
[DEPRECATED]

NYTimes Objective-C Style Guide is deprecated. No more development will be taking place. Thanks for all your support!

NYTimes Objective-C Style Guide

This style guide outlines the coding conventions of the iOS teams at The New York Times. We welcome your feedback in issues and pull requests. Also, we're hiring.

Thanks to all of our contributors.

Introduction

Here are some of the documents from Apple that informed the style guide. If something isn't mentioned here, it's probably covered in great detail in one of these:

- The Objective-C Programming Language
- Cocoa Fundamentals Guide
- Coding Guidelines for Cocoa
- iOS App Programming Guide

This style guide conforms to IETF's RFC 2119. In particular, code which goes against the RECOMMENDED/SHOULD style is allowed, but should be carefully considered.

Table of Contents

- Dot Notation Syntax
- Spacing
- Conditionals
 - Ternary Operator
- Error handling
- Methods
- Variables
- Naming
 - Categories

-
- Comments
 - Init & Dealloc
 - Literals
 - CGRect Functions
 - Constants
 - Enumerated Types
 - Bitmasks
 - Private Properties
 - Image Naming
 - Booleans
 - Singletons
 - Imports
 - Protocols
 - Xcode Project

Dot Notation Syntax

Dot notation is RECOMMENDED over bracket notation for getting and setting properties.

For example:

```
1 view.backgroundColor = UIColor.orangeColor;
2 UIApplication.sharedApplication.delegate;
```

Not:

```
1 [view setBackgroundColor:[UIColor orangeColor]];
2 [UIApplication sharedApplication].delegate;
```

Spacing

- Indentation MUST use 4 spaces. Never indent with tabs. Be sure to set this preference in Xcode.
- Method braces and other braces (**if/else/switch/while** etc.) MUST open on the same line as the statement. Braces MUST close on a new line.

For example:

```
1 if (user.isHappy) {
2     // Do something
3 }
4 else {
5     // Do something else
```

```
6 }
```

- There SHOULD be exactly one blank line between methods to aid in visual clarity and organization.
- Whitespace within methods MAY separate functionality, though this inclination often indicates an opportunity to split the method into several, smaller methods. In methods with long or verbose names, a single line of whitespace MAY be used to provide visual separation before the method's body.
- `@synthesize` and `@dynamic` MUST each be declared on new lines in the implementation.

Conditionals

Conditional bodies MUST use braces even when a conditional body could be written without braces (e.g., it is one line only) to prevent errors. These errors include adding a second line and expecting it to be part of the if-statement. Another, even more dangerous defect can happen where the line “inside” the if-statement is commented out, and the next line unwittingly becomes part of the if-statement. In addition, this style is more consistent with all other conditionals, and therefore more easily scannable.

For example:

```
1 if (!error) {  
2     return success;  
3 }
```

Not:

```
1 if (!error)  
2     return success;
```

or

```
1 if (!error) return success;
```

Ternary Operator

The intent of the ternary operator, `?`, is to increase clarity or code neatness. The ternary SHOULD only evaluate a single condition per expression. Evaluating multiple conditions is usually more understandable as an if statement or refactored into named variables.

For example:

```
1 result = a > b ? x : y;
```

Not:

```
1 result = a > b ? x = c > d ? c : d : y;
```

Error Handling

When methods return an error parameter by reference, code **MUST** switch on the returned value and **MUST NOT** switch on the error variable.

For example:

```
1 NSError *error;  
2 if (![self trySomethingWithError:&error]) {  
3     // Handle Error  
4 }
```

Not:

```
1 NSError *error;  
2 [self trySomethingWithError:&error];  
3 if (error) {  
4     // Handle Error  
5 }
```

Some of Apple's APIs write garbage values to the error parameter (if non-NULL) in successful cases, so switching on the error can cause false negatives (and subsequently crash).

Methods

- In method signatures, there **SHOULD** be a space after the scope (– or + symbol). There **SHOULD** be a space between the method segments.

For example:

```
1 - (void)setExampleText:(NSString *)text image:(UIImage *)image;
```

- Methods exceeding 80 characters **SHOULD** be represented like a form with a new line after each argument

For example:

```
1 - (void)setExampleText:(NSString *)text
2         image:(UIImage *)image
3         color:(UIColor *)color
4         alternativeText:(NSString *)altText;
```

Variables

Variables SHOULD be named descriptively, with the variable's name clearly communicating what the variable *is* and pertinent information a programmer needs to use that value properly.

For example:

- `NSString *title`: It is reasonable to assume a “title” is a string.
- `NSString *titleHTML`: This indicates a title that may contain HTML which needs parsing for display. *“HTML” is needed for a programmer to use this variable effectively.*
- `NSAttributedString *titleAttributedString`: A title, already formatted for display. *AttributedString hints that this value is not just a vanilla title, and adding it could be a reasonable choice depending on context.*
- `NSDate *now`: No further clarification is needed.
- `NSDate *lastModifiedDate`: Simply `lastModified` can be ambiguous; depending on context, one could reasonably assume it is one of a few different types.
- `NSURL *URL` vs. `NSString *URLString`: In situations when a value can reasonably be represented by different classes, it is often useful to disambiguate in the variable's name.
- `NSString *releaseDateString`: Another example where a value could be represented by another class, and the name can help disambiguate.

Single letter variable names are NOT RECOMMENDED, except as simple counter variables in loops.

Asterisks indicating a type is a pointer MUST be “attached to” the variable name. **For example,** `NSString *text` **not** `NSString* text` or `NSString * text`, except in the case of constants (`NSString * const NYTConstantString`).

Property definitions SHOULD be used in place of naked instance variables whenever possible. Direct instance variable access SHOULD be avoided except in initializer methods (`init`, `initWithCoder:`, etc...), `dealloc` methods and within custom setters and getters. For more information, see Apple's docs on using accessor methods in initializer methods and `dealloc`.

For example:

```
1 @interface NYTSection : NSObject
2
3 @property (nonatomic) NSString *headline;
```

```
4
5 @end
```

Not:

```
1 @interface NYTSection : NSObject {
2     NSString *headline;
3 }
```

Variable Qualifiers When it comes to the variable qualifiers introduced with ARC, the qualifier (`__strong`, `__weak`, `__unsafe_unretained`, `__autoreleasing`) SHOULD be placed between the asterisks and the variable name, e.g., `NSString * __weak text`.

Naming

Apple naming conventions SHOULD be adhered to wherever possible, especially those related to memory management rules (NARC).

Long, descriptive method and variable names are good.

For example:

```
1 UIButton *settingsButton;
```

Not

```
1 UIButton *setBut;
```

A three letter prefix (e.g., `NYT`) MUST be used for class names and constants, however MAY be omitted for Core Data entity names. Constants MUST be camel-case with all words capitalized and prefixed by the related class name for clarity. A two letter prefix (e.g., `NS`) is reserved for use by Apple.

For example:

```
1 static const NSTimeInterval
   NYTArticleViewControllerNavigationFadeAnimationDuration = 0.3;
```

Not:

```
1 static const NSTimeInterval fadetime = 1.7;
```

Properties and local variables MUST be camel-case with the leading word being lowercase.

Instance variables MUST be camel-case with the leading word being lowercase, and MUST be prefixed with an underscore. This is consistent with instance variables synthesized automatically by LLVM. **If LLVM can synthesize the variable automatically, then let it.**

For example:

```
1 @synthesize descriptiveVariableName = _descriptiveVariableName;
```

Not:

```
1 id varnm;
```

Categories

Categories are RECOMMENDED to concisely segment functionality and should be named to describe that functionality.

For example:

```
1 @interface UIViewController (NYTMediaPlaying)
2 @interface NSString (NSStringEncodingDetection)
```

Not:

```
1 @interface NYTAdvertisement (private)
2 @interface NSString (NYTAdditions)
```

Methods and properties added in categories MUST be named with an app- or organization-specific prefix. This avoids unintentionally overriding an existing method, and it reduces the chance of two categories from different libraries adding a method of the same name. (The Objective-C runtime doesn't specify which method will be called in the latter case, which can lead to unintended effects.)

For example:

```
1 @interface NSArray (NYTAccessors)
2 - (id)nyt_objectOrNilAtIndex:(NSUInteger)index;
3 @end
```

Not:

```
1 @interface NSArray (NYTAccessors)
2 - (id)objectOrNilAtIndex:(NSUInteger)index;
3 @end
```

Comments

When they are needed, comments SHOULD be used to explain **why** a particular piece of code does something. Any comments that are used MUST be kept up-to-date or deleted.

Block comments are NOT RECOMMENDED, as code should be as self-documenting as possible, with only the need for intermittent, few-line explanations. This does not apply to those comments used to generate documentation.

init and dealloc

`dealloc` methods SHOULD be placed at the top of the implementation, directly after the `@synthesize` and `@dynamic` statements. `init` methods SHOULD be placed directly below the `dealloc` methods of any class.

`init` methods should be structured like this:

```
1 - (instancetype)init {
2     self = [super init]; // or call the designated initializer
3     if (self) {
4         // Custom initialization
5     }
6     return self;
7 }
```

Literals

`NSString`, `NSDictionary`, `NSArray`, and `NSNumber` literals SHOULD be used whenever creating immutable instances of those objects. Pay special care that `nil` values not be passed into `NSArray` and `NSDictionary` literals, as this will cause a crash.

For example:

```
1 NSArray *names = @[@"Brian", @"Matt", @"Chris", @"Alex", @"Steve", @"Paul"];
2 NSDictionary *productManagers = @{@"iPhone": @"Kate", @"iPad": @"Kamal", @"Mobile Web": @"Bill"};
3 NSNumber *shouldUseLiterals = @YES;
4 NSNumber *buildingZipCode = @10018;
```

Not:

```
1 NSArray *names = [NSArray arrayWithObjects:@"Brian", @"Matt", @"Chris", @"Alex", @"Steve", @"Paul", nil];
2 NSDictionary *productManagers = [NSDictionary dictionaryWithObjectsAndKeys:@"Kate", @"iPhone", @"Kamal", @"iPad", @"Bill", @"Mobile Web", nil];
3 NSNumber *shouldUseLiterals = [NSNumber numberWithBool:YES];
4 NSNumber *buildingZipCode = [NSNumber numberWithInt:10018];
```

CGRect Functions

When accessing the `x`, `y`, `width`, or `height` of a `CGRect`, code MUST use the `CGGeometry` functions instead of direct struct member access. From Apple's `CGGeometry` reference:

All functions described in this reference that take `CGRect` data structures as inputs implicitly standardize those rectangles before calculating their results. For this reason, your applications should avoid directly reading and writing the data stored in the `CGRect` data structure. Instead, use the functions described here to manipulate rectangles and to retrieve their characteristics.

For example:

```
1 CGRect frame = self.view.frame;
2
3 CGFloat x = CGRectGetMinX(frame);
4 CGFloat y = CGRectGetMinY(frame);
5 CGFloat width = CGRectGetWidth(frame);
6 CGFloat height = CGRectGetHeight(frame);
```

Not:

```
1 CGRect frame = self.view.frame;
2
3 CGFloat x = frame.origin.x;
4 CGFloat y = frame.origin.y;
5 CGFloat width = frame.size.width;
6 CGFloat height = frame.size.height;
```

Constants

Constants are RECOMMENDED over in-line string literals or numbers, as they allow for easy reproduction of commonly used variables and can be quickly changed without the need for find and replace. Constants MUST be declared as **static** constants. Constants MAY be declared as `#define` when explicitly being used as a macro.

For example:

```
1 static NSString * const NYTAboutViewControllerCompanyName = @"The New
   York Times Company";
2
3 static const CGFloat NYTImageThumbnailHeight = 50.0;
```

Not:

```
1 #define CompanyName @"The New York Times Company"
2
```

```
3 #define thumbnailHeight 2
```

Enumerated Types

When using `enums`, the new fixed underlying type specification MUST be used; it provides stronger type checking and code completion. The SDK includes a macro to facilitate and encourage use of fixed underlying types: `NS_ENUM()`.

Example:

```
1 typedef NS_ENUM(NSInteger, NYTAdRequestState) {
2     NYTAdRequestStateInactive,
3     NYTAdRequestStateLoading
4 };
```

Bitmasks

When working with bitmasks, the `NS_OPTIONS` macro MUST be used.

Example:

```
1 typedef NS_OPTIONS(NSUInteger, NYTAdCategory) {
2     NYTAdCategoryAutos      = 1 << 0,
3     NYTAdCategoryJobs      = 1 << 1,
4     NYTAdCategoryRealState = 1 << 2,
5     NYTAdCategoryTechnology = 1 << 3
6 };
```

Private Properties

Private properties SHALL be declared in class extensions (anonymous categories) in the implementation file of a class.

For example:

```
1 @interface NYTAdvertisement ()
2
3 @property (nonatomic, strong) GADBannerView *googleAdView;
4 @property (nonatomic, strong) ADBannerView *iAdView;
5 @property (nonatomic, strong) UIWebView *adXWebView;
6
7 @end
```

Image Naming

Image names should be named consistently to preserve organization and developer sanity. Images SHOULD be named as one camel case string with a description of their purpose, followed by the unprefixed name of the class or property they are customizing (if there is one), followed by a further description of color and/or placement, and finally their state.

For example:

- `RefreshBarButtonItem/RefreshBarButtonItem@2x` and `RefreshBarButtonItemSelected/RefreshBarButtonItemSelected@2x`
- `ArticleNavigationBarWhite/ArticleNavigationBarWhite@2x` and `ArticleNavigationBar/ArticleNavigationBarBlackSelected@2x`.

Images that are used for a similar purpose SHOULD be grouped in respective groups in an Images folder or Asset Catalog.

Booleans

Values MUST NOT be compared directly to `YES`, because `YES` is defined as 1, and a `BOOL` in Objective-C is a `CHAR` type that is 8 bits long (so a value of 11111110 will return `NO` if compared to `YES`).

For an object pointer:

```
1 if (!someObject) {  
2 }  
3  
4 if (someObject == nil) {  
5 }
```

For a `BOOL` value:

```
1 if (isAwesome)  
2 if (!someNumber.boolValue)  
3 if (someNumber.boolValue == NO)
```

Not:

```
1 if (isAwesome == YES) // Never do this.
```

If the name of a `BOOL` property is expressed as an adjective, the property's name MAY omit the `is` prefix but should specify the conventional name for the getter.

For example:

```
1 @property (assign, getter=isEditable) BOOL editable;
```

Text and example taken from the Cocoa Naming Guidelines.

Singletons

Singleton objects SHOULD use a thread-safe pattern for creating their shared instance.

```
1 + (instancetype)sharedInstance {
2     static id sharedInstance = nil;
3
4     static dispatch_once_t onceToken;
5     dispatch_once(&onceToken, ^{
6         sharedInstance = [[[self class] alloc] init];
7     });
8
9     return sharedInstance;
10 }
```

This will prevent possible and sometimes frequent crashes.

Imports

If there is more than one import statement, statements MUST be grouped together. Groups MAY be commented.

Note: For modules use the @import syntax.

```
1 // Frameworks
2 @import QuartzCore;
3
4 // Models
5 #import "NYTUser.h"
6
7 // Views
8 #import "NYTButton.h"
9 #import "NYTUIView.h"
```

Protocols

In a delegate or data source protocol, the first parameter to each method SHOULD be the object sending the message.

This helps disambiguate in cases when an object is the delegate for multiple similarly-typed objects, and it helps clarify intent to readers of a class implementing these delegate methods.

For example:

```
1 - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(  
    NSIndexPath *)indexPath;
```

Not:

```
1 - (void)didSelectRowAtIndexPath:(NSIndexPath *)indexPath;
```

Xcode project

The physical files SHOULD be kept in sync with the Xcode project files in order to avoid file sprawl. Any Xcode groups created SHOULD be reflected by folders in the filesystem. Code SHOULD be grouped not only by type, but also by feature for greater clarity.

Target Build Setting “Treat Warnings as Errors” SHOULD be enabled. Enable as many additional warnings as possible. If you need to ignore a specific warning, use Clang’s pragma feature.

Other Objective-C Style Guides

If ours doesn’t fit your tastes, have a look at some other style guides:

- [Google](#)
- [GitHub](#)
- [Adium](#)
- [Sam Soffes](#)
- [CocoaDevCentral](#)
- [Luke Redpath](#)
- [Marcus Zarra](#)
- [Wikimedia](#)