Jekyll theme for documentation — mydoc product

version 6.0

Last generated: October 07, 2019

Company Logo

Table of Contents

Acknowledgments

De Re MulleObjC

Summary: What you need to know to use MulleObjC to its full potential.

Note: This is a draft/work in progress. Stuff will appear and disappear haphazardly. Some of the content is only applicable to mulle-objc 9.0.0, which has not been released as of this writing.

Why Objective-C ?

If you are comfortable writing in C, you will notice that C is fine until you reach a certain level of complexity. If you don't want to spend your lifetime in C++ beaureaucracy, Objective-C is the answer. It is:

- Easy to learn
- Fun
- No magic
- A complexity manager
- A dynamic messenger
- A powerful class-system

Install mulle-objc

This guide doesn't have detailed installation instructions for **mulle-objc**. Read the instructions on foundation-developer and follow them.

Afterwards you should have **mulle-clang** and **mulle-sde** in your PATH.

```
mulle-clang --version
mulle-sde --version
```

Development

Objective-C is an Object Oriented Programming Language . With that comes the expectation of a plug-n-play programming environment. It should be possible to add and remove functionality, without breaking the application. This expectation has been historically never fulfilled, due to deficiencies with the compilation tools, the Objective-C runtime and the way headers are handled.

Modern Workflow

The modern workflow provides such a plug-n-play environment.

If you are a complete newbie learn Objective-C (page 0) with the modern workflow (page 0).

If you want to experience the full power MulleObjC has to offer, then use the modern workflow (page 0).

Legacy Workflow

If you are a seasoned Objective-C programmer, who would like to try porting some of his existing code to mulle-objc, then start with the legacy workflow (page 0).

Next

The next step is to Learn Objective-C (page 0). Or if you already know the language, skip to Basics (page 0)

Porting unichar

Summary: How classes are loaded and how to orchestrate them.

Apple Foundation uses UTF-16 as `unichar`, whereas the MulleObjC Foundation used UTF-32 as `unichar`. As long as your code is not assuming 16-bit for its size, there should be no problem.

When accessing string contents as `unichar *` with `dataUsingEncoding:` use the generic `NSUnicodeStringEncoding` instead of `NSUTF32StringEncoding` / `NSUTF16StringEncoding`.

```
  data = [s dataUsingEncoding:NSUnicodeStringEncoding];
  here_some_unichars( (unichar *) [data bytes], [data length]
/ sizeof( unichar));
```

TODO: How about `printf` with `%S` ?

-finalize

`-finalize` runs automatically before `-dealloc`

When the `-retainCount` is decremented to zero via `-release`, an object gets the `-finalize` message first before `-dealloc`. If the retainCount remains unchanged throughout `-finalize`, then `-dealloc` is called. It is guaranteed that `-finalize` is only called once.

During `-finalize` all **@properties** with pointers (e.g. `void *`) or objects (e.g. `NSArray *`) will be cleared.

So `-finalize` is used to free resources and cut links to other objects. Objects that are cleared by `-finalize` will be released with `-autorelease` and not with `-release`.

`-dealloc` will ideally at most contain `-release` calls and `[super dealloc]`. Anything else can be done in `-finalize`. The idea is, that a finalized object is still useable in the object hierarchy, but not active anymore. A good example, where this is useful, is a window controller, where the window close button has been clicked. It may still redraw, but it doesn't react to any event actions any more.

`-performFinalize` runs `-finalize` on demand

You can use `-performFinalize` to finalize an object "manually", but you shouldn't call `-dealloc` manually on a reference counted object.

> ℹ **Note:** Never call `-finalize` directly, always use `-performFinalize`.

Your `-finalize` must call `[super finalize]`

If you write a `-finalize` method call `[super finalize]` so that NSObject can clean up any properties.

Write `-finalize/-dealloc` portably

If you use -finalize, you will be incompatible with non-ARC Apple. This can be remedied, by structuring your `-finalize/-dealloc` code like this:

```
- (void) _finalize
{
}

- (void) finalize
{
  [self _finalize];

   [super finalize];
}

- (void) dealloc
{
#ifndef __MULLE_OBJC__
   [self _finalize];
#endif
   [super dealloc];
}
```

Caveat

`-finalize` is single-threaded, just like `-init` and `-dealloc` when called during `release`. When you `-performFinalize` it can only be guaranteed that no other thread will be executing `-finalize` (ever). But it is *not guaranteed* that no other thread is accessing the object.

ℹ Note: What's the deal with the `clearer` in `struct _mulle_objc_property` ? It's an optimization so that the clearer code doesn't need to run for objects, that have no usefully clearable properties.

New Property attributes

dynamic

Indicate that a property is not backed by an ivar. This is useful for forwarding properties.

```
@property( copy, dynamic) NSString   *bar;
```

E.g.

```
- (void) setBar:(NSString *) s
{
   [_other setBar:s];
}
```

serializable

Indicate that a property should be serialized by `NSCoder`, MulleEOF or some other persistence method. Then `-initWithCoder:`, `-encodeWithCoder:` do not need to be written but can be inherited from the `NSCoder` protocol running on properties only. The optional value can be used to indicate the destination class.

```
@property( assign, serializable=Bar) NSArray   *foo;
```

Protocolclasses allow default implementations

Summary: Protocolclasses allow default implementations

A *protocolclass* is an extension to the Objective-C language. It allows a class to inherit default implementations from a protocol.

`@protocolclass` as a keyword doesn't exist yet, but it's effect can be simulated by: `@class foo; @protocol foo; @end; @interface foo <foo>; @end` .

For a class to become a protocolclass it must meet the following requirements:

- it must be a root class (not inherit from another class)
- it must implement the protocol of the same name
- it must not implement any other protocols
- it must not have instance variables

Also a protocolclass can not have any categories, this isn't enforced, but categories on protocolclass aren't used by the runtime for method lookup. The protocol of the protocolclass can adopt other protocols.

Creating a protocolclass

Lets's write a *protocolclass* with everything in it. Let's assume we do need some internal storage in the protocol. We define this with a 'C' struct `FooIvars` .

`Foo.h` :

```
struct FooIvars
{
   int   a;
};

@class Foo;
@protocol Foo
- (struct FooIvars *) getFooIvars;
@optional
- (void) doTheFooThing;
@end

@interface Foo <Foo>
@end
```

`Foo.m`:

```
#import "Foo.h"

@implementation Foo

- (void) doTheFooThing
{
  struct FooIvars   *ivars;

  ivars = [self getFooIvars];
  ivars->a += 1848;
}

@end
```

Supplement an existing protocol with protocolclass methods

When you do this, your protocolclass should adopt that other protocol and redeclare those methods, for which implementations are provided as `@optional`.

> ℹ **Note:** Redeclaration as optional is a mulle-objc specific feature.

Use of PROTOCOLCLASS macros

`PROTOCOLCLASS` macros can make your life a little easier. The above example would be transformed with macros into:

`Foo.h`:

```
struct FooIvars
{
   int   a;
};

PROTOCOLCLASS_INTERFACE0( Foo)
- (struct FooIvars *) getFooIvars;
@optional
- (void) doTheFooThing;
PROTOCOLCLASS_END()

`Foo.m`:

```objective-c
#import "Foo.h"

#pragma clang diagnostic ignored "-Wobjc-root-class"
PROTOCOLCLASS_IMPLEMENTATION( Foo)
PROTOCOLCLASS_END()
```

Due to deficiences in the way variadic arguments are handled in C macros, you must use `PROTOCOLCLASS_INTERFACE0` instead of `PROTOCOLCLASS_INTERFACE`, if your protocolclass adopts no further protocols.

Calling NSObject methods from your protocolclass methods

If your protocolclass wants to use NSObject methods, it should declare this in its protocol.

`Foo.h`:
```

```
#import <Foundation/Foundation.h>

PROTOCOLCLASS_INTERFACE( MyProtocolClass, NSObject)
@optional
- (void) doSomethingWithObject:(id) object;
PROTOCOLCLASS_END()
```

This will create a lot of unimplemented method warnings though. They are usefully turned off with a `#pragma` :

`Foo.m` :

```
#import "Foo.h"

#pragma clang diagnostic ignored "-Wprotocol"
#pragma clang diagnostic ignored "-Wobjc-root-class"

PROTOCOLCLASS_IMPLEMENTATION( Foo)
- (BOOL) doSomethingWithObject:(id<MyProtocolClass>) object
{
   return( [object isKindOfClass:[NSString class]]);
}
PROTOCOLCLASS_END()
```

Using the protocolclass

Using the protocolclass is very simple. You just adopt the protocol.

`MyClass.h` :

```
#import "Foo.h"

@interface MyClass : NSObject < Foo>
{
  struct FooIvars   ivars;
}
@end
```

`MyClass.m` :

```
#import "MyClass.h"

@implementation MyClass

- (struct FooIvars *) getFooIvars
{
   return( &ivars);
}
@end
```

Now users of your class can call `-doTheFooThing`. Protocolclasses become a time and space saver, if multiple classes adopt them.

Things to watch out for

Reaching the protocolclass via super

In the class adopting your protocol, a call to super will first search the protocolclasses in order of adoption then the superclass, in the case `MyClass` that is `Foo` first then `NSObject`.

Do not use @property in your protocol (yet)

This will create an instance variable in your root class, which taints it. In the future this will be remedied by:

- use `dynamic` property attribute
- use `propertyclass` keyword

Calling super from the protocolclass

> ❗ **Warning:** Not recommended!

As a protocolclass is a root class, there is no way to call **super** from a protocolclass. There is a way around this though.

You can search for the overridden implementation of a selector, given the class and category of the implementation.

```
   IMP   imp;

   imp = MulleObjCSearchOverriddenIMP( self, @selector( doTheFo
oThing), @selector( Foo), 0);
   return( (*imp)( self, @selector( doTheFooThing), self));
```

This works fine, unless the receiver is implementing the same protocolclass again and is calling super.

classcluster

Summary:

UNTESTED DRAFT

A classcluster is a fairly complicated setup, that is used to hide implementation details of various classes under one common name. **mulle-objc** simplifies the setup, but it is by no means simple.

As an example, we want to create a classcluster for a **BitSet** class. We assume, that in the majority of cases, the bitset will be empty. So as an optimization we want to provide a special **EmptyBitSet** class, to conserve space. Otherwise

We also want to have a mutable variant **MutableBitSet**, that is a subclass of **BitSet** (and not the other way around, like in Apple Foundation).

The base class

This is the “user” facing class. All other classes will be more or less hidden. This class defines the API. It also adopts the MulleObjCClassCluster protocolclass.

```
@interface BitSet : NSObject < MulleObjCClassCluster>

- (instancetype) initWithBits:(NSUInteger *) bits
                        count:(NSUInteger) count;
@end

// methods to be implemented by classcluster classes
@interface BitSet( ClassCluster)
- (BOOL) boolAtIndex:(NSUInteger) index;
@end
```

The implementation now either produces an **EmptyBitSet** or a **ConcreteBitSet**, depending on all input bits being clear or not:

```
#import "BitSet.h"

@implementation BitSet

- (instancetype) init
{
    return( [[EmptyBitSet sharedInstance] retain]);
}

- (instancetype) initWithBits:(NSUInteger *) bits
                        count:(NSUInteger) count
{
   NSUInteger   *p;
   NSUInteger   *sentinel;

   p        = bits;
   sentinel = &p[ count];
   while( p < sentinel)
      if( *p++)
         return( [ConcreteBitset newWithBits:bits
                                       count:count]);
   return( [[EmptyBitSet sharedInstance] retain]);
}

@end
```

Some notes on the implementation.

As we are implementing a classcluster, we know that the `-initWithBits:count:` method is operating on a special kind of object, the placeholder. A placeholder is a constant instance of **BitSet** in our case. A constant instance ignores all `-retain` / `-release` calls, so we do not need to `-[self release]` in `-initWithBits:count:` to avoid leaks.

We are using a shared instance for **EmptyBitSet** to reduce the footprint of the application. Only one instance of this immutable bitset will be generated.

ℹ Note:

The concrete classes

The **EmptyBitSet** is very simple, as the instance creation is done with `+sharedInstance` provided by `MulleObjCSingleton` already:

```
#import "BitSet.h"

@interface EmptyBitSet : BitSet <MulleObjCSingleton>

- (BOOL) boolAtIndex:(NSUInteger) index;

@end
```

```
#import "EmptyBitSet.h"

@implementation EmptyBitSet

- (BOOL) boolAtIndex:(NSUInteger) index
{
    return( NO);
}
@end
```

```
#import "BitSet.h"

@interface ConcreteBitSet : BitSet
{
    NSUInteger   *_bits;
    NSUInteger   _count;
}

- (BOOL) boolAtIndex:(NSUInteger) index;

@end
```

In the case of **ConcreteBitSet** we need to be aware that we are subclassing the placeholder class **BitSet**. So `+alloc` will just produce the same placeholder object that 'self' already is. We therefore need to implement the instance creation and deallocation with primitive runtime functions ourselves:

```
#import "ConcreteBitSet.h"

@implementation ConcreteBitSet

- (id) newWithBits:(NSUInteger *) bits
             count:(NSUInteger) count
{
    ConcreteBitSet  *set;
    size_t          size;

    set         = NSAllocateObject( self, 0, NULL);
    size        = sizeof( NSUInteger) * count;
    set->_bits  = MulleObjCObjectAllocateNonZeroedMemory( set,
size);
    set->_count = count;
    memcpy( set->_bits, bits, size);
    return( set);
}

- (void) dealloc
{
   MulleObjCObjectDeallocateMemory( self->_bits);
   NSDeallocateObject( self);
}

- (BOOL) boolAtIndex:(NSUInteger) index
{
   NSUInteger  i;
   NSUInteger  bit;

   i   = index / (sizeof( NSUInteger) * CHAR_BIT);
   bit = 1 << (index & (sizeof( NSUInteger) * CHAR_BIT - 1));

   if( i >= _count)
      return( NO);
   return( _bits[ i] & bit ? YES : NO);
}

@end
```

Adding a class to a classcluster

You chance upon the CountOnes project and its AVX2 implementation and would like to support it with a class `ConcreteAVX2Bitset` for larger bitsets, when AVX2 is available.

You could then expand the current init method in **BitSet**:

```
- (instancetype) initWithBits:(NSUInteger *) bits
                        count:(NSUInteger) count
{
   NSUInteger   *p;
   NSUInteger   *sentinel;

   p        = bits;
   sentinel = &p[ count];
   while( p < sentinel)
      if( *p++)
         if( count >= 16)
             return( [ConcreteAVX2Bitset newWithBits:bits
                                               count:count]);
         else
             return( [ConcreteBitset newWithBits:bits
                                           count:count]);
   return( [[EmptyBitSet sharedInstance] retain]);
}
```

Or you could add a new `-initWithAVX2Bits:count:` method. The advantage of using another `-init` methods is, that you can add it to your classcluster using a category:

```
@implementation BitSet( ConcreteAVX2Bitset)

- (instancetype) initWithAVX2Bits:(NSUInteger *) bits
                            count:(NSUInteger) count
{
   return( [ConcreteAVX2Bitset newWithBits:bits
                                     count:count]);
}
```

Subclassing a classcluster

The main obstacle to subclassing a classcluster is the reimplementation of the instance allocation methods. You should reimplement the following methods in your subclass:

```
+ (instancetype) alloc
{
   return( NSAllocateObject( self, 0, NULL));
}


+ (instancetype) allocWithZone:(NSZone *) zone
{
   return( NSAllocateObject( self, 0, NULL));
}


+ (instancetype) new
{
   return( [NSAllocateObject( self, 0, NULL) init]);
}
```

Now your subclass and its subclass will create instances of the proper class. But you will also need to override the init functions of the classcluster. In the case of **BitSet** there is only one `-initWithBits:count:` , which makes this easy.

Classcluster on top of classcluster

It’s entirely possible to create a classcluster on top of another classcluster, as we will show with **MutableBitSet**.

```
#import "BitSet.h"

@interface MutableBitSet : BitSet <MulleObjCSingleton>

- (BOOL) setBool:(BOOL) flag
         atIndex:(NSUInteger) index;
- (BOOL) boolAtIndex:(NSUInteger) index;

@end
```

The main thing we will have to override is the init method, as we will have to use different classes. Because I am extremely lazy, I will restrict the code to just one class:

```
#import "MutableBitSet.h"

@implementation MutableBitSet

- (instancetype) initWithBits:(NSUInteger *) bits
                        count:(NSUInteger) count
{
   return( [ConcreteMutableBitset newWithBits:bits
                                        count:count]);
}
@end
```

Singleton classes

Summary:

Create a Singleton class

To create a singleton class that instantiates via `+sharedInstance`, you merely adopt the **MulleObjCSingleton** protocol and you are done.

```
@interface Foo : NSObject < MulleObjCSingleton>
@end

@implementation Foo
@end
```

That's because **MulleObjCSingleton** is a protocolclass (page 9).

If you subclass a singleton class, your subclass should also adopt **MulleObjCSingleton**. Now your class will create a new shared instance. Otherwise your subclass would be ignored by `sharedInstance`. Your subclass singleton will now coexist with the base class singleton.

Modifications

Change name of the initializer

If you want to use a different name than `+sharedInstance` add this to your class:

```
+ (instancetype) myInit
{
   return( MulleObjCSingletonCreate( self));
}
```

Init the singleton differently

Usually the singleton will be allocated with `-init`. You can override `-init` for the singleton instance with `-__initSingleton`. This could be useful, if your singleton needs a special setup and you have a not-so-true singleton class like **NSNotificationCenter**, which can instantiate other instances.

Prevent other instances of the same class

"Poison" the `-init` method by returning the singleton instead. Initialize the singleton with `-__initSingleton`. As a result you can now expect to be able to use pointer equality for `-isEqual:` comparisons.

```
- (id) init
{
   Class   cls;

   cls = MulleObjCGetClass( self);
   [self release];
   return( [MulleObjCSingletonCreate( cls) retain]);
}

- (id) __initSingleton
{
   return( self);
}
```

Technical considerations

Infinitely retained

A singleton is infinitely retained. If you know you are dealing with a singleton, you do not have to retain or release this. It isn't really recommended to exploit this, as it makes the code more brittle.

Thread safety

The actual creation of the singleton instance is thread safe, there are no duplicate instances in multiple threads.

Testing

Usually singletons aren’t releases. Under a test environment, the universe (page 9) will be shutdown in an orderly fashion and your singleton will be release.

If you want to leak-check within the test though, the singleton will appear as a leak. To circumvent this you can set the environment variable `MULLE_OBJC_EPHEMERAL_SINGLETON` to YES. The singleton will be now be created in a thread-unsafe manner and will only last as long as the enclosing **NSAutoreleasePool**.

forward

Summary: MulleObjC forward speed makes object composition an alternative

ℹ Note: See Objective-C Runtime Programming Guide for more information about `NSInvocation` and forwarding in general. Most if not all of it is applicable to MulleObjC

Introduction

One of the “Patterns” of OO programming is composition. Basically you add references to other objects into your object and forward messages to them. An example of composition would be this:

```
@interface MyOrderedDictionary : NSDictionary
{
   NSDictionary   *_other;
   NSArray        *_order;
}
- (NSUInteger) count;
- (id) objectAtIndex:(NSUInteger) index;
- (id) objectForKey:(id <NSCopying) key;

@end
```

```
@implementation MyOrderedDictionary

- (NSUInteger) count
{
   return( [_order count]);
}

- (id) objectAtIndex:(NSUInteger) index
{
   return( [_order objectAtIndex:index]);
}

- (id) objectForKey:(id <NSCopying) key
{
   return( [_other objectForKey:key]);
}

@end
```

This is nice but can get tedious if you need to implement a lot of methods.

Less typing with forwarding and NSInvocation

Once you have this code in place, you can forward all unknown method calls to the array instance but send `-objectForKey:` to the dictionary instance:

```
@implementation MyOrderedDictionary

- (NSMethodSignature *) methodSignatureForSelector:(SEL) sel
{
   if( sel == @selector( objectForKey:)
      return( [_other methodSignatureForSelector:sel]);
   return( [_order methodSignatureForSelector:sel]);
}


- (void) forwardInvocation:(NSInvocation *) anInvocation
{
   if( [anInvocation selector] == @selector( objectForKey:)
      [anInvocation setTarget:_other];
   else
      [anInvocation setTarget:_order];
   [anInvocation invoke];
}

@end
```

This is still a lot of typing, it’s also slow. It’s not as slow as in the Apple runtime (ca. 10* faster) but still quite slow.

Even less typing with forward:

As selectors are a type of integer in **mulle-objc**, we can even use a switch here:

```
@implementation MyOrderedDictionary

- (void *) forward:(void *) _param
{
   switch( _cmd)
   {
   case @selector( objectForKey:) :
      target = _other;
      break;

   default:
      target = _order;
      break;
   }
   return( mulle_objc_object_call( target, (mulle_objc_methodi
d_t) _cmd, _param));
}

@end
```

This is way faster than using NSInvocation and the difference to a handcoded forward method as in the introduction is minimal. This makes composition feasible.

tps

Summary:

Note: If you are using the **MulleFoundation**, then tagged pointers will be used for **NSNumber** and **NSString** leaving little (64 bit) or no (32 bit) room for own tagged pointer classes.

Create a tagged pointer (TPS) class

A tagged pointer class is great, if you have a lot of instances that are extremely small. Let's say you want to encode a 24 bit color in a tagged pointer, here's how to do it.

Choose a free index

On 32 bit you have 1-3 available, on 64 bit it is 1-7. You can use this function to search for a free index:

```
i =  mulle_objc_universe_search_free_taggedpointerclass( univer
se);
if( ! i)
   return( i);
```

Let the TPS class inherit your class

Subclass your color class and adorn your `@interface` declaration with the `MulleObjCTaggedPointer` protocol: Your class must be abstract and not contain any instance variables. Use `#ifdef __MULLE_OBJC_TPS__` around your code, as the user can turn off TPS code with a compiler option. Your code should run fine without TPS enabled (just don't use the TPS class then).

```
#ifdef __MULLE_OBJC_TPS__

@interface MyTPSColor : MyColor <MulleObjCTaggedPointer>
@end

#endif
```

Create a +load method

This will hookup your class into the TPS system at runtime. It's assumed you are using a fixed number scheme here and the TPS index chosen is '3'.

```
#ifdef __MULLE_OBJC_TPS__

@implementation MyTPSColor

+ (void) load
{
   if( MulleObjCTaggedPointerRegisterClassAtIndex( self, 0x3))
   {
      perror( "Need tag pointer aware runtime for MyTPSColor wi
th empty slot #3\n");
      abort();
   }
}
@end

#endif
```

Create TPS instance depending on input

Convert you color to a 24 bit value and create the instance with the `C` function `MulleObjCCreateTaggedPointerWithUnsignedIntegerValueAndIndex`. Do not use `+alloc`!

```
static inline MyColor   *TPSColorNew( unsigned char r, unsigne
d char g , unsigned char b)
{
   NSUInteger   value;

   value = (((NSUInteger) << 16) | ((NSUInteger) g << 8) | (NSU
Integer) b);
   return( (MyColor *) MulleObjCCreateTaggedPointerWithUnsigned
IntegerValueAndIndex( value, 0x3));
}
```

Retrieve value from TPS instance

```
@implementation MyTPSColor

...

- (unsigned char) getRedComponent
{
   NSUInteger   value;

   value = MulleObjCTaggedPointerGetUnsignedIntegerValue( sel
f);
   return( value >> 16);
}
```

And that’s about it.

unload

Summary:

NOTHING HERE YET

kvc

Summary:

NOTHING HERE YET

universe

Summary:

NOTHING HERE YET

subscripting

Summary:

The use of [] to index into an Objective-C array like into a C array is known as “subscripting”. It will never be supported by **mulle-objc**, because it introduces an un-C like ambiguity. This also precludes subscripting for dictionaries.

Translate array[0]

Use `[array objectAtIndex:0]` or the MulleObjC shortcut `[array :0]`. The latter will make your code incompatible with other runtimes though.

Translate dictionary[@”key”]

Use `[dictionary objectForKey:@"key"]` or the MulleObjC shortcut `[dictionary :@"key"]`. The latter will make your code incompatible with other runtimes though.

ARC

Summary: This is a list of porting tips for ARC code

ARC as a technology is not available in **mulle-objc** and never will be.

Ideally though, code should remain functional in ARC but work flawlessly in MulleObjC.

Use convenience constructors

Outside of `-init` and `-dealloc`, replace `[[obj alloc] init]` calls with convenience constructors like `+[NSArray array]`, if available.

Create your own convenience constructors

If a convenience constructor is not available, it might be useful to create your own with a category. Consider this if there is a lot of calls for the same class/method combination.

This is the code to replace a `[[Foo alloc] initWithRandomNumber]` with `[Foo fooWithRandomNumber]`:

```
@interface Foo( Convenience)

+ (instancetype) fooWithRandomNumber;

@end


@implementation Foo( Convenience)

+ (instancetype) fooWithRandomNumber
{
   id   obj;

   obj = [[Foo alloc] initWithRandomNumber:rand()];
#if ! __has_feature(objc_arc)
   obj = [obj autorelease];
#endif
   return( obj);
}
@end
```

Wrap alloc/init calls

You could also use this idea to wrap your `[[obj alloc] init]` code

```
#if __has_feature(objc_arc)
# define AUTORELEASE( x)  x
#else
# define AUTORELEASE( x)  NSAutoreleaseObject( x)
#endif
```

So you can simplify the above written `+fooWithRandomNumber` like this:

```
+ (instancetype) fooWithRandomNumber
{
   return( AUTORELEASE( [[Foo alloc] initWithRandomNumber:ran
d()] ));
}
```

Add `[super dealloc]` to `-dealloc`

You could use this idea to modify your `-dealloc` code

```
#if __has_feature(objc_arc)
# define SUPER_DEALLOC()
#else
# define SUPER_DEALLOC()  [super dealloc]
#endif
```

```
- (void) dealloc
{
   SUPER_DEALLOC();
}
```

Fix convenience constructors in -init

```
- (id) init
{
   self = [super init];
   if( self)
      _foo = [NSArray array];
   return( self);
}
```

Here an instance variable is initialized with an autoreleased `NSArray`, which will soon be unavailable.

Write `_foo = [[NSArray alloc] init];` to make your code ARC and MulleObjC compatible.

Release instance variables manually

There is often no `-dealloc` method in ARC code. That is fine if the class has only properties. Then MulleObjC will clean up automatically. If your class has non-property instance variables, they must be released in `-dealloc` or `-finalize`.

Since `-finalize` isn't used in ARC code, it can be a good place to do it. Othewise you could use `#if __has_feature( objc_arc)` in `-dealloc` .

```
#ifdef __MULLE_OBJC__
- (void) finalize
{
   [_foo autorelease];
   _foo = nil;

   [super finalize];
}
#endif
```

Remember to use `-autorelease` instead of `-release` . Also `nil` out the instance variable in `-finalize` .

ℹ Note:

^blocks

Summary: There are no blocks and never will be

Disable blocks

A good first step is to wrap all method declarations and definitions with

```
#ifdef __has_extension(blocks)
#endif
```

If you are lucky, blocks are just a non-integral,value-added feature in the ported library.

Rewrite

Generally at this point, you should check *how much* blocks code there is. If it is used only a few places, here are some ideas how to convert the code for MulleObjC.

Transform to NSInvocation

Blocks that are stored for later execution, are basically a form of `NSInvocation`. Transform your blocks code into a method and create an `NSInvocation` for it.

Transform to C function

If the block is used immediately, perhaps to map it to an NSArray, extract the code into a C function. Encapsulate the parameters into a C struct. No

Objective-C++

Summary: It's just not a good idea

Note: C++ is already the most complex language in the world and adding Objective-C on top of it, is like the worst of both worlds.

Use C++ from Objective-C

Create a C code wrapper to call the C++ functions. Then call the C code from Objective-C.

C++

`cpp.h:`

```
#ifdef __cplusplus
extern "C"
{
#endif
   void  call_cpp1( char *);
   char  *call_cpp2( void);
#ifdef __cplusplus
};
#endif
```

Objective-C

`foo.m`

```
#include <cpp.h>

@implementation Foo

- (void) callCPlusPlus1:(char *) s
{
   call_cpp1( s);
}

- (char *) callCPlusPlus2
{
   return( call_cpp2);
}

@end
```

Use Objective-C from C++

This is possible too, but you need to link against the `mulle-objc-runtime.h` only:

> TODO: test this does this work with mulle-c11 ?

```
#ifdef __cplusplus
extern "C"
{
#include <mulle-objc/mulle-objc-runtime.h>
};
#endif
```

Now you can use the runtime functions to create instances and call them. It’s very cumbersome though.

. syntax for properties

Summary: It's gone

Hopefully there will be a code-conversion tool in the future, but for now translate dot syntax to Objective-C calls.

e.g.

```
- (void) setup
{
   self.propertyA         = 0;
   self.propertyB.numberC = 1;
   self.propertyB.numberD = 2;
}
```

```
- (void) setup
{
   id   propertyB;

   [self setPropertyA:0];

   propertyB = [self propertyB];
   [propertyB setNumberC:1];
   [propertyB setNumberD:2];
}
```

This is also better code.

Porting @import

Summary:

You will have to use `#import` instead.

Porting @package

Summary:

Replace with `@public`.

Protocols are a kind of @selector

Summary: Protocols are a kind of @selector and not a kind of class

This will be tricky, but it is a very rare occurence. In Objective-C you can actually treat a **@protocol** as an object and assign it to `id`. You can not in MulleObjC.

There are no good tips for this yet.

Porting @property

Porting properties in a fashion that works in ARC code and in MulleObjC is tricky. It is best if you can restrict yourself to **assign**, **copy** and **retain**.

Property deallocation

> ℹ **Note:** This is a rare case, where MulleObjC is compatible with ARC, but incompatible with MMR.

In Apple ARC, properties are automatically cleared during `-dealloc`. In Apple Manual Retain-Release Mode (MRR) you have to do it yourself during `-dealloc`.

In MulleObjC all properties that reference objects or pointers are cleared during `-finalize` by setting them to **0**.

readonly properties - they have no setter - are not cleared. But in mulle-objc *they are backed by an **ivar***. It is open to discussion if you want to release them in `-dealloc` for compatibility or use `-finalize` to break possible retain cycles.

Here is how to write `-dealloc` for compatiblity with MMR (also see ARC Porting tips (page 36)):

```
#if __has_feature(objc_arc) || defined( __MULLE_OBJC__)
# define PROPERTY_RELEASE( p)
#else
# define PROPERTY_RELEASE( p)  [_p release]
#endif
#if __has_feature(objc_arc)
# define SUPER_DEALLOC()
#else
# define SUPER_DEALLOC( p)  [super dealloc]
#endif

- (void) dealloc
{
    PROPERTY_RELEASE( a)
    PROPERTY_RELEASE( b)
    PROPERTY_RELEASE( c)
    SUPER_DEALLOC()
}
```

Missing Attributes

atomic

Yup it's gone. Use locking or the atomic operations provided by mulle-thread (page 0).

weak

Use **assign** instead.

> ℹ **Note:** Use C containers to manage weak references instead. They won't *magically* cleanup though.

strong

When declaring a property use **copy** or **retain**. You usually use **copy** for `NSValue`, `NSDate`, `NSNumber` and `NSString` arguments, and **retain** for everything else.

nullable

One of the strong points of Objective-C is its gracious handling of `nil` values, which simplifies coding a lot. Remember that messaging `nil` also produces `nil` . With the introduction of `nonnull` `nullable` was also introduced. It is superflous.

You can easily get rid of `nullable` compile errors with:

```
#define nullable
```

> ℹ **Note:** Tedious checks for `nil` means you are optimizing your code for the error case. Use **nonnull** sparingly. If a `nil` parameter has no ill effect, don’t mark the code **nonnull**.

unsafe_unretained

Use **assign** instead.

class

Remove the property. Use `static` variables in your `@implementation` then write and declare `+` accessors for them.

> ℹ **Note:** `class` is likely to make a comeback in a future version.

synthesize

Summary:

NOTHING HERE YET

Synchronized is gone

> **Summary:** Synchronized as a keyword and property is not available

> ℹ **Note:** See Threading Programming Guide for more information about `@synchronized`.

Use mulle-thread for least hassle.

mulle-thread (page 0) is available on all platforms, that run Objective-C.

Use `mulle_thread_mutex_t` to transform

```
- (void) myFunction
{
   @synchronized()
   {
   }
}
```

to

```
static mulle_thread_mutex_t   lock;


+ (void) load
{
   mulle_thread_mutex_init( &lock);
}


- (void) myFunction
{
   mulle_thread_mutex_lock( &lock);
   {

   }
   mulle_thread_mutex_unlock( &lock);
}
```

Use NSLock instead

```
static NSLock   lock;


+ (void) initialize
{
   if( ! lock)
      lock = [[NSLock alloc] init];
}

+ (void) deinitialize
{
   if( lock)
   {
      [lock release];
      lock = nil;
   }
}


- (void) myFunction
{
   [lock lock];
   {

   }
   [lock unlock];
}
```

Good points

- code works in all runtimes without another dependency
- `+deinitialize` will not be called by other runtimes, it's a harmless addition

Bad points

- A **NSLock** is slower than a `mulle_thread_mutex_t`

- The lock has not become a proper MulleObjC root object, so this code will leak in tests.

You could fix this with deleting `+deinitialize` and rewriting `+initialize` as:

```
+ (void) initialize
{
   if( ! lock)
   {
      lock = [[NSLock alloc] init];
#ifdef __MULLE_OBJC__
      [lock _becomeRootObject];
      [lock release;]
#endif
   }
}

// + (void) deinitialize clashes with  _becomeRootObject and mu
st be removed
```

Porting Variable Arguments in Methods

Summary: Variable Arguments in MulleObjC

Intro

Variable arguments in methods follow the Mulle MetaABI and are incompatible with `va_list`. C functions continue to use `va_list` though. So MulleObjC will support both formats.

A typical variable argument method

va_list

This is the `+[NSString stringWithFormat:]` method as presumably coded in the Apple Foundation. Conventionally the `va_list` parameter in Apple Foundation methods is called “arguments:”:

```
+ (instancetype) stringWithFormat:(NSString *) format, ...
{
   NSString   *s;
   va_list    args;

   va_start( args, format);
   s = [self stringWithFormat:format
                    arguments:args];
   va_end( args);
   return( s);
}
```

mulle_vararg_list

In MulleObjC the type is `mulle_vararg_list`. And if it is used as a parameter its called “mulleVarargList:” by convention. `va_list` which is still a possibility type due to C code (e.g. `NSLog`), is called `varargList:` instead for discrimination.

This is how `+[NSString stringWithFormat:]` is actually coded in MulleFoundation:

```
+ (instancetype) stringWithFormat:(NSString *) format, ...
{
   NSString             *s;
   mulle_vararg_list    args;

   mulle_vararg_start( args, format);
   s = [self stringWithFormat:format
              mulleVarargList:args];
   mulle_vararg_end( args);
   return( s);
}
```

So that’s pretty similar.

MulleObjC supports both

It’s not an either or scenarion, as MulleObjC supports both:

```
+ (instancetype) stringWithFormat:(NSString *) format
                  mulleVarargList:(mulle_vararg_list) arguments
{
   return( [[[self alloc] initWithFormat:format
                         mulleVarargList:arguments] autoreleas
e]);
}


+ (instancetype) stringWithFormat:(NSString *) format
                       varargList:(va_list) args
{
   return( [[[self alloc] initWithFormat:format
                                varargList:args] autoreleas
e]);
}
```

Accessing variable arguments

The actual access of variable arguments of `mulle_vararg_list` is very different though.

See objc-compat for some details on how to achieve this portably.

nszone

Summary:

Zones are dead. Do not use the `withZone:` methods anymore.

MulleObjC will work well enough if you use them, but they are just superflous. Incidentally I don’t think Apple Objective-C uses zones either anymore.

Compiler transforms -zone calls

With that being said, the **mulle-objc** compiler will transform any call to `-zone` into `NULL`.

About the theme's author

Summary: I have used this theme for projects that I've worked on as a professional technical writer.

“De Re MulleObjC” is based on tomjoht/documentation-theme-jekyll . This is the short bio of the author.

Tom Johnson

My name is Tom Johnson, and I’m a technical writer, blogger, and podcaster based in San Jose, California. For more details, see my technical writing blog and my course on API documentation . See my blog’s about page for more details about me.

I have used this theme and variations of it for various documentation projects. This theme has undergone several major iterations, and now it’s fairly stable and full of all the features that I need. You are welcome to use it for your documentation projects for free.

I think this theme does pretty much everything that you can do with something like OxygenXML, but without the constraints of structured authoring. Everything is completely open and changeable, so if you start tinkering around with the theme’s files, you can break things. But it’s completely empowering as well!

With a completely open architecture and code base, you can modify the code to make it do exactly what you want, without having to jump through all kinds of confusing or proprietary code.

If there’s a feature you need but it isn’t available here, let me know and I might add it. Alternatively, if you fork the theme, I would love to see your modifications and enhancements. Thanks for using Jekyll.