Objective-C Compiler Warnings

Adrian Kosmaczewski

2009-07-16

A recent comment by Joe D'Andrea in a previous post reminded me about the importance of removing compiler warnings in Xcode projects. Most importantly, it reminded me of a conversation with a fellow developer a couple of weeks ago, in which he told me that he was surprised to see that my projects compiled all the time without warnings. Not a single one. Nada. And that I took the time to remove them before checking code into source control.

He actually didn't know you could remove all compiler warnings; he thought Objective-C was the land of compiler warnings. This situation, I think, is far from exceptional, and due mostly to cultural and technical reasons.

It is my opinion, that removing compiler warnings is **basic project hygiene**, like writing unit tests, or using the Clang Static Analyzer. I will explain in this post some techniques I use to remove warnings in my Objective-C code.

First of all, why does the Objective-C compiler (or compilers in general) output "warnings"? Many developers are puzzled the first time they encounter them, since even if the compiler complained, the application usually runs anyway without (perceptible) problems.

Warnings are used to signal specific issues in the source code which could potentially lead to crashes or misbehavior under some circumstances, but which should not (pay attention to the verb "should") block the normal compilation and (hopefully) execution of your code (otherwise, it would be a compiler error).

It's the way used by your compiler to say:

> Hey, I'm not sure, but there's something fishy in here.

Not removing warnings, as I said above, is a problem that originates both in the programming background of the developer, and specific technical issues.

Culturally speaking, many other programming environments either do not have compilers at all (at least not "visible" ones, like Ruby or PHP) or simply do not spit warnings for anything else than deprecated methods (like C# or Java); this situation has made many developers new to the iPhone platform to blatantly ignore them.

Technically, given the fact that Objective-C is the "other" object-oriented superset of C, and that it behaves as a coin with both a static and a dynamic side, compiler warnings convey a great amount of precious information that must *never* be ignored.

In this sense, Objective-C has a lot in common with C++. Ignoring warnings in C++ is strongly discouraged, and Scott Meyers explains this in chapter 9 of his book "Effective C++", stating that (third edition, page 263):

> Take compiler warnings seriously, and strive to compile warning-free at the maximum warning level supported by your compilers

In the case of Objective-C, this can be done by setting GCC_TREAT_WARNINGS_AS_ERRORS (-Werror) to true in your build settings.

Steve McConnell takes this advice to another level of importance in his classic book "Code Complete" (second edition, page 557):

> Set your compiler's warning level to the highest, pickiest level possible, and fix the errors it reports. It's sloppy to ignore compiler errors. It's even sloppier to turn off the warnings so that you can't even see them. Children sometimes think that if they close their eyes and can't see you, they've made you go away (…).

Assume that the people who wrote the compiler know a great deal more about your language than you do. If they're warning you about something, it usually means you have an opportunity to learn something new about your language.

To give a concrete example of the importance of warnings, many of us have had to migrate applications developed for iPhone OS 2.x to the 3.0 operating system, mostly because failure to run on the new version of the OS was ground for removal from the App Store. That moment of truth, the rebuild of the Xcode project, unveiled a plethora of compiler warnings, most due to deprecated methods, like the tableView:accessoryTypeForRowWithIndexPath: method of the UITableViewDelegate protocol, or the initWithFrame:reuseIdentifier: method of the UITableViewCell class (which, incidentally, are properly marked as such in the documentation, too).

Compiler warnings in Objective-C have a multitude of reasons:

- Using deprecated symbols;
- Calling method names not declared in included headers;
- Calling methods belonging to implicit protocols;
- Using some ambiguous commands which might be intentional but are syntactically valid anyway;
- Forgetting to return a result in methods not returning "void";
- Forgetting to #import the header file of a class declared as a forward "@class";
- Downcasting values and pointers implicitly.

Many solutions exist for these problems, and I do not claim to know them all; I'll just describe some of them, and hopefully some of the readers of this post will add others in the comments below.

1) Make implicit protocols explicit

This is a really simple one, and it's also good for documentation and code clarity reasons. Get all the references of delegate methods you use in the code, and group the methods into their own header file. Import the header file whenever you need, and make classes explicitly implement them:

```
//...
@interface NewClass : NSObject <SomeProtocol>
{
    //...
```

Of course, take advantage of Objective-C's @required and @optional keywords in your protocol declarations; they are used by the compiler to verify (or not) the existence of delegate methods in your class implementation. This way, you'll surely remove some warnings.

2) Do not use "id" as the data type for delegate fields Using id as datatype for delegate fields

I personally use the following declaration for delegate fields:

```
//...
NSObject<SomeProtocol> *delegate;
//...
```

instead of simply

```
//...
id delegate<SomeProtocol>;
//...
```

This is because I always check on delegates before calling their methods. Call me paranoid, but this is what a good delegate call looks to me:

```
if([delegate respondsToSelector:@selector(someObj:doesThis:)])
{
    [delegate someObj:self doesThis:@"123"];
}
```

Using NSObject instead of id in the delegate declaration avoid yet another warning. **Update, 2009-07-17:** This is because using id raises a warning that "respondsToSelector:" is not defined in SomeProtocol. The solution for this is making SomeProtocol inherit from the NSObject protocol (I always forget this double life of the NSObject symbol):

```
@protocol SomeProtocol <NSObject>
//...
@end
```

This way, you can use id<SomeProtocol> variables without problem.

3) Create categories for private methods

Objective-C uses the @private, @public and @protected identifiers only for instance fields, but otherwise methods can only be marked as instance ("-") or static ("+"), but all methods specified in the header file are public by default.

If you need to specify private methods, do that in your implementation file, creating what's called a "category" of your own class, which basically "extends" the class with new methods:

```
#import "NewClass.h"

@implementation NewClass (Private)
// your methods here
@end

@implementation NewClass
// ...
@end
```

This way you can define private methods, not exposing them in the public interface file. And you remove some more warnings.

4) Turn implicit type conversions and casts into explicit ones:

This one is inspired by McConnell's Code Complete (second edition, page 293):

```
int i;
float y, x;
y = x + (float)i
```

Even if the compiler could work out the "y = x + i" expression without problem, the code above will remove yet another warning, and will make your code more obvious and easier to read, since it clearly states your intentions.

5.1) Support earlier OS versions via runtime checks

If you have to write iPhone applications compatible with both the 3.0 and 2.x versions, this sample from Apple https://developer.apple.com/iphone/library/samplecode/MailComposer/listing7 provides instructions on how to do it.

"MailComposer runs on earlier and later releases of the iPhone OS and uses new APIs introduced in iPhone SDK 3.0. See below for steps that describe how to target earlier 0S versions while building with newly released APIs."

It is worth noting that this sample application compiles without a single warning!

5.2) Support earlier OS versions via #defines

Use #ifdef IPHONE_OS_3.0 and IPHONE_OS_2.2.1 if you can (or must) provide different binaries for each supported platform. This might be the case for in-house applications, but again, it might help removing some warnings too.

6) Use @class in the @interface, #import on the @implementation

Whenever you use a class on a header file, to avoid cross-references, use the @class keyword to reference it, but do not forget to #import its header file in the implementation!

```
@class AnotherClass

@interface SomeClass : NSObject
{
@private
    AnotherClass *field;
    //...
}
```

and then

```
#import "AnotherClass.h"

@implementation SomeClass

- (id)init
{
    if (self = [super init])
    {
        field = [[AnotherClass alloc] init];
    }
    return self;
}

@end
```

The problem is, if you do not #import the file, you get a warning…

The advantage of this technique is not obvious in small projects, but it is strongly recommended anyway; it prevents header files from cross-referencing each other, and it reduces build times in projects anyway. It is the Objective-C analog to the technique of using "class" statements in C++ header files, instead of #include.

For more information: Matt Gallagher also wrote about this issue, and I strongly recommend his article too! Read also this article from Apple about the issue of compiler warnings which covers all the options available on GCC in great detail.

Finally, all of this boils down to the fact that iPhone programming is not as easy as web development, and that iPhone applications, for many reasons, require patience and attention. Removing warnings from your code is just one of many steps to have great iPhone applications, running smoothly without problems.

Update, 2009-07-17: Regarding the "id vs. NSObject" issue, I've used NSObject because you get a warning when calling respondsToSelector: on a variable of type id. Now, after reading all the comments I've gone back to Xcode and found out that you can define SomeProtocol as implementing the NSObject protocol itself, and this solves the problem. Thanks everyone for the heads-up!