoryObject alloc] init]                  // Verantwortung durch „alloc“ (Regel I)
NSLog(@"Number of references: %lu", [object retainCount]);
[object retain];                                                    // noch mehr Verantwortung
NSLog(@"Number of references: %lu", [object retainCount]);
[object release];                                                   // Verantwortung wieder abgeben (Regel II)
NSLog(@"Number of references: %lu", [object retainCount]);
[object release]                                                    // noch mehr Verantwortung abgeben (Regel II);
```

Objective-C: Verantwortung (II)

- Das Spiel mit der Verantwortung:

```
NSMutableArray* array = [[NSMutableArray alloc] init];
[array addObject:[MemoryObject new]];
...
[array release];
```

- Verstoß gegen Regel II: Es wird „inline“ ein Objekt mit alloc angelegt, d.h. das erzeugende Objekt ist in der Verantwortung und muss sie wieder abgeben!

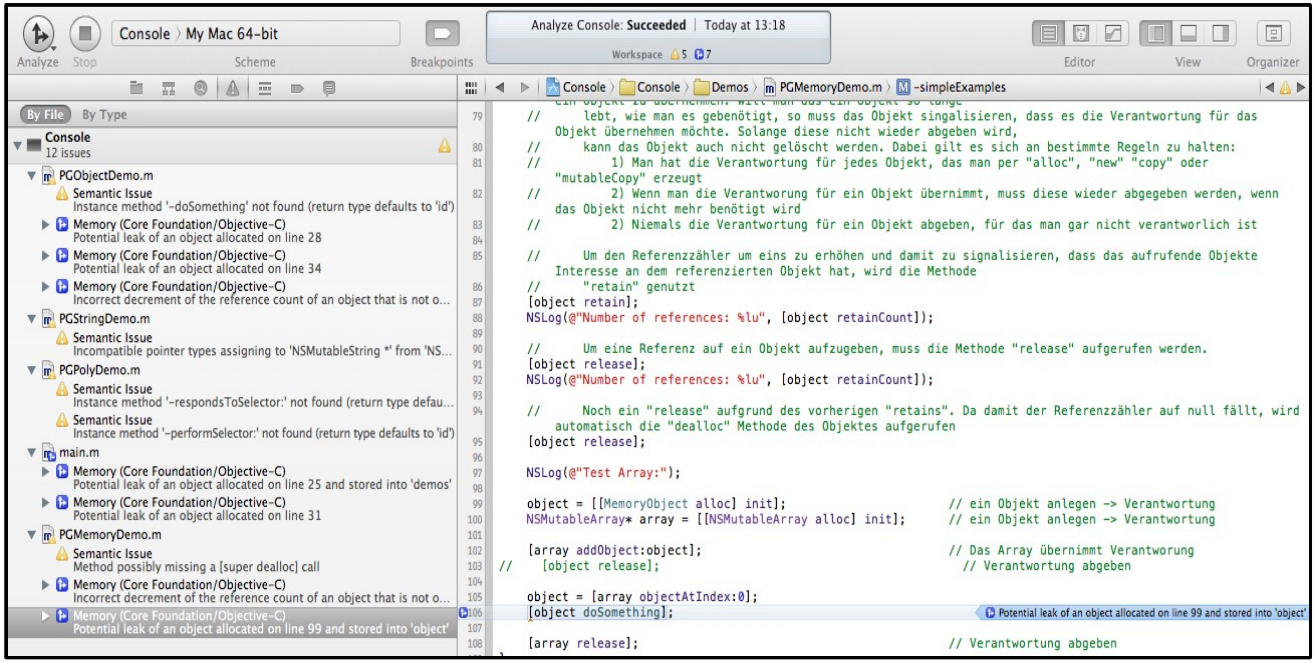
```
NSMutableArray* array = [[NSMutableArray alloc] init];
MemoryObject* object = [MemoryObject new]];
[array addObject:object];
[object release];
...
[array release];
```

- Was passiert?

```
...
[array addObject:object];
[object release];
object = [array objectAtIndex:0];
[object doSomething];
```

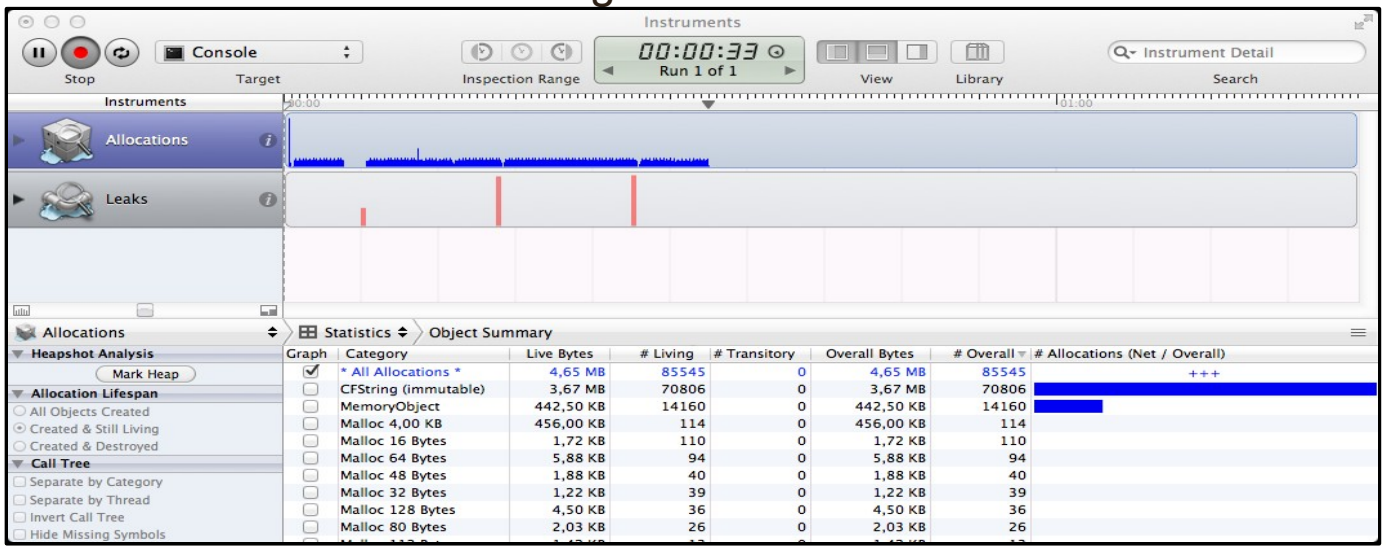
Xcode: Code-Analyse

- **Code-Analyse** (Produkt→Analyze oder „Run“-Button gedrückt halten → Analyze): Xcode analysiert den Quellcode auf das richtige Verhältnis von Erzeugung, release und retain und gibt Warnungen aus, falls ein Codefragment potentiell zu Leaks oder einem Overrelease führen könnte(!)



Xcode: Leak-Monitor

- Zum „Lieferumfang“ der Xcode-Umgebung gehört das Programm „*`Instruments`*“, welches eine Reihe von Werkzeugen bereitstellt, um ein in Ausführung befindliches Programm zu überwachen
- Der **`Leak-Monitors`** ist eines dieser Werkzeuge und hilft potentielle Speicherlecks zu entdecken
- Instruments wird über Product → Profile gestartet



Xcode: Zombies

- **`Zombies`** werden eingesetzt, um das zu frühe Freigeben eines Objektes (Overrelease) erkennen zu können:
- In der Regel führt ein Overrelease beim nächsten Zugriff auf das Objekt zu einem Absturz mit einer wenig aussagekräftigen Fehlermeldung (z.B.: `Program received signal: EXC_BAD_ACCESS`)
- Bei aktivierten Zombies werden Objekte nicht wirklich freigegeben sondern nur als „freigegeben“ markiert, so dass beim nächsten Aufruf Rückschlüsse auf Herkunft und Art des zu früh freigegebenen Objektes gezogen werden können
- Zombies sollten wegen der hohen Speicherbelastung nur zu Debugzwecken eingesetzt werden
- Zombies aktivieren: Product→Edit Scheme... und im sich öffnenden Dialog unter „Diagnostics“ „Enable Zombie Objects“ anhaken
- Auf zur Überwachung von Zombies gibt es im Instruments-Programm ein Werkzeug

Objective-C: Verantwortung und Properties

- Zur Erinnerung: Der @property-Anweisung können mit assign (weak), retain (strong) und copy Attribute zugeteilt werden, die bestimmen, wie die Setter-Methode implementiert werden soll
- Die Auswahl hat unmittelbaren Einfluss auf das Speichermanagement:
  - assign besagt, dass das zugehörige Objekt, das zugewiesene zwar kennen, aber dafür keine Verantwortung übernehmen will
  - retain veranlasst, dass das zugehörige Objekte Verantwortung für das zugewiesene Objekt übernehmen wird
  - copy besagt, dass das zugehörige Objekt zwar Verantwortung übernimmt, allerdings nicht für das übergebene, sondern für eine Kopie des übergebenen Objektes

```
@interface MemoryObject : NSObject <NSCopying> {
   MemoryObject* _buddy;
   NSString* _message;
}
```

```
-(void) setBuddyByAssign: (MemoryObject*) buddy;
-(void) setBuddyByRetain: (MemoryObject*) buddy;
-(void) setBuddyByCopy: (MemoryObject*) buddy;

@end
```

Objective-C: Verantwortung und Properties (assign)

- setBuddyByAssign entspricht @property(assign):

```
-(void) setBuddyByAssign:(MemoryObject *)buddy {
    if (_buddy == buddy) return ;
    _buddy = buddy;
}
```

- Was passiert?

```
    MemoryObject* object = [MemoryObject new];
    MemoryObject* buddy = [MemoryObject new];

    [object setBuddyByAssign:buddy];
    [buddy release];

    [object.buddy doSomething];

    [object release];
```

- Da die Assign-Methode den Referenzzähler unangetastet lässt, fällt dieser mit dem release auf dem „buddy“ Objekt auf 0. Das Objekt wird damit für den Zugriff durch doSomething zu früh freigeben.

Objective-C: Verantwortung und Properties (retain)

- setBuddyByRetain entspricht @property(retain):

```
-(void) setBuddyByRetain:(MemoryObject *)buddy {
    if (_buddy == buddy) return ;
    [_buddy release];
    _buddy = [buddy retain];
}
```

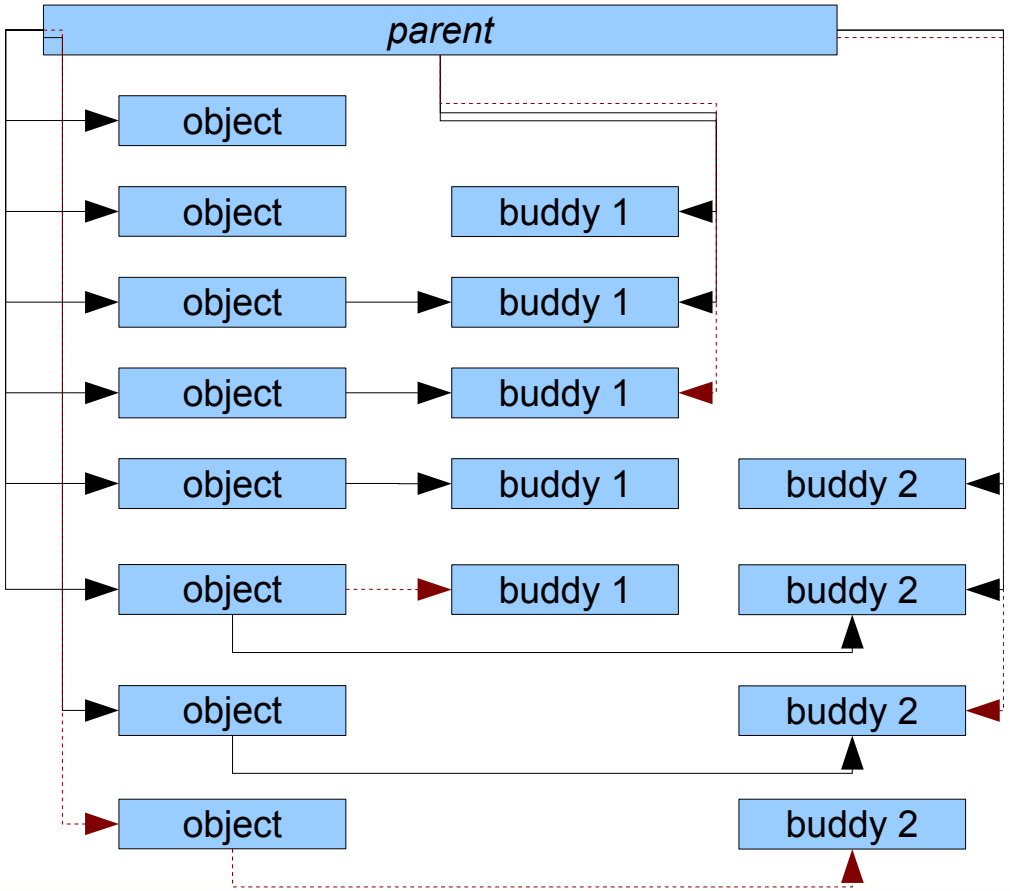
- Da das Objekt durch retain die Verantwortung für ein ihm zugewiesenes übernimmt, muss es dieses auch wieder freigeben:
  1. Wird setBuddyByRetain erneut aufgerufen muss die Verantwortung für das „alte“ Objekte abgegeben werden
  2. Wird das Objekt selbst nicht mehr gebraucht, muss es ebenfalls sein „buddy“-Objekt freigeben

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

Objective-C: Verantwortung und Properties (retain II)

- Was passiert?

```
object = [MemoryObject new];
buddy = [MemoryObject new];
[object setBuddyByRetain: buddy];
[buddy release];
buddy = [MemoryObject new];
[object setBuddyByRetain:buddy];
[buddy release];
[object release];
```



Objective-C: Verantwortung und Properties (copy)

- setBuddyByCopy entspricht @property(copy):

```
-(void) setBuddyByCopy:(MemoryObject *)buddy {
    if (_buddy == buddy) return ;
    [_buddy release];
    _buddy = [buddy copy];
}
```

- Die copy-Methode gehört zu den Vorschriften des NSCopying-Protokolls, welches implementiert werden sollte, um zu signalisieren, dass eine Klasse den Kopiermechanismus unterstützt

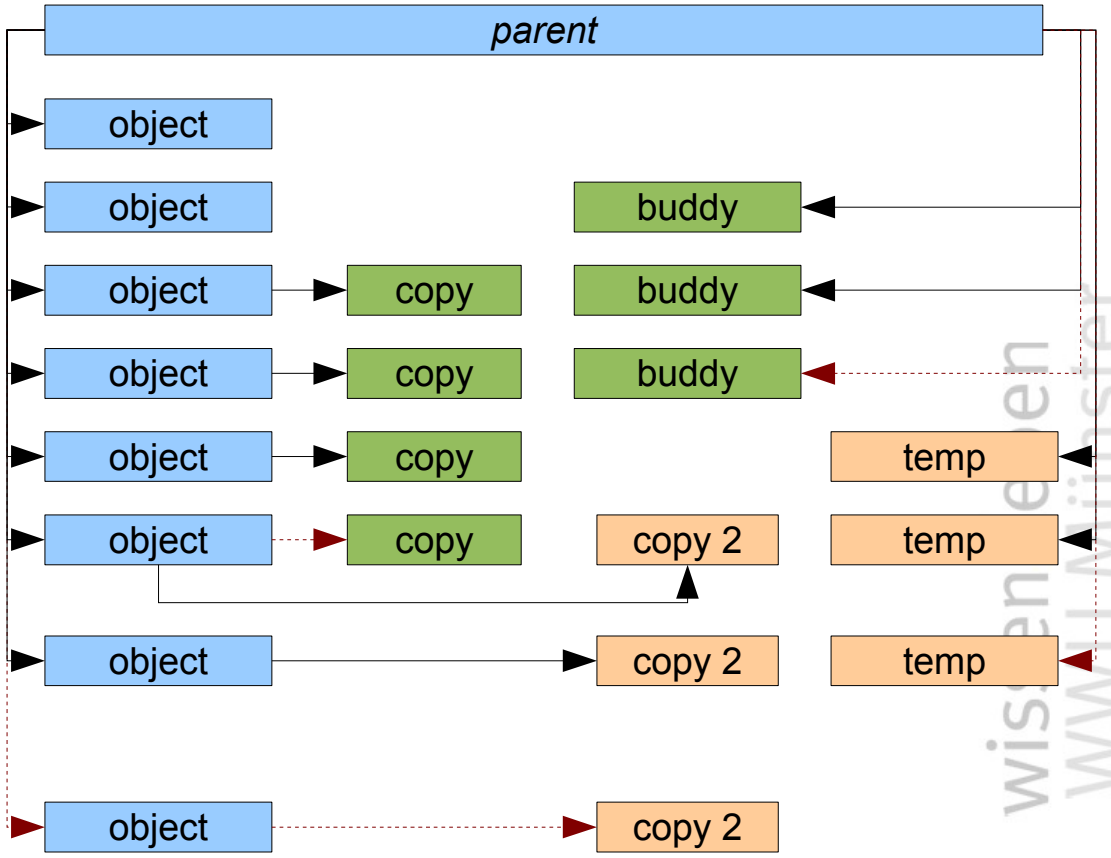
```
-(id) copy {
    MemoryObject* copy = [[self class] new];
    [copy setBuddyByCopy:_buddy];
    return copy;
}
```

- Da das Objekt die Verantwortung für die Kopien übernimmt, muss es analog zu retain dafür sorgen, dass die belegten Ressourcen wieder freigegeben werden, sobald das Objekt nicht mehr benötigt wird

Objective-C: Verantwortung und Properties (copy II)

- Was passiert?

```
MemoryObject* object = [MemoryObject new];
MemoryObject* buddy = [MemoryObject new];
[object setBuddyByCopy:buddy];
[buddy release];
MemoryObject* temp = [MemoryObject new];
[object setBuddyByCopy:temp];
[temp release];
...
[object release];
```



Objective-C: Zeiger, Methoden, Verantwortung

- Häufig wird innerhalb einer Methode ein Objekt erzeugt und als Ergebnis geliefert
- Zumeist ist es aber nicht nötig, dass das erzeugende Objekt nach dem Methodenaufruf noch verantwortlich ist
- Naiv könnte man meinen, dass ein release des Rätsels Lösung wäre:

```
-(MemoryObject*) createMemoryObject {
    MemoryObject* object = [[MemoryObject alloc] init];
    [object release];
    return object;}
```

- Warum wäre diese Lösung falsch?
- Das release würde dafür sorgen, dass der Referenzzähler auf 0 fällt; das Objekt würde freigegeben werden und zurückgeliefert würde ein Zeiger auf einen invaliden Speicherbereich (Overrlease)
- Warum überlässt man der aufrufenden Methode nicht das Freigeben?
- Nach Regel I ist dies nur erlaubt, wenn die Methode mit „alloc“, „new“, „copy“ oder „mutableCopy“. In allen anderen Fällen muss der Aufrufer nicht davon ausgehen, dass er die Verantwortung „ablösen“ muss!

Objective-C: Autorelease

- Die Lösung für das Problem ist die autorelease-Anweisung:

```
MemoryObject* object = [[MemoryObject alloc] init];
[object autorelease];
return object;
```

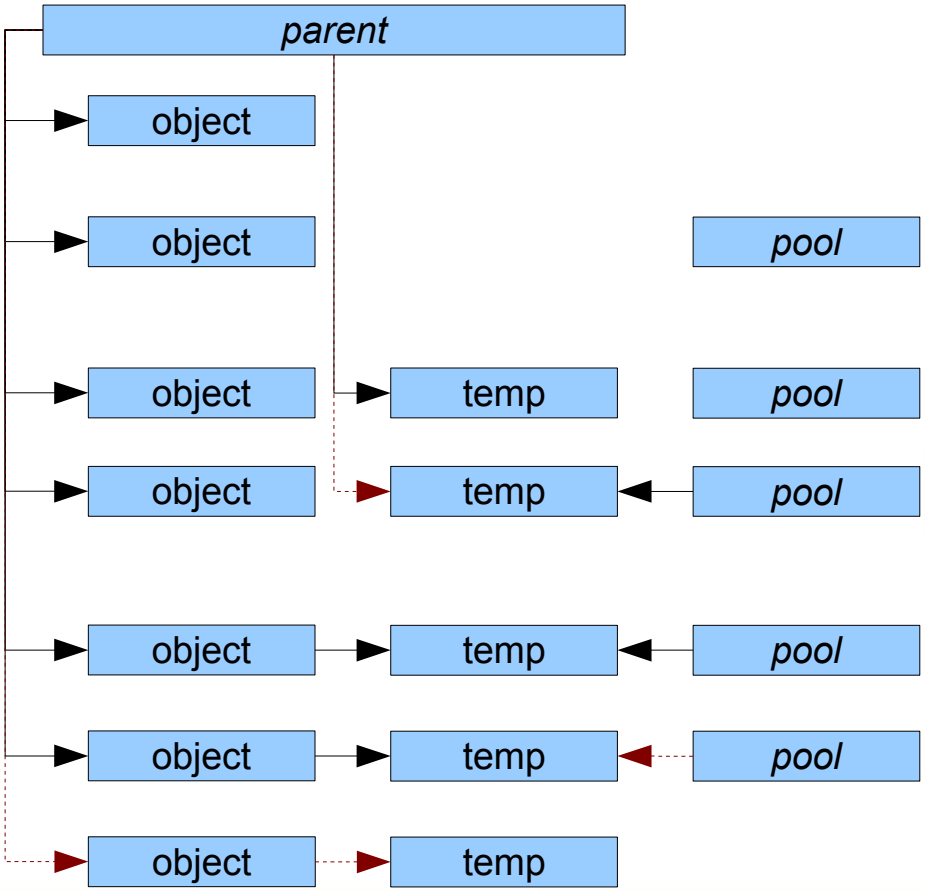
- Objekte auf denen autorelease aufgerufen wurde, werden im sogenannten „Autorelease Pool“ registriert
- Die Verantwortung für das Objekt wird auf den Pool übertragen und erst wieder abgegeben, wenn er „geleert“ wird
- Innerhalb eines Programmes (Threads) können mehrere Pools definiert werden, allerdings ist zu jedem Zeitpunkt immer nur ein Pool aktiv
- Pools sind als Stack angeordnet, d.h. der zuletzt erzeugte, ist der aktuell aktive
- Mit einem Auruf von drain wird der Pool geleert

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
MemoryObject* buddy = [self createMemoryObject];
… //
[pool drain];
```

Objective-C: Autorelease (II)

- Was passiert?

```
MemoryObject* object = [MemoryObject new];

NSAutoreleasePool *pool =
        [[NSAutoreleasePool alloc] init];

buddy = [self createMemoryObject]

MemoryObject* temp = [[MemoryObject alloc] init];

[temp autorelease];

return temp;

[object setBuddyByRetain:buddy];

[pool drain];

[object release];
```



Objective-C: Autorelease (III)

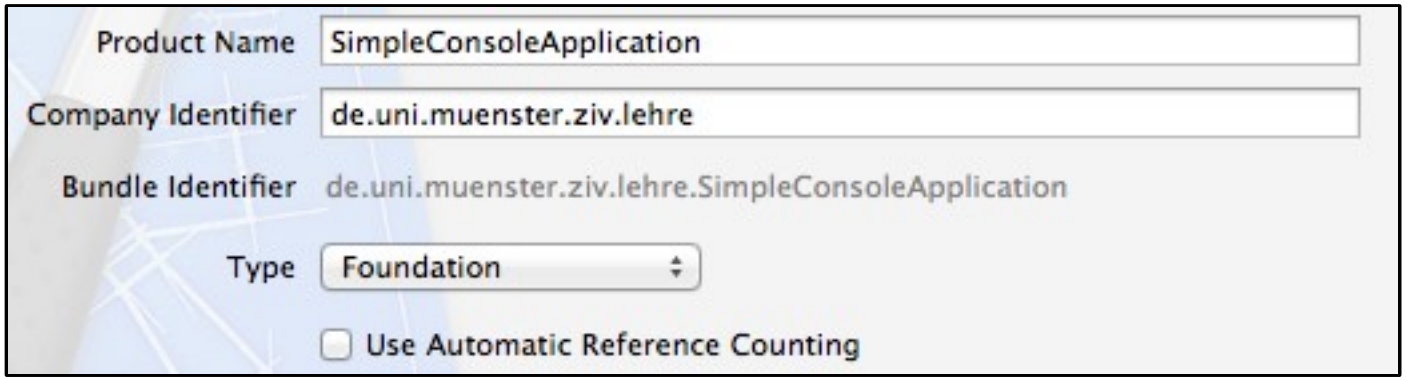
- Anstatt den Autorelease-Pool händisch zu erzeugen und zu leeren, kann auch das Schlüsselwort @autoreleasepool genutzt werden (ab iOS 4.3.x)
- Der @autoreleasepool-Notation ist man bereits in der main-Methode begegnet:

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        ....
    }
    return 0;
}
```

- Mit der schließenden }-Klammer wird der Pool automatisch geleert
- Viele Cocoa-Klassen bietet neben den „init“-Konstruktoren noch sogenannte Komfortkonstruktoren als Klassenmethoden
- Komfortkonstruktoren liefern ein neu erzeugtes Objekt, das bereits im aktuellen Autorelease-Pool registriert wurde
- Fehlerquelle: Für ein so erzeugtes Objekt muss explizit Verantwortung übernommen werden, wenn man dem Pool diese nicht überlassen will!

Objective-C: ARC

- Die manuelle Speicherverwaltung über Referenzzähler ist fehleranfällig und erfordert hohe Aufmerksamkeit im Umgang mit Referenzen
- Mit iOS 5 hat Apple seinen Objective-C Compiler (LLVM) um Automatic Reference Counting (ARC) erweitert
- Mit ARC nimmt der Compiler dem Programmier die Aufgabe ab, den Referenzzähler mit retain, release und Co. zu verwalten
- Die Zähler-Methoden sind sogar verboten (Compilerfehler)!
- ARC kann beim Erstellen eines Projektes aktiviert werden:



Objective-C: ARC (I)

- Motivationsbeispiel

```
-(void) addToArray: (NSMutableArray*) array {
    MemoryObject* object = [MemoryObject new];

    [array addObject:object];
}

-(void) motivate {
    NSMutableArray* array = [NSMutableArray new];
    [self addToArray:array];

    //...

}
```

Mit ARC

```
-(void) addToArray: (NSMutableArray*) array {
    MemoryObject* object = [MemoryObject new];

    [array addObject:object];
    [object release];
}

-(void) motivate {
    NSMutableArray* array = [NSMutableArray new];
    [self addToArray:array];

    //...

    [array release];
}
```

Ohne ARC

- Mit ARC muss weder das Objekt noch der Array manuell freigegeben werden
- Der Compiler erkennt automatisch, dass sowohl array als auch object nach der Methode nicht mehr gebraucht werden

Objective-C: ARC (II)

- Autorelease

```
-(MemoryObject*) createAutoreleased {

    return  [MemoryObject new];
}

-(void) testAutorelease {
    @autoreleasepool {
        MemoryObject* object = [self createAutoreleased];

        //...
    }
}
```

Mit ARC

```
-(MemoryObject*) createMemoryObject {
    MemoryObject* object = [[MemoryObject alloc] init];
    [object autorelease];
    return object;
}

-(void) motivate {
    @autoreleasepool {
        MemoryObject* object = [self createAutoreleased];

        //...
    }
}
```

Ohne ARC

- Mit ARC muss das Objekt vor der Rückgabe nicht eigenständig zu einem Pool hinzugefügt werden

Objective-C: ARC (III)

- Bei der Einführung zu Properties wurde bereits erwähnt, dass es mit strong und weak zwei zu retain und assign synonyme Schlüsselwörter gibt
- Wird ARC eingesetzt gibt es allerdings einen kleinen, aber wichtigen Unterschied: Sobald ein Objekt, das mit strong oder weak referenziert wurde, freigegeben wird (Referenzzähler = 0), wird die Referenz auf nil gesetzt!

```
@interface MemoryObject : NSObject

@property(strong) MemoryObject* buddy;
@property(weak) MemoryObject* stranger;

@end
```

```
MemoryObject* object = [MemoryObject new];
@autoreleasepool {
    MemoryObject* buddy = [MemoryObject new];
    MemoryObject* stranger = [MemoryObject new];

    object.buddy = buddy;
    object.stranger = stranger;
}
[object.stranger doSomething]; // stranger == nil!
```

- Da object außerhalb des Pools definiert wurde, überlebt auch das mit strong referenzierte buddy
- stranger ist hingegen weak definiert, so dass der Compiler davon ausgeht, dass das Objekt nach dem Pool freigegeben werden kann

Objective-C: ARC (IV)

- Nicht jede Variable ist auch gleichzeitig ein Attribut und zudem muss nicht jedes Attribut als Property definiert sein. Daher gibt es neben den Property-Anweisungen noch die Schlüsselwörter __strong und __weak für den Umgang mit Variablen

```
MemoryObject* __strong strongPtr;
MemoryObject* __weak weakPtr;

@autoreleasepool {
    MemoryObject* strongObject = [self createAutoreleased];
    MemoryObject* weakObject = [self createAutoreleased];

    strongPtr = strongObject;
    weakPtr = weakObject;
}

[strongPtr doSomething];
[weakPtr doSomething];
```

- Da strongPtr mit __strong definiert wurde, wird bei der Zuweisung der Referenzzähler erhöht, weakPtr hingegen überlebt den Pool nicht.