Objective-C Style Guide

Why Do We Need A Style Guide?

Any team of developers should be working from a single playbook when it comes to the code style used. This makes the code more maintainable and easier to consume.

Often our customers will have access to our source code. Ideally an entire project will appear as if it's been written by a single person. Or at least reviewed and edited to be consistent throughout.

The original version of this document was based on (stolen from) bits of these documents:

- Apple's Coding Guidelines for Cocoa (http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html)
- WebKit Style Guide (http://www.webkit.org/coding/coding-style.html)
- Google Objective-C Style Guide (http://google-styleguide.googlecode.com/svn/trunk/objcguide.xml)
- Zarra Studios Coding Style Guide (http://www.cimgf.com/zds-code-style-guide/)

It has since been updated through discussions and meetings with other developers using these guidelines.

Suggestions

Generally speaking, we want to format our code the way Apple does. This will make it easier for other iOS programmers to read. It will also be more consistent than just making up your own style as you go along. If there's confusion about coding style, find an example of similar code from Apple. Preferably from their documentation, not from a sample project (many of which don't follow Apple's standards). This is just a guideline. Apple sometimes contradicts itself.

Whenever you work in a source file, leave it cleaner than when you found it. If you find code that violates this guide, correct it. If the code is outdated then update it.

When there's a choice of formatting styles, follow the convention already used in a given source file.

Use the static analyzer. It does some very useful style checking.

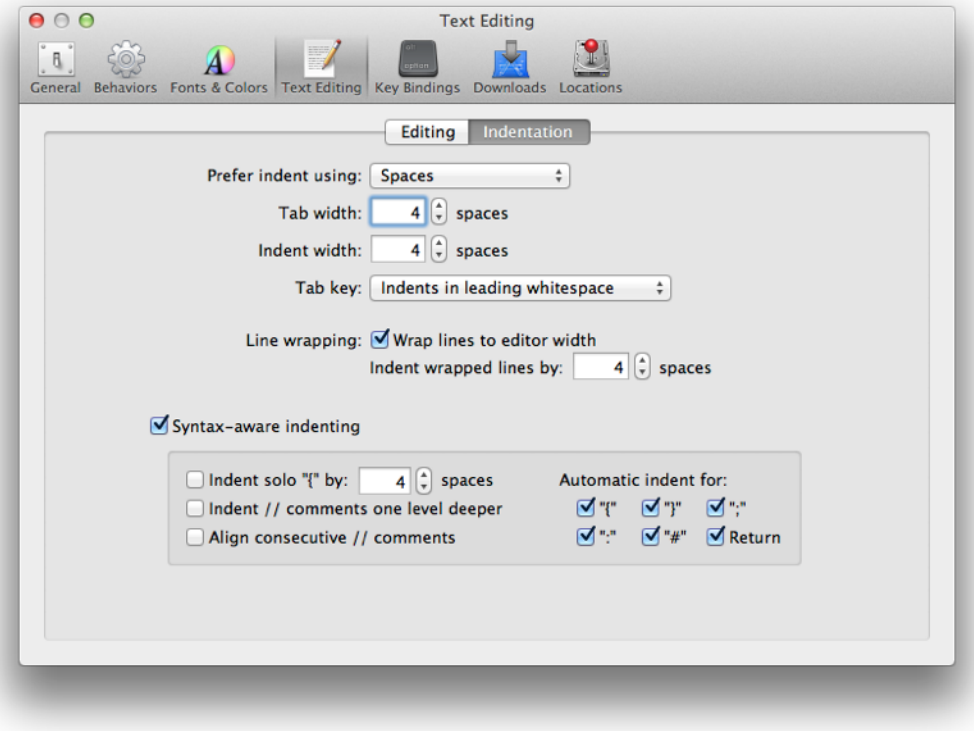
Ask somebody to review your code. As soon as you're aware somebody is going to critique your code, you'll start writing better code.

These conventions are not intended to be applied to C, C++, or Swift code within Objective-C projects.

Indenting

Use spaces, not tabs, to indent code. The indent size is 4 spaces. Code editors should be configured to convert tabs that you type to 4 spaces.

Specifically, we'd like you to configure Xcode to handle indentation as shown here (from Xcode 4.4):



If you want to wrap a long line of code then make sure it is colon-aligned per Xcode's automatic formatting.

Blank Lines

To visually separate the elements of the .m file, use two blank lines:

- Following the #imports
- Between methods
- Around #pragma marks
- Above and below @interface..@end

There should never be more than two consecutive blank lines in any Objective-C source file.

Use single blank lines generously throughout methods to visually group lines of code. Do not use two consecutive blank lines in methods.

Never place a blank line before the first line of code following the opening bracket, or before any closing bracket.

Class Names

All projects should adopt a three-letter prefix. We avoid two-letter prefixes to minimize conflicts with third-party frameworks and future Apple frameworks. All classes specific to the project should be named with this prefix.

File Header Comments

Each .h and .m file should begin with comments containing the file name and copyright information similar to this:

```
//
//  XXXNavigationController.h
//
//  Copyright 2014 ABC Company, Inc. All rights reserved.
//
```

This is a modified version of the default comments added by Xcode. Notice the project name and creator/creation date have been removed.

Pragma Marks

Separate methods in the class implementation into groups using these #pragma marks:

```
#pragma mark - Class Methods
#pragma mark - Setup & Teardown
#pragma mark - (superclass name)
#pragma mark - Public
#pragma mark - Private
#pragma mark - Actions
#pragma mark - Notifications
#pragma mark - (delegate protocol name)
#pragma mark - Accessors
```

Do not include a `#pragma mark` if there are no methods in that section.

Generally these groups should appear in the order listed above. Additional groups may be inserted or appended where appropriate.

Be sure to include the dash so a divider line is added to the drop-down menu in Xcode. Subcategories of major groups may be created without the dash.

```
#pragma mark - Private
  ...
#pragma mark Video Handling
  ...
#pragma mark In App Purchase
```

Place two blank lines above and below each `#pragma mark`.

Do not use the older-style two-line `#pragma marks`.

RIGHT:

```
#pragma mark - Setup & Teardown
```

WRONG:

```
- #pragma mark -
- #pragma mark Setup & Teardown
```

Many of the users of this guide have created code snippets in Xcode to simplify this process. Copy and paste the code below to create your own code snippets.

Title: All Class Pragma Marks
Completion Shortcut: psec
Platform: iOS
Language: Objective-C
Completion Scopes: Class Implementation

```
#pragma mark - <#SectionName#>
```

Title: Pragma Mark Section
Completion Shortcut: psecall
Platform: iOS
Language: Objective-C
Completion Scopes: Class Implementation

```
#pragma mark - Class Methods

#pragma mark - Setup & Teardown

#pragma mark - (superclass name)

#pragma mark - Public

#pragma mark - Private

#pragma mark - Notifications

#pragma mark - (delegate protocol name)

#pragma mark - Actions

#pragma mark - Accessors
```

Method Names

Method names should follow the conventions defined by the Cocoa libraries. They should almost never contain abbreviated words.

RIGHT:

```
- (void)beginRegistrationWithNavigationController:(UINavigationController *)controller
```

WRONG:

```
- (void)beginRegWithNavCon:(UINavigationController *)controller
```

Be consistent. Similar methods that do the same thing in two different classes should probably have the same name.

EXAMPLE:

```
- (void)handleError:(ABCHTTPURLConnection *)connection withError:(NSError *)error
```

Note that the first parameter is a connection, not an error. So better:

```
- (void)handleConnection:(ABCHTTPURLConnection *)connection withError:(NSError *)error
```

Apple generally doesn't use "with" when returning an error value. Better still:

```
- (void)handleConnection:(ABCHTTPURLConnection *)connection error:(NSError *)error
```

This is a delegate method. Normally these contain "will" or "did", and rarely start with "handle":

```
- (void)connection:(ABCHTTPURLConnection *)connection didFailWithError:(NSError *)error
```

Turns out this is exactly how Apple names this method in the NSURLConnection class. So anyone that's familiar with Cocoa should understand what this method does.

Method names should never begin with the underscore character. This format is reserved for use by Apple.

Method Signatures

Method signatures should have a space after the scope (the plus or the minus sign). There should be a space between the parameter type and pointer (asterisk) character. There should be a space between the method segments. Other than that there should be no spaces in the method signature.

RIGHT:

```
- (void)setExample:(NSString *)example animated:(BOOL)animated;
```

WRONG:

```
- (void) setExample:(NSString *)example animated:(BOOL)animated;
-(void)setExample:(NSString *)example animated:(BOOL)animated;
- (void)setExample:(NSString *) example animated:(BOOL) animated;
```

Method signatures in the class implementation should never be followed by a semicolon.

Curly Brackets

A method's opening bracket should be on the following line and flush left. Code should be indented 4 spaces from the opening bracket. This is the only place where the opening bracket is placed at the beginning of a new line. In all other cases, the opening bracket is placed inline.

In all cases, the closing bracket is always on a new line by itself.

RIGHT:

```
- (NSString *)foo
{
    return @"foo";
}
```

WRONG:

```
- (NSString *)foo {
    return @"foo";
}
```

Code within an `if` or `for` statement should aways be enclosed in brackets, even if not required by the compiler.

RIGHT:

```
if (timeToGetCoffee) {
    [self buyCoffee];
    [self chugIt];
}
else {
    self.outtaHere = YES;
}
```

WRONG:

```
if (timeToGetCoffee)
{
    [self buyCoffee];
    [self chugIt];
}
else
    self.outtaHere = YES;
```

WRONG:

```
if (timeToGetCoffee)
{
    [self buyCoffee];
    [self chugIt];
} else {
    self.outtaHere = YES;
}
```

Switch tests:

RIGHT:

```
switch (input)
    case 1:
        [self buyCoffee];
        [self chugIt];
        break;

    case 2: {
        BOOL out = YES;
        self.outtaHere = out;
        break;
    }

    default:
        break;
}
```

Parentheses

Keywords such as `if`, `while`, `do`, `switch`, etc. should have a space after them. There should be no space following the opening parenthesis or before the closing parenthesis.

RIGHT:

```
if (timeToGetCoffee) {
```

WRONG:

```
if(timeToGetCoffee){
if ( timeToGetCoffee ) {
```

Spaces are also used in the `@property` declarations.

RIGHT:

```
@property (strong, nonatomic) UIViewController *viewController;
@property (strong, nonatomic, getter = isLoading) BOOL loading;
```

WRONG:

```
@property(nonatomic,strong) UIViewController *viewController;
@property(strong,nonatomic,getter=isLoading) BOOL loading;
```

C functions should not use a space before the opening parenthesis. However, spaces should be used between parameters.

RIGHT:

```
CGRect frame = CGRectMake(10.0f, 20.0f, 15.0f, 15.0f);
```

WRONG:

```
CGRect frame = CGRectMake (10.0f, 20.0f, 15.0f, 15.0f);
CGRect frame = CGRectMake(10.0f,20.0f,15.0f,15.0f);
```

Dot Notation

Use dot notation to get and set properties. Do not use dot notation for functions.

RIGHT:

```
UIView *view = self.view;
NSInteger total = [objectsArray count];
```

WRONG:

```
UIView *view = [self view];
NSInteger total = objectsArray.count;
```

Variables

Use `NSInteger` instead of `int`, `BOOL` instead of `Boolean`, etc.

Use `CGFloat` instead of `float` or `double` when working with views, drawing, or anywhere that Apple specifies a `CGFloat` value. For clarity, hard-code values to `CGFloat` using one decimal place and the float notation. Do not omit the leading zero in values less than one or the trailing zero for fractal values.

RIGHT:

```
CGFloat top = 23.0f;
```

WRONG:

```
CGFloat top = 23;
CGFloat top = 23f;
CGFloat top = 23.f;
CGFloat top = .0f;
CGFloat top = 0f;
```

Objective-C uses `YES` and `NO` for Boolean variables. Therefore, the use of `True` and `False` are incorrect in Objective-C code.

Don't directly access structure or object elements if getters & setters are available, except in `-init` and `-dealloc` methods, or when overriding accessor methods.

RIGHT:

```
NSString *string = self.title;
CGPoint point = CGPointMake(10.0f, 20.0f);
```

WRONG:

```
NSString *string = _title;
CGPoint point = {10.0f, 20.0f};
```

Variable/Property/Parameter Names

Instance names should be descriptive of what they represent. Single letter variable names should never be used, even for incrementers. If it's an index, name it "index."

RIGHT:

```
- (void)somethingDidHappen:(NSNotification *)notification
```

WRONG:

```
- (void)somethingDidHappen:(NSNotification *)notif
```

Always use consistent names. A property that does the same thing in two different classes shouldn't be called `postDictionary` in one class and `updateDictionary` in another class. Use the same variable or property names used in the Cocoa libraries when appropriate.

RIGHT:

```
@property (strong, nonatomic) IBOutlet UITableView *tableView;
```

WRONG:

```
@property (strong, nonatomic) IBOutlet UITableView *gamesTableView;
```

In general, you shouldn't abbreviate names. However, some abbreviations are either well established or have been used in the past, and so you may continue to use them. See Apple's Coding Guidelines for Cocoa document for a list of acceptable abbreviations.

Never use underscores within variable, property, or parameter names.

Never, ever prefix any variable with "my."

Automatic Reference Counting

We use automatic reference counting (ARC) now. Existing projects that do not use ARC should be updated to use ARC.

Third-party source code and frameworks that do not use ARC may continue to be used without modification.

Properties

Property definitions should be used rather than defining instance variables. Let the compiler create the needed instance variables automatically. This will help to insure proper memory management. Direct ivar access should be avoided except in initialization and `-dealloc` methods, where it should always be used instead of using the property.

Even for simple variables like Booleans (BOOL) or integers (NSInteger), use properties for consistency, not instance variables.

Xcode no longer requires `@synthesize` statements. Use them only if you require a unique instance variable name. When using `@synthesize` and `@dynamic`, place each one on a single line.

RIGHT:

```
@synthesize parentViewController = _readWriteParentViewController;
@dynamic nameLabel;
```

WRONG:

```
@synthesize fullName = _fullName;
@synthesize nameLabel;
@dynamic nameLabel, addressLabel;
```

Except in `-init` and `-dealloc` methods, always use accessor methods, whether hard-coded (`@dynamic`) or synthesized, when setting and getting class instance variables.

RIGHT:

```
self.aProperty = something;
```

WRONG:

```
_anInstanceVariable = something;
```

Properties that conform to NSCopying should generally copy rather than retain the value.

RIGHT:

```
@property (copy, nonatomic) NSString *name;
```

WRONG:

```
@property (strong, nonatomic) NSString *name;
```

IBActions

Do not use the `(id)sender` parameter in an IBAction method signature unless it's needed. If it is needed, you can specify the parameter type (e.g. UIButton) rather than using `id`. Use `id` only when creating very generic actions where you don't know in advance the type of control that will trigger it, but still need a reference to it.

RIGHT:

```
- (IBAction)beginUpdate;
- (IBAction)beginUpdate:(UIButton *)button;
```

WRONG (mostly):

```
- (IBAction)beginUpdate:(id)sender;
```

Comparisons

Compare objects to `nil`, rather than just testing the object directly. This makes it obvious that the property is an object, as opposed to a Boolean value.

RIGHT:

```
if (anObject == nil) {
    ....
}
```

WRONG:

```
if (anObject) {
    ....
}
```

When testing the existence of an object, `nil` belongs to the right of the == operator.

RIGHT:

```
if (anObject == nil) {
    ....
}
```

WRONG:

```
if (nil == anObject) {
    ....
}
```

- dealloc

Now that we're using ARC, we're no setting the object's instance variables to nil in the `-dealloc` method. In most cases the method should be removed.

Block Indentation

Either of these forms of block indentation is acceptable:

RIGHT:

```
[UIView animateWithDuration:0.3f animations:^{
     containerView.alpha = 1.0f;
}];

[UIView animateWithDuration:0.3f
                 animations:^{
                     containerView.alpha = 1.0f;
                 }];
```

Empty Methods

Any method that is empty or just calls super should be removed, unless it is required by the compiler.

Unnecessary/unused code from Xcode templates should also be completely removed from source files.

Operators

Use spaces around logical and arithmetic operators. However, no space should be used before the ++ and -- operators.

RIGHT:

```
frame.origin.x = truncf((contentRect.size.width - frame.size.width) / 2);
count++;
```

WRONG:

```
frame.origin.x=truncf((contentRect.size.width-frame.size.width) /2);
count ++;
```

Constants

Follow the Constants section in Apple's Introduction to Coding Guidelines for Cocoa when naming constants. Specifically:

Constants should begin with the three-letter project prefix.

Do not begin constants with the letter "k" in Objective-C code (that's a C thing).

Do not use all uppercase letters (that's for creating preprocessor constants).

Do not use `#define` to create constants.

Integers, floats, and Boolean constants should look something like this:

```
const CGFloat XXXClassNameBackgroundAlpha = 0.5f;
```

String constants should look like this:

```
NSString * const XXXClassNameCompanyNameKey = @"CompanyName";
```

App Delegate

The application delegate should be named XXXAppDelegate, where XXX is the project's three-letter prefix. It should not be named Project_AppDelegate or anything else that Xcode might suggest.

Resource Files

Resource (.xib) files shared by multiple classes should be saved in the project's Resource folder and placed in the Resource folder (preferably in a subfolder for better organization) in Xcode's Project Navigator.

Resource files used by a single class should be placed alongside the .h and .m files in Xcode's Project Navigator. The actual file should be saved in the Classes folder with the associated class files. The resource file name should usually end in "view" "menu", "window" or "cell". View controller xib files should not end with the word "controller."

Resource file names (images, xibs, etc.) should begin with the project's three-letter project code. Resources associated with a specific class should begin with the class name.

Selected or highlighted image variants should have the same file name as the normal image, with a "-selected" extension. Generally we just use "-selected" for both selected and highlighted image variations, unless both selected and highlighted variants are needed.

If multiple sizes of an image are needed, append generic adjectives such as "large" or "small" (or other names recommended by Apple), rather than specific ones such as "login" or "accounts." This allows the images to be more easily used in other areas of the application.

If an image or xib file is specific to iPhone or iPad, use the "~iphone" or "~ipad" filename extension (even if the project is currently designed for only one platform). Omit this extension only if the resource is intended to be used on both platforms.

Importing Headers

Framework imports should be in the Prefix.pch file and not in the individual class or header files. Any existing framework imports in the class implementations or headers should be removed. This includes Foundation.h and UIKit.h imports.

Comments

Comments should be rare. Code is easier to read than comments. Comments go stale, code doesn't. If the code is self-explanatory (as it should be when descriptive variable and method names are used) then comments are not necessary.

However, comment code that isn't self-explanatory, or that deviates from what another coder might expect to see.

RIGHT:

```
// We are doing this in a strange way because the normal solution caused the monster's
// eyes to turn green. Therefore this solution is the correct one.
```

WRONG:

```
// Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
```

Do not comment closing parenthesis.

WRONG:

```
if (something) {
    ...
}  // if
```

Code that is no longer needed should be completely removed from the source files. Do not simply comment it out. If we need to restore it, we can recover it from GitHub.

Example

The following pages show examples of properly formatted .h and .m files.

```
//
//  XXXIconButton.h
//
//  Copyright 2014 ABC Company, Inc. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface XXXIconButton : UIButton

@end
```

```
//
//  XXXIconButton.m
//
//  Copyright 2014 ABC Company, Inc. All rights reserved.
//

#import "XXXIconButton.h"


static CGFloat const XXXIconButtonPadding = -2.0f;  // Padding between the image and text frames.


@implementation XXXIconButton


#pragma mark - Setup & Teardown


- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self iconButtonInit];
    }
    return self;
}


- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    if (self) {
        [self iconButtonInit];
    }
    return self;
}


- (void)iconButtonInit
{
    self.titleLabel.textAlignment = NSTextAlignmentCenter;
}


#pragma mark - UIButton


- (CGRect)imageRectForContentRect:(CGRect)contentRect
{
    CGRect imageRect = [super imageRectForContentRect:contentRect];
    CGSize titleSize = [self.titleLabel.text sizeWithFont:self.titleLabel.font];

    imageRect.origin.x = truncf((contentRect.size.width - imageRect.size.width) / 2);
    imageRect.origin.y = truncf((contentRect.size.height - imageRect.size.height -
        titleSize.height - XXXIconButtonPadding) / 2);

    return imageRect;
}


- (CGRect)titleRectForContentRect:(CGRect)contentRect
{
    CGRect imageRect = [super imageRectForContentRect:contentRect];
    CGRect titleRect = [super titleRectForContentRect:contentRect];

    titleRect.origin.x = 0.0f;
    titleRect.origin.y = truncf((contentRect.size.height - imageRect.size.height -
        titleRect.size.height - XXXIconButtonPadding) / 2) + imageRect.size.height +
        XXXIconButtonPadding;
    titleRect.size.width = self.bounds.size.width;

    return titleRect;
}


@end
```