

Coding Conventions

OBJECTIVE C - ONIX INSPECTION

Coding Conventions OBJECTIVE C - ONIX INSPECTION

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| No | Date | Content | Author |
| 1 | 30.10.2018 | First version | Thai Nguyen |
|  |  |  |  |

# Approvals

|  |  |  |  |
| --- | --- | --- | --- |
| No | Date | Role | Approved By |
|  |  |  |  |

# **Table of Contents**

OBJECTIVE C - ONIX INSPECTION

[Revision History 1](#_Toc528596491)

[Approvals 1](#_Toc528596492)

[**Table of Contents** 2](#_Toc528596493)

[**Table of Figures** 2](#_Toc528596494)

[Introduction 3](#_Toc528596495)

[Purpose 3](#_Toc528596496)

[Related Documents 3](#_Toc528596497)

[IOS 4](#_Toc528596498)

[Naming Conventions 4](#_Toc528596499)

[Indentation 4](#_Toc528596500)

[Layout 4](#_Toc528596501)

[Exception Handling / Logging 5](#_Toc528596502)

[Comment 6](#_Toc528675086)

# **Table of Figures**

No table of figures entries found.

# Introduction

## Purpose

The purpose of this document is describing about our code conventions

## Related Documents

-

# **OBJECTIVE C - ONIX INSPECTION**

### Naming Conventions

Names should be as descriptive as possible, within reason.

Avoid non-standard abbreviations (including non-standard acronyms and initialisms). Don’t worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. For example:



### Indentation

* Indent using 2 spaces (this conserves space in print and makes line wrapping less likely). Never indent with tabs. Be sure to set this preference in Xcode.
* Method braces and other braces (if/else/switch/while etc.) always open on the same line as the statement but close on a new line.

**Preferred:**

if (user.isHappy) {

//Do something

} else {

//Do something else

}

**Not Preferred:**

if (user.isHappy)

{

//Do something

}

else {

//Do something else

}

* There should be exactly one blank line between methods to aid in visual clarity and organization. Whitespace within methods should separate functionality, but often there should probably be new methods.
* Prefer using auto-synthesis. But if necessary, @synthesize and @dynamic should each be declared on new lines in the implementation.
* Colon-aligning method invocation should often be avoided. There are cases where a method signature may have >= 3 colons and colon-aligning makes the code more readable. Please do **NOT** however colon align methods containing blocks because Xcode's indenting makes it illegible.

**Preferred:**

// blocks are easily readable

[UIView animateWithDuration:1.0 animations:^{

// something

} completion:^(BOOL finished) {

// something

}];

**Not Preferred:**

// colon-aligning makes the block indentation hard to read

[UIView animateWithDuration:1.0

animations:^{

// something

}

completion:^(BOOL finished) {

// something

}];

### Layout

Init methods should follow the convention provided by Apple's generated code template. A return type of 'instancetype' should also be used instead of 'id'.

- (instancetype)init {

self = [super init];

if (self) {

// ...

}

return self;

}

We require braces around the statements for a conditional.

if (condition) {

body();

}

### Exception Handling / Logging

Format exceptions with @catch and @finally labels on the same line as the preceding }. Add a space between the @ label and the opening brace ({), as well as between the @catch and the caught object declaration. If you must use Objective-C exceptions, format them as follows. However, see Avoid Throwing Exceptions for reasons why you should not be using exceptions.

// GOOD:



Avoid Throwing Exceptions

Don’t @throw Objective-C exceptions, but you should be prepared to catch them from third-party or OS calls.

We do compile with -fobjc-exceptions (mainly so we get @synchronized), but we don’t @throw. Use of @try, @catch, and @finally are allowed when required to properly use 3rd party code or libraries. If you do use them, please document exactly which methods you expect to throw.

Nil Checks

Avoid nil pointer checks that exist only to prevent sending messages to nil. Sending a message to nil reliably returns nil as a pointer, zero as an integer or floating-point value, structs initialized to 0, and \_Complex values equal to {0, 0}.

// AVOID:

if (dataSource) { // AVOID.

[dataSource moveItemAtIndex:1 toIndex:0];

}

// GOOD:

[dataSource moveItemAtIndex:1 toIndex:0];

Note that this applies to nil as a message target, not as a parameter value. Individual methods may or may not safely handle nil parameter values.

Note too that this is distinct from checking C/C++ pointers and block pointers against NULL, which the runtime does not handle and will cause your application to crash. You still need to make sure you do not dereference a NULL pointer.

### Comment

**File Comments**

A file may optionally start with a description of its contents. Every file may contain the following items, in order:

License boilerplate if necessary. Choose the appropriate boilerplate for the license used by the project.

A basic description of the contents of the file if necessary.

If you make significant changes to a file with an author line, consider deleting the author line since revision history already provides a more detailed and accurate record of authorship.

**Declaration Comments**

Every non-trivial interface, public and private, should have an accompanying comment describing its purpose and how it fits into the larger picture.

Comments should be used to document classes, properties, ivars, functions, categories, protocol declarations, and enums.



Doxygen-style comments are encouraged for interfaces as they are parsed by Xcode to display formatted documentation. There is a wide variety of Doxygen commands; use them consistently within a project.

If you have already described an interface in detail in the comments at the top of your file, feel free to simply state, “See comment at top of file for a complete description”, but be sure to have some sort of comment.

Additionally, each method should have a comment explaining its function, arguments, return value, thread or queue assumptions, and any side effects. Documentation comments should be in the header for public methods, or immediately preceding the method for non-trivial private methods.

Use descriptive form (“Opens the file”) rather than imperative form (“Open the file”) for method and function comments. The comment describes the function; it does not tell the function what to do.

Document the thread usage assumptions the class, properties, or methods make, if any. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

Any sentinel values for properties and ivars, such as NULL or -1, should be documented in comments.

Declaration comments explain how a method or function is used. Comments explaining how a method or function is implemented should be with the implementation rather than with the declaration.

Implementation Comments

Provide comments explaining tricky, subtle, or complicated sections of code.



When useful, also provide comments about implementation approaches that were considered or abandoned.

End-of-line comments should be separated from the code by at least 2 spaces. If you have several comments on subsequent lines, it can often be more readable to line them up.



…