RFC for a New Init Mechanism

(Another Way to Initialize Objects in Objective C)

Writing one's own initialization methods in an Objective-C program while respecting the Apple – originally NeXT – official writing rules can be a tedious task.

This article aims at offering another, simpler and more generic approach, inspired from scripting languages like Python or Ruby.

Let's start with a short reminder of the current and historical systems. For initialization methods to be added to one of our classes, three rules should be complied with:

- Redefining the designated initialization method of the superclass is mandatory only if the class defines a new one (with a different prototype).
- If a variant of an initialization method differs from the original only by the number of expected parameters (and not by their nature), then the method that accepts a 'low' number of parameters has to call one that accepts just a little more.
  (Implication: all initialization methods have to call directly or indirectly the designated initialization method of their own class.)
- The designated initialization method of a class must call the designated initialization method of its superclass.

For those familiar with initialization issues, remember how boring it can be to have to choose between two exclusive methods which can both apply to becoming the designated initialization method? Example:

- `(id) initProbeName: (NSString) *aName temperature InDegreesFahrenheit: (double) aTemperature`
- `(id) initProbeName: (NSString) *aName temperature InDegreesCelsius: (double) aTemperature`

Now imagine there is a way to write your initialization method only once, then use it in all your subclass hierarchies.

The principle is simple and understated: it consists in providing one unique method to perform all initializations, whatever the properties you want to define.

Furthermore, you don't have to create new initialization methods, again and again, every time you add a new class.

Implementation

First idea:

```
MyObject (subclass of NSObject)
```

Intuitively, the first idea that comes to the mind of an object-oriented developer who wants to add behaviors to a class is to 'subclass'.

In this particular case, we would have to subclass NSObject. That would probably work but that method is both inelegant and 'heavy':

- we are inserting a new class into an already complex hierarchy, and we must keep in mind from now on that we must prefer our own root class (rather than the official one, namely NSObject). Much ado about a tiny method…
- ii order to remain consistent, we would have to edit some of our parent class declarations for projects we want to maintain and in which we want to add our new mechanism.

This issue is conveniently solved by the concept of Category. Categories play at least three roles, the most prominent of which – and that is the heart of the issue at hand – is to enable developers to expand the list of a class' methods (no new properties) without requiring that the developer own the source code.

So this is the choice we retained.

Better integration, NSObject+MyInit (category):

For "not so modern" Objective C: Listing 1 and listing 2. Only for "modern" Objective C: Listing 3 and Listing 4.

Few lines but a compendium of concepts (some recents and other more historical):

- Category (to augment a class whose, in this case, we do not have the source code) – simply with the syntax NewClassName (CategoryName),
- fast iteration – with for / in construction (I would prefer @for / in to show it's an Objective C exclusiveness),
- KVC (Key-Value Coding) – by using here the "setValue:forKey:" method,
- "block" – availability of blocks is not really "modern" but it is always interesting to show usage of this feature,
- exceptions handling – with @try / @catch construction and a NSException instance, => a lot of controversy about NSException use on iOS (discussion in *http://stackoverflow.com/questions/4310560/usage-of-nsexception-in-iphone-apps* with a link on a full article: *http://club15cc.com/code/objective-c/dispelling-nsexception-myths-in-ios-can-we-use-try-catch-finally*)
- modern dictionary manipulation syntaxes (incidentally of NSArray lists): initialization (container literals), access (subscripting),
- pseudo type instancetype (which is an efficient alternative of id for this kind of methods),
- more below…

Listing 1. *file NSObject+InitWithDictionary.h*

```
@interface NSObject (InitWithDictionary)

- (id) initWithDictionary: (NSDictionary *) aDict;

@end
```

Listing 2. *file NSObject+InitWithDictionary.m*

```
#import "NSObject+InitWithDictionary.h"

@implementation NSObject (InitWithDictionary)

- (id) initWithDictionary: (NSDictionary *) aDict
{
    if (self = [self init])
    {
        [aDict enumerateKeysAndObjectsUsingBlock: ^(id
                   iVarName, id initValue, BOOL *stop)
        {
            @try
            {
                [self setValue: initValue forKey:
                   iVarName];
            }
            @catch (NSException *exception)
            {
                NSLog(@"initWithDictionary assign
                   error (%@) value %@", [exception
                   reason], initValue);
            }
        } ];
    }
    return self;
}

@end
```

Listing 3. *file NSObject+InitWithDictionary.h*

```
@interface NSObject (InitWithDictionary)

- (instancetype) initWithDictionary: (NSDictionary *) aDict;

@end
```

Listing 4. *file NSObject+InitWithDictionary.m*

```
#import "NSObject+InitWithDictionary.h"

@implementation NSObject (InitWithDictionary)

- (instancetype) initWithDictionary: (NSDictionary *) aDict
{
    if (self = [self init])
    {
        for (NSString *iVarName in aDict)
        {
            @try
            {
                [self setValue: aDict[iVarName]
                   forKey: iVarName];
            }
            @catch (NSException *exception)
            {
                NSLog(@"initWithDictionary assign
                   error (%@) value %@", [exception
                   reason], aDict[iVarName]);
            }
        }
    }
    return self;
}

@end
```

I detect the begining of a controversy. With such a mechanism the user (the consumer) will reap the error messages (resulting from the exceptions).

That is why it is important to use wisely the NSString constants (which carry our property names). This will never be a panacea, but prevent a significant proportion of inevitable typographical errors.

Question: how-to assign default values to properties during object instanciation?

Probably 2 ways. I don't know at that time which one is the best (if there is one). Start redefining initWithDictionary: which begins to send message [super initWithDictionary: aDict] then continue giving initial values to properties.

The other way you have to assign initial values for your objects is to redefine your own init method which – of course – starts with the message [super init]. It's possible simply because the main method, initWithDictionary:, send message to self.

In fact I thinks both solutions are not really equivalent and meet two different needs. (Up to you to discover which one.)

And if it really is unavoidable, I do not see any cons-indication to mix the two principles (historical initialization

Listing 5. *file Person.h*

```
#import "NSObject+InitWithDictionary.h"

@interface Person : NSObject

@property (copy, nonatomic) NSString *lastname;
FOUNDATION_EXPORT NSString * const PROPNAME_LASTNAME;
@property (copy, nonatomic) NSString *firstname;
FOUNDATION_EXPORT NSString * const PROPNAME_FIRSTNAME;
@property (assign, nonatomic) char gender; //
                    'F'emale, 'M'ale, 'U'nknown
FOUNDATION_EXPORT NSString * const PROPNAME_GENDER;
@property (assign, nonatomic) int age;
FOUNDATION_EXPORT NSString * const PROPNAME_AGE;

- (id) init;
- (void) setAge: (int) newAge;

@end
```

Listing 6. *file Person.m*

```
#import "Person.h"

@implementation Person

@synthesize lastname = _lastname, firstname = _firstname,
                    gender = _gender, age = _age;

NSString * const PROPNAME_LASTNAME = @"lastname";
NSString * const PROPNAME_FIRSTNAME = @"firstname";
NSString * const PROPNAME_GENDER = @"gender";
NSString * const PROPNAME_AGE = @"age";

- (id) init
{
    if (self = [super init])
    {
        self.gender = 'U';
        self.age = -1;   // -1 for unknown age
    }
    return self;
}

- (void) setAge: (int) newAge
{
    if ((newAge == -1) || ((newAge >= 0) && (newAge <
                150)))
    {
        _age = newAge;
    }
    else
    {
        NSLog(@"Age off limit (%i)", newAge);
    }
}

@end
```

Listing 7. *file main.m*

```
#import "Person.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        Person *yannick = [[Person alloc]
                    initWithDictionary: [NSDictionary
                    dictionaryWithObjectsAndKeys:
                    @"Yannick", PROPNAME_FIRSTNAME,
                    @"Cadin", PROPNAME_LASTNAME,
                    [NSNumber numberWithInt: 45],
                    PROPNAME_AGE, nil]];

        NSLog(@"Welcome to %@ %@ who is %d years
                    old", [yannick firstname], [yannick
                    lastname], [yannick age]);
    }
    return 0;
}
```

method prototype and call to the method described here). Such a situation could arise in the context of a Protocol.

If it requires the definition of a method, for example `initProbeName: tempatureInDegreesCelsius`, then why not write:

```
- (instancetype) initProbeName: (NSString)
*aName temperatureInDegreesCelsius: (double) aTemperature
{
... optional checking (like if (aTemperature > 1000) ...)
return [self initWithDictionary: @{ @"name" : aName,
                    @"temperature" : @(aTemperature) }];
}
```

The syntax @(count) is a good example of what the last release of Objective C admits, it's called boxing.

Usage

For "classic" Objective C: Listing 5-7. The idea offered here became even more interesting with the recent

Listing 8. *file Person.h*

```
The same as for "classic" Objective C, only replace
(id) init;
by
(instancetype) init;
```

Listing 9. *file Person.m*

```
The same as for "classic" Objective C, only replace
(id) init
by
(instancetype) init
AND… You can remove all the "@synthesize" lines.
```

Listing 10. *file main.m*

```
#import "Person.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        Person *yannick = [[Person alloc]
initWithDictionary: @{ PROPNAME_FIRSTNAME :
@"Yannick", PROPNAME_LASTNAME : @"Cadin", PROPNAME_
                    AGE : @45 } ];

        NSLog(@"Welcome to %@ %@ who is %d years
                    old", yannick.firstname, yannick.
                    lastname, yannick.age);
    }
    return 0;
}
```

changes in Objective-C. In particular literals and container literals which permit a very short wrtiing of many expressions, especially NSDictionary instances. The same for NSArray, NSNumber, boolean constants, etc. *http://clang.llvm.org/docs/ObjectiveCLiterals.html*. For "modern" Objective C: Listing 8-10.

End of concept overviews:

- dot notation (not so recent).
- automatic (implicit) synthesize.
- boxing (in an example above).
- modern syntaxes (literals) of constants writing (NSNumber, booleans, etc).

Depending the release of Objective C you want to write for, you have to use the first syntax or you can prefer the second (for the latest releases). My own opinion (thanks to add yours ;-)

Pros

- simplicity.
- the KVC mechanism allow keeping custom initialization of each variable. It's not a short circuit.

Cons

(performance) slowness (but this needs to be qualified because it probably depends on ratio: class hierarchy depth – given the potentially large number of messages – from the number of values to allocate / initialize) the main reasons for the poor performances: NSExceptions management and use of the KVC mechanism.

For interested readers: *https://github.com/nicklockwood/BaseModel*. For discussion about const in Objective C: *http://stackoverflow.com/questions/538996/constants-in-objective-c*.

YANNICK CADIN

Yannick Cadin is 45 years old, including 27 devoted mainly to computer industry (hardware and software). Programming of UBI-SOFT's very first game as freelance developer. Many salary jobs in small or medium companies: video game publisher, professional software editor, textile CAM software editor, Decision Tools software editor, computer resellers or distributors, consulting companies, value-added resellers (VAR), mapping company and even in the public sector with the title of Chief Operating Officer at the Louvre Museum. Manager of MICRO REPONSE specialized in providing computers running NEXTSTEP / OpenStep. Positions held: most of them, sales engineer, developer, support technician, trainer, ... Freelance writer on a more or less regular basis for a dozen magazines for the last 26 years. Casual proofreader and speaker. Red Hat Linux, Ubuntu, LPI, * BSD and Apple certified. Currently Manager of Diablotin.