styleguide

Google Objective-C Style Guide

> Objective-C is a dynamic, object-oriented extension of C. It's designed to be easy to use and read, while enabling sophisticated object-oriented design. It is the primary development language for applications on OS X and on iOS.
>
> Apple has already written a very good, and widely accepted, Cocoa Coding Guidelines for Objective-C. Please read it in addition to this guide.
>
> The purpose of this document is to describe the Objective-C (and Objective-C++) coding guidelines and practices that should be used for iOS and OS X code. These guidelines have evolved and been proven over time on other projects and teams. Open-source projects developed by Google conform to the requirements in this guide.
>
> Note that this guide is not an Objective-C tutorial. We assume that the reader is familiar with the language. If you are new to Objective-C or need a refresher, please read Programming with Objective-C.

Principles

Optimize for the reader, not the writer

Codebases often have extended lifetimes and more time is spent reading the code than writing it. We explicitly choose to optimize for the experience of our average software engineer reading, maintaining, and debugging code in our codebase rather than the ease of writing said code. For example, when something surprising or unusual is happening in a snippet of code, leaving textual hints for the reader is valuable.

Be consistent

When the style guide allows multiple options it is preferable to pick one option over mixed usage of multiple options. Using one style consistently throughout a codebase lets engineers focus on other (more important) issues. Consistency also enables better automation because consistent code allows more efficient development and operation of tools that format or refactor code. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop

worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.

Be consistent with Apple SDKs

Consistency with the way Apple SDKs use Objective-C has value for the same reasons as consistency within our code base. If an Objective-C feature solves a problem that's an argument for using it. However, sometimes language features and idioms are flawed, or were just designed with assumptions that are not universal. In those cases it is appropriate to constrain or ban language features or idioms.

Style rules should pull their weight

The benefit of a style rule must be large enough to justify asking engineers to remember it. The benefit is measured relative to the codebase we would get without the rule, so a rule against a very harmful practice may still have a small benefit if people are unlikely to do it anyway. This principle mostly explains the rules we don't have, rather than the rules we do: for example, goto contravenes many of the following principles, but is not discussed due to its extreme rarity.

Example

They say an example is worth a thousand words, so let's start off with an example that should give you a feel for the style, spacing, naming, and so on.

Here is an example header file, demonstrating the correct commenting and spacing for an `@interface` declaration.

```
// GOOD:

#import <Foundation/Foundation.h>

@class Bar;

/**
 * A sample class demonstrating good Objective-C style. All interfaces,
 * categories, and protocols (read: all non-trivial top-level declarations
 * in a header) MUST be commented. Comments must also be adjacent to the
 * object they're documenting.
 */
@interface Foo : NSObject

/** The retained Bar. */
@property(nonatomic) Bar *bar;
```

```
/** The current drawing attributes. */
@property(nonatomic, copy) NSDictionary<NSString *, NSNumber *> *attributes;

/**
 * Convenience creation method.
 * See -initWithBar: for details about @c bar.
 *
 * @param bar The string for fooing.
 * @return An instance of Foo.
 */
+ (instancetype)fooWithBar:(Bar *)bar;

/**
 * Initializes and returns a Foo object using the provided Bar instance.
 *
 * @param bar A string that represents a thing that does a thing.
 */
- (instancetype)initWithBar:(Bar *)bar NS_DESIGNATED_INITIALIZER;

/**
 * Does some work with @c blah.
 *
 * @param blah
 * @return YES if the work was completed; NO otherwise.
 */
- (BOOL)doWorkWithBlah:(NSString *)blah;

@end
```

An example source file, demonstrating the correct commenting and spacing for the `@implementation` of an interface.

```
// GOOD:

#import "Shared/Util/Foo.h"

@implementation Foo {
  /** The string used for displaying "hi". */
  NSString *_string;
}

+ (instancetype)fooWithBar:(Bar *)bar {
  return [[self alloc] initWithBar:bar];
}

- (instancetype)init {
  // Classes with a custom designated initializer should always override
  // the superclass's designated initializer.
```

```
  return [self initWithBar:nil];
}

- (instancetype)initWithBar:(Bar *)bar {
  self = [super init];
  if (self) {
    _bar = [bar copy];
    _string = [[NSString alloc] initWithFormat:@"hi %d", 3];
    _attributes = @{
      @"color" : [UIColor blueColor],
      @"hidden" : @NO
    };
  }
  return self;
}

- (BOOL)doWorkWithBlah:(NSString *)blah {
  // Work should be done here.
  return NO;
}

@end
```

Naming

Names should be as descriptive as possible, within reason. Follow standard Objective-C naming rules.

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:

```
// GOOD:

// Good names.
int numberOfErrors = 0;
int completedConnectionsCount = 0;
tickets = [[NSMutableArray alloc] init];
userInfo = [someObject object];
port = [network port];
NSDate *gAppLaunchDate;



// AVOID:

// Names to avoid.
```

```
int w;
int nerr;
int nCompConns;
tix = [[NSMutableArray alloc] init];
obj = [someObject object];
p = [network port];
```

Any class, category, method, function, or variable name should use all capitals for acronyms and initialisms within the name. This follows Apple's standard of using all capitals within a name for acronyms such as URL, ID, TIFF, and EXIF.

Names of C functions and typedefs should be capitalized and use camel case as appropriate for the surrounding code.

File Names

File names should reflect the name of the class implementation that they contain—including case.

Follow the convention that your project uses. File extensions should be as follows:

Extension	Type
.h	C/C++/Objective-C header file
.m	Objective-C implementation file
.mm	Objective-C++ implementation file
.cc	Pure C++ implementation file
.c	C implementation file

Files containing code that may be shared across projects or used in a large project should have a clearly unique name, typically including the project or class prefix.

File names for categories should include the name of the class being extended, like GTMNSString+Utils.h or NSTextView+GTMAutocomplete.h

Prefixes

Prefixes are commonly required in Objective-C to avoid naming collisions in a global namespace. Classes, protocols, global functions, and global constants should generally be named with a prefix that begins with a capital letter followed by one or more capital letters or numbers.

WARNING: Apple reserves two-letter prefixes—see Conventions in Programming with Objective-C —so prefixes with a minimum of three characters are considered best practice.

```
// GOOD:

/** An example error domain. */
extern NSString *GTMExampleErrorDomain;

/** Gets the default time zone. */
extern NSTimeZone *GTMGetDefaultTimeZone(void);

/** An example delegate. */
@protocol GTMExampleDelegate <NSObject>
@end

/** An example class. */
@interface GTMExample : NSObject
@end
```

Class Names

Class names (along with category and protocol names) should start as uppercase and use mixed case to delimit words.

Classes and protocols in code shared across multiple applications must have an appropriate prefix (e.g. GTMSendMessage). Prefixes are recommended, but not required, for other classes and protocols.

Category Naming

Category names should start with an appropriate prefix identifying the category as part of a project or open for general use.

Category source file names should begin with the class being extended followed by a plus sign and the name of the category, e.g., `NSString+GTMParsing.h` . Methods in a category should be prefixed with a lowercase version of the prefix used for the category name followed by an underscore (e.g., `gtm_myCategoryMethodOnAString:`) in order to prevent collisions in Objective-C's global namespace.

There should be a single space between the class name and the opening parenthesis of the category.

```
// GOOD:

// UIViewController+GTMCrashReporting.h
```

```
/** A category that adds metadata to include in crash reports to UIViewController. */
@interface UIViewController (GTMCrashReporting)

/** A unique identifier to represent the view controller in crash reports. */
@property(nonatomic, setter=gtm_setUniqueIdentifier:) int gtm_uniqueIdentifier;

/** Returns an encoded representation of the view controller's current state. */
- (nullable NSData *)gtm_encodedState;

@end
```

If a class is not shared with other projects, categories extending it may omit name prefixes and method name prefixes.

```
// GOOD:

/** This category extends a class that is not shared with other projects. */
@interface XYZDataObject (Storage)
- (NSString *)storageIdentifier;
@end
```

Objective-C Method Names

Method and parameter names typically start as lowercase and then use mixed case.

Proper capitalization should be respected, including at the beginning of names.

```
// GOOD:

+ (NSURL *)URLWithString:(NSString *)URLString;
```

The method name should read like a sentence if possible, meaning you should choose parameter names that flow with the method name. Objective-C method names tend to be very long, but this has the benefit that a block of code can almost read like prose, thus rendering many implementation comments unnecessary.

Use prepositions and conjunctions like “with”, “from”, and “to” in the second and later parameter names only where necessary to clarify the meaning or behavior of the method.

```
// GOOD:

- (void)addTarget:(id)target action:(SEL)action;                    // GOOD; no
- (CGPoint)convertPoint:(CGPoint)point fromView:(UIView *)view;      // GOOD; co
```

```
- (void)replaceCharactersInRange:(NSRange)aRange
            withAttributedString:(NSAttributedString *)attributedString;  // GOOD.
```

A method that returns an object should have a name beginning with a noun identifying the object returned:

```
// GOOD:

- (Sandwich *)sandwich;      // GOOD.


// AVOID:

- (Sandwich *)makeSandwich;  // AVOID.
```

An accessor method should be named the same as the object it's getting, but it should not be prefixed with the word `get`. For example:

```
// GOOD:

- (id)delegate;     // GOOD.


// AVOID:

- (id)getDelegate;  // AVOID.
```

Accessors that return the value of boolean adjectives have method names beginning with `is`, but property names for those methods omit the `is`.

Dot notation is used only with property names, not with method names.

```
// GOOD:

@property(nonatomic, getter=isGlorious) BOOL glorious;
- (BOOL)isGlorious;

BOOL isGood = object.glorious;      // GOOD.
BOOL isGood = [object isGlorious];  // GOOD.


// AVOID:
```

```
BOOL isGood = object.isGlorious;    // AVOID.

// GOOD:

NSArray<Frog *> *frogs = [NSArray<Frog *> arrayWithObject:frog];
NSEnumerator *enumerator = [frogs reverseObjectEnumerator];  // GOOD.


// AVOID:

NSEnumerator *enumerator = frogs.reverseObjectEnumerator;    // AVOID.
```

See Apple's Guide to Naming Methods for more details on Objective-C naming.

These guidelines are for Objective-C methods only. C++ method names continue to follow the rules set in the C++ style guide.

Function Names

Function names should start with a capital letter and have a capital letter for each new word (a.k.a. "camel case" or "Pascal case").

```
// GOOD:

static void AddTableEntry(NSString *tableEntry);
static BOOL DeleteFile(const char *filename);
```

Because Objective-C does not provide namespacing, non-static functions should have a prefix that minimizes the chance of a name collision.

```
// GOOD:

extern NSTimeZone *GTMGetDefaultTimeZone(void);
extern NSString *GTMGetURLScheme(NSURL *URL);
```

Variable Names

Variable names typically start with a lowercase and use mixed case to delimit words.

Instance variables have leading underscores. File scope or global variables have a prefix `g`. For example: `myLocalVariable`, `_myInstanceVariable`, `gMyGlobalVariable`.

Common Variable Names

Readers should be able to infer the variable type from the name, but do not use Hungarian notation for syntactic attributes, such as the static type of a variable (int or pointer).

File scope or global variables (as opposed to constants) declared outside the scope of a method or function should be rare, and should have the prefix g.

```
// GOOD:

static int gGlobalCounter;
```

Instance Variables

Instance variable names are mixed case and should be prefixed with an underscore, like `_usernameTextField`.

NOTE: Google's previous convention for Objective-C ivars was a trailing underscore. Existing projects may opt to continue using trailing underscores in new code in order to maintain consistency within the project codebase. Consistency of prefix or suffix underscores should be maintained within each class.

Constants

Constant symbols (const global and static variables and constants created with #define) should use mixed case to delimit words.

Global and file scope constants should have an appropriate prefix.

```
// GOOD:

extern NSString *const GTLServiceErrorDomain;

typedef NS_ENUM(NSInteger, GTLServiceError) {
  GTLServiceErrorQueryResultMissing = -3000,
  GTLServiceErrorWaitTimedOut       = -3001,
};
```

Because Objective-C does not provide namespacing, constants with external linkage should have a prefix that minimizes the chance of a name collision, typically like `ClassNameConstantName` or `ClassNameEnumName`.

For interoperability with Swift code, enumerated values should have names that extend the typedef name:

```
// GOOD:

typedef NS_ENUM(NSInteger, DisplayTinge) {
  DisplayTingeGreen = 1,
  DisplayTingeBlue = 2,
};
```

A lowercase k can be used as a standalone prefix for constants of static storage duration declared within implementation files:

```
// GOOD:

static const int kFileCount = 12;
static NSString *const kUserKey = @"kUserKey";
```

NOTE: Previous convention was for public constant names to begin with a lowercase k followed by a project-specific prefix. This practice is no longer recommended.

Types and Declarations

Method Declarations

As shown in the example, the recommended order for declarations in an `@interface` declaration are: properties, class methods, initializers, and then finally instance methods. The class methods section should begin with any convenience constructors.

Local Variables

Declare variables in the narrowest practical scopes, and close to their use. Initialize variables in their declarations.

```
// GOOD:

CLLocation *location = [self lastKnownLocation];
for (int meters = 1; meters < 10; meters++) {
  reportFrogsWithinRadius(location, meters);
}
```

Occasionally, efficiency will make it more appropriate to declare a variable outside the scope of its use. This example declares meters separate from initialization, and needlessly sends the lastKnownLocation message each time through the loop:

```
// AVOID:

int meters;                                             // AVOID.
for (meters = 1; meters < 10; meters++) {
  CLLocation *location = [self lastKnownLocation];  // AVOID.
  reportFrogsWithinRadius(location, meters);
}
```

Under Automatic Reference Counting, strong and weak pointers to Objective-C objects are automatically initialized to `nil`, so explicit initialization to `nil` is not required for those common cases. However, automatic initialization does *not* occur for many Objective-C pointer types, including object pointers declared with the `__unsafe_unretained` ownership qualifier and CoreFoundation object pointer types. When in doubt, prefer to initialize all Objective-C local variables.

Unsigned Integers

Avoid unsigned integers except when matching types used by system interfaces.

Subtle errors crop up when doing math or counting down to zero using unsigned integers. Rely only on signed integers in math expressions except when matching NSUInteger in system interfaces.

```
// GOOD:

NSUInteger numberOfObjects = array.count;
for (NSInteger counter = numberOfObjects - 1; counter > 0; --counter)


// AVOID:

for (NSUInteger counter = numberOfObjects - 1; counter > 0; --counter)  // AVOID.
```

Unsigned integers may be used for flags and bitmasks, though often NS_OPTIONS or NS_ENUM will be more appropriate.

Types with Inconsistent Sizes

Due to sizes that differ in 32- and 64-bit builds, avoid types long, NSInteger, NSUInteger, and CGFloat except when matching system interfaces.

Types long, NSInteger, NSUInteger, and CGFloat vary in size between 32- and 64-bit builds. Use of these types is appropriate when handling values exposed by system interfaces, but they should

be avoided for most other computations.

```
// GOOD:
int32_t scalar1 = proto.intValue;
int64_t scalar2 = proto.longValue;
NSUInteger numberOfObjects = array.count;
CGFloat offset = view.bounds.origin.x;

// AVOID:
NSInteger scalar2 = proto.longValue;  // AVOID.
```

File and buffer sizes often exceed 32-bit limits, so they should be declared using `int64_t`, not with `long`, `NSInteger`, or `NSUInteger`.

Comments

Comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names and then trying to explain them through comments.

Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but use a consistent style. When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous—the next one may be you!

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.

```
// GOOD:

/**
 * A delegate for NSApplication to handle notifications about app
 * launch and shutdown. Owned by the main app controller.
 */
@interface MyAppDelegate : NSObject {
  /**
   * The background task in progress, if any. This is initialized
   * to the value UIBackgroundTaskInvalid.
   */
  UIBackgroundTaskIdentifier _backgroundTaskID;
}

/** The factory that creates and manages fetchers for the app. */
@property(nonatomic) GTMSessionFetcherService *fetcherService;

@end
```

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.

```
// GOOD:

// Set the property to nil before invoking the completion handler to
// avoid the risk of reentrancy leading to the callback being
// invoked again.
CompletionHandler handler = self.completionHandler;
self.completionHandler = nil;
handler();
```

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.

```
// GOOD:

[self doSomethingWithALongName];  // Two spaces before the comment.
[self doSomethingShort];          // More spacing to align the comment.
```

Disambiguating Symbols

Where needed to avoid ambiguity, use backticks or vertical bars to quote variable names and symbols in comments in preference to using quotation marks or naming the symbols inline.

In Doxygen-style comments, prefer demarcating symbols with a monospace text command, such as `@c`.

Demarcation helps provide clarity when a symbol is a common word that might make the sentence read like it was poorly constructed. A common example is the symbol `count`:

```
// GOOD:

// Sometimes `count` will be less than zero.
```

or when quoting something which already contains quotes

```
// GOOD:

// Remember to call `StringWithoutSpaces("foo bar baz")`
```

Backticks or vertical bars are not needed when a symbol is self-apparent.

```
// GOOD:

// This class serves as a delegate to GTMDepthCharge.
```

Doxygen formatting is also suitable for identifying symbols.

```
// GOOD:

/** @param maximum The highest value for @c count. */
```

Object Ownership

For objects not managed by ARC, make the pointer ownership model as explicit as possible when it falls outside the most common Objective-C usage idioms.

Manual Reference Counting

Instance variables for NSObject-derived objects are presumed to be retained; if they are not retained, they should be either commented as weak or declared with the `__weak` lifetime qualifier.

An exception is in Mac software for instance variables labeled as `@IBOutlets`, which are presumed to not be retained.

Where instance variables are pointers to Core Foundation, C++, and other non-Objective-C objects, they should always be declared with strong and weak comments to indicate which pointers are and are not retained. Core Foundation and other non-Objective-C object pointers require explicit memory management, even when building for automatic reference counting.

Examples of strong and weak declarations:

```
// GOOD:

@interface MyDelegate : NSObject

@property(nonatomic) NSString *doohickey;
@property(nonatomic, weak) NSString *parent;

@end


@implementation MyDelegate {
  IBOutlet NSButton *_okButton;  // Normal NSControl; implicitly weak on Mac only

  AnObjcObject *_doohickey;  // My doohickey
  __weak MyObjcParent *_parent;  // To send messages back (owns this instance)

  // non-NSObject pointers...
  CWackyCPPClass *_wacky;  // Strong, some cross-platform object
  CFDictionaryRef *_dict;  // Strong
}
@end
```

Automatic Reference Counting

Object ownership and lifetime are explicit when using ARC, so no additional comments are required for automatically retained objects.

C Language Features

Macros

Avoid macros, especially where `const` variables, enums, XCode snippets, or C functions may be used instead.

Macros make the code you see different from the code the compiler sees. Modern C renders traditional uses of macros for constants and utility functions unnecessary. Macros should only be used when there is no other solution available.

Where a macro is needed, use a unique name to avoid the risk of a symbol collision in the compilation unit. If practical, keep the scope limited by `#undefining` the macro after its use.

Macro names should use `SHOUTY_SNAKE_CASE` —all uppercase letters with underscores between words. Function-like macros may use C function naming practices. Do not define macros that appear to be C or Objective-C keywords.

```
// GOOD:

#define GTM_EXPERIMENTAL_BUILD ...      // GOOD

// Assert unless X > Y
#define GTM_ASSERT_GT(X, Y) ...         // GOOD, macro style.

// Assert unless X > Y
#define GTMAssertGreaterThan(X, Y) ...  // GOOD, function style.


// AVOID:

#define kIsExperimentalBuild ...        // AVOID

#define unless(X) if(!(X))              // AVOID
```

Avoid macros that expand to unbalanced C or Objective-C constructs. Avoid macros that introduce scope, or may obscure the capturing of values in blocks.

Avoid macros that generate class, property, or method definitions in headers to be used as public API. These only make the code hard to understand, and the language already has better ways of doing this.

Avoid macros that generate method implementations, or that generate declarations of variables that are later used outside of the macro. Macros shouldn't make code hard to understand by hiding where and how a variable is declared.

```
// AVOID:

#define ARRAY_ADDER(CLASS) \
  -(void)add ## CLASS ## :(CLASS *)obj toArray:(NSMutableArray *)array

ARRAY_ADDER(NSString) {
  if (array.count > 5) {              // AVOID -- where is 'array' defined?
    ...
```

```
    }
}
```

Examples of acceptable macro use include assertion and debug logging macros that are conditionally compiled based on build settings—often, these are not compiled into release builds.

Nonstandard Extensions

Nonstandard extensions to C/Objective-C may not be used unless otherwise specified.

Compilers support various extensions that are not part of standard C. Examples include compound statement expressions (e.g. `foo = ({ int x; Bar(&x); x })`).

`__attribute__` is an approved exception, as it is used in Objective-C API specifications.

The binary form of the conditional operator, `A ?: B`, is an approved exception.

Cocoa and Objective-C Features

Identify Designated Initializer

Clearly identify your designated initializer.

It is important for those who might be subclassing your class that the designated initializer be clearly identified. That way, they only need to override a single initializer (of potentially several) to guarantee the initializer of their subclass is called. It also helps those debugging your class in the future understand the flow of initialization code if they need to step through it. Identify the designated initializer using comments or the `NS_DESIGNATED_INITIALIZER` macro. If you use `NS_DESIGNATED_INITIALIZER`, mark unsupported initializers with `NS_UNAVAILABLE`.

Override Designated Initializer

When writing a subclass that requires an `init...` method, make sure you override the designated initializer of the superclass.

If you fail to override the designated initializer of the superclass, your initializer may not be called in all cases, leading to subtle and very difficult to find bugs.

Overridden NSObject Method Placement

Put overridden methods of NSObject at the top of an `@implementation`.

This commonly applies to (but is not limited to) the `init...`, `copyWithZone:`, and `dealloc` methods. The `init...` methods should be grouped together, followed by other typical

`NSObject` methods such as `description`, `isEqual:`, and `hash`.

Convenience class factory methods for creating instances may precede the `NSObject` methods.

Initialization

Don't initialize instance variables to `0` or `nil` in the `init` method; doing so is redundant.

All instance variables for a newly allocated object are initialized to `0` (except for isa), so don't clutter up the init method by re-initializing variables to `0` or `nil`.

Instance Variables In Headers Should Be @protected or @private

Instance variables should typically be declared in implementation files or auto-synthesized by properties. When ivars are declared in a header file, they should be marked `@protected` or `@private`.

```
// GOOD:

@interface MyClass : NSObject {
 @protected
  id _myInstanceVariable;
}
@end
```

Do Not Use +new

Do not invoke the `NSObject` class method `new`, nor override it in a subclass. `+new` is rarely used and contrasts greatly with initializer usage. Instead, use `+alloc` and `-init` methods to instantiate retained objects.

Keep the Public API Simple

Keep your class simple; avoid "kitchen-sink" APIs. If a method doesn't need to be public, keep it out of the public interface.

Unlike C++, Objective-C doesn't differentiate between public and private methods; any message may be sent to an object. As a result, avoid placing methods in the public API unless they are actually expected to be used by a consumer of the class. This helps reduce the likelihood they'll be called when you're not expecting it. This includes methods that are being overridden from the parent class.

Since internal methods are not really private, it's easy to accidentally override a superclass's "private" method, thus making a very difficult bug to squash. In general, private methods should have a fairly unique name that will prevent subclasses from unintentionally overriding them.

#import and #include

`#import` Objective-C and Objective-C++ headers, and `#include` C/C++ headers.

C/C++ headers include other C/C++ headers using `#include`. Using `#import` on C/C++ headers prevents future inclusions using `#include` and could result in unintended compilation behavior. C/C++ headers should provide their own `#define` guard.

Order of Includes

The standard order for header inclusion is the related header, operating system headers, language library headers, and finally groups of headers for other dependencies.

The related header precedes others to ensure it has no hidden dependencies. For implementation files the related header is the header file. For test files the related header is the header containing the tested interface.

A blank line may separate logically distinct groups of included headers.

Within each group the includes should be ordered alphabetically.

Import headers using their path relative to the project's source directory.

```
// GOOD:

#import "ProjectX/BazViewController.h"

#import <Foundation/Foundation.h>

#include <unistd.h>
#include <vector>

#include "base/basictypes.h"
#include "base/integral_types.h"
#include "util/math/mathutil.h"

#import "ProjectX/BazModel.h"
#import "Shared/Util/Foo.h"
```

Use Umbrella Headers for System Frameworks

Import umbrella headers for system frameworks and system libraries rather than include individual files.

While it may seem tempting to include individual system headers from a framework such as Cocoa or Foundation, in fact it's less work on the compiler if you include the top-level root framework. The root framework is generally pre-compiled and can be loaded much more quickly. In addition, remember to use `@import` or `#import` rather than `#include` for Objective-C frameworks.

```
// GOOD:

@import UIKit;     // GOOD.
#import <Foundation/Foundation.h>     // GOOD.


// AVOID:

#import <Foundation/NSArray.h>        // AVOID.
#import <Foundation/NSString.h>
...
```

Avoid Messaging the Current Object Within Initializers and `-dealloc`

Code in initializers and `-dealloc` should avoid invoking instance methods.

Superclass initialization completes before subclass initialization. Until all classes have had a chance to initialize their instance state any method invocation on self may lead to a subclass operating on uninitialized instance state.

A similar issue exists for `-dealloc`, where a method invocation may cause a class to operate on state that has been deallocated.

One case where this is less obvious is property accessors. These can be overridden just like any other selector. Whenever practical, directly assign to and release ivars in initializers and `-dealloc`, rather than rely on accessors.

```
// GOOD:

- (instancetype)init {
  self = [super init];
  if (self) {
    _bar = 23;  // GOOD.
  }
```

```
  return self;
}
```

Beware of factoring common initialization code into helper methods:

- Methods can be overridden in subclasses, either deliberately, or accidentally due to naming collisions.
- When editing a helper method, it may not be obvious that the code is being run from an initializer.

```
// AVOID:

- (instancetype)init {
  self = [super init];
  if (self) {
    self.bar = 23;  // AVOID.
    [self sharedMethod];  // AVOID. Fragile to subclassing or future extension.
  }
  return self;
}



// GOOD:

- (void)dealloc {
  [_notifier removeObserver:self];  // GOOD.
}



// AVOID:

- (void)dealloc {
  [self removeNotifications];  // AVOID.
}
```

Setters copy NSStrings

Setters taking an `NSString` should always copy the string it accepts. This is often also appropriate for collections like `NSArray` and `NSDictionary`.

Never just retain the string, as it may be a `NSMutableString`. This avoids the caller changing it under you without your knowledge.

Code receiving and holding collection objects should also consider that the passed collection may be mutable, and thus the collection could be more safely held as a copy or mutable copy of the

original.

```
// GOOD:

@property(nonatomic, copy) NSString *name;

- (void)setZigfoos:(NSArray<Zigfoo *> *)zigfoos {
  // Ensure that we're holding an immutable collection.
  _zigfoos = [zigfoos copy];
}
```

Use Lightweight Generics to Document Contained Types

All projects compiling on Xcode 7 or newer versions should make use of the Objective-C lightweight generics notation to type contained objects.

Every `NSArray`, `NSDictionary`, or `NSSet` reference should be declared using lightweight generics for improved type safety and to explicitly document usage.

```
// GOOD:

@property(nonatomic, copy) NSArray<Location *> *locations;
@property(nonatomic, copy, readonly) NSSet<NSString *> *identifiers;

NSMutableArray<MyLocation *> *mutableLocations = [otherObject.locations mutableCopy];
```

If the fully-annotated types become complex, consider using a typedef to preserve readability.

```
// GOOD:

typedef NSSet<NSDictionary<NSString *, NSDate *> *> TimeZoneMappingSet;
TimeZoneMappingSet *timeZoneMappings = [TimeZoneMappingSet setWithObjects:...];
```

Use the most descriptive common superclass or protocol available. In the most generic case when nothing else is known, declare the collection to be explicitly heterogenous using id.

```
// GOOD:

@property(nonatomic, copy) NSArray<id> *unknowns;
```

Avoid Throwing Exceptions

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

This follows the recommendation to use error objects for error delivery in Apple's Introduction to Exception Programming Topics for Cocoa.

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];  // GOOD.
```

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.

Nullability

Interfaces can be decorated with nullability annotations to describe how the interface should be used and how it behaves. Use of nullability regions (e.g., `NS_ASSUME_NONNULL_BEGIN` and `NS_ASSUME_NONNULL_END`) and explicit nullability annotations are both accepted. Prefer using the `_Nullable` and `_Nonnull` keywords over the `__nullable` and `__nonnull` keywords. For Objective-C methods and properties prefer using the context-sensitive, non-underscored keywords, e.g., `nonnull` and `nullable`.

```objc
// GOOD:

/** A class representing an owned book. */
@interface GTMBook : NSObject

/** The title of the book. */
@property(readonly, copy, nonnull) NSString *title;

/** The author of the book, if one exists. */
@property(readonly, copy, nullable) NSString *author;

/** The owner of the book. Setting nil resets to the default owner. */
@property(copy, null_resettable) NSString *owner;

/** Initializes a book with a title and an optional author. */
- (nonnull instancetype)initWithTitle:(nonnull NSString *)title
                               author:(nullable NSString *)author
    NS_DESIGNATED_INITIALIZER;

/** Returns nil because a book is expected to have a title. */
- (nullable instancetype)init;

@end

/** Loads books from the file specified by the given path. */
NSArray<GTMBook *> *_Nullable GTMLoadBooksFromFile(NSString *_Nonnull path);


// AVOID:

NSArray<GTMBook *> *__nullable GTMLoadBooksFromTitle(NSString *__nonnull path);
```

Be careful assuming that a pointer is not null based on a non-null qualifier because the compiler may not guarantee that the pointer is not null.

BOOL Pitfalls

Be careful when converting general integral values to `BOOL`. Avoid comparing directly with `YES`.

`BOOL` in OS X and in 32-bit iOS builds is defined as a signed `char`, so it may have values other than `YES` (`1`) and `NO` (`0`). Do not cast or convert general integral values directly to `BOOL`.

Common mistakes include casting or converting an array's size, a pointer value, or the result of a bitwise logic operation to a `BOOL` that could, depending on the value of the last byte of the integer value, still result in a `NO` value. When converting a general integral value to a `BOOL`, use ternary operators to return a `YES` or `NO` value.

You can safely interchange and convert `BOOL`, `_Bool` and `bool` (see C++ Std 4.7.4, 4.12 and C99 Std 6.3.1.2). Use `BOOL` in Objective-C method signatures.

Using logical operators (`&&`, `||` and `!`) with `BOOL` is also valid and will return values that can be safely converted to `BOOL` without the need for a ternary operator.

```
// AVOID:

- (BOOL)isBold {
  return [self fontTraits] & NSFontBoldTrait;  // AVOID.
}
- (BOOL)isValid {
  return [self stringValue];  // AVOID.
}



// GOOD:

- (BOOL)isBold {
  return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
}
- (BOOL)isValid {
  return [self stringValue] != nil;
}
- (BOOL)isEnabled {
  return [self isValid] && [self isBold];
}
```

Also, don't directly compare `BOOL` variables directly with `YES`. Not only is it harder to read for those well-versed in C, but the first point above demonstrates that return values may not always be what you expect.

```
// AVOID:

BOOL great = [foo isGreat];
if (great == YES) {  // AVOID.
  // ...be great!
}



// GOOD:

BOOL great = [foo isGreat];
if (great) {         // GOOD.
```

```
  // ...be great!
}
```

Interfaces Without Instance Variables

Omit the empty set of braces on interfaces that do not declare any instance variables.

```
// GOOD:

@interface MyClass : NSObject
// Does a lot of stuff.
- (void)fooBarBam;
@end



// AVOID:

@interface MyClass : NSObject {
}
// Does a lot of stuff.
- (void)fooBarBam;
@end
```

Cocoa Patterns

Delegate Pattern

Delegates, target objects, and block pointers should not be retained when doing so would create a retain cycle.

To avoid causing a retain cycle, a delegate or target pointer should be released as soon as it is clear there will no longer be a need to message the object.

If there is no clear time at which the delegate or target pointer is no longer needed, the pointer should only be retained weakly.

Block pointers cannot be retained weakly. To avoid causing retain cycles in the client code, block pointers should be used for callbacks only where they can be explicitly released after they have been called or once they are no longer needed. Otherwise, callbacks should be done via weak delegate or target pointers.

Objective-C++

Style Matches the Language

Within an Objective-C++ source file, follow the style for the language of the function or method you're implementing. In order to minimize clashes between the differing naming styles when mixing Cocoa/Objective-C and C++, follow the style of the method being implemented.

For code in an `@implementation` block, use the Objective-C naming rules. For code in a method of a C++ class, use the C++ naming rules.

For code in an Objective-C++ file outside of a class implementation, be consistent within the file.

```
// GOOD:

// file: cross_platform_header.h

class CrossPlatformAPI {
 public:
  ...
  int DoSomethingPlatformSpecific();  // impl on each platform
 private:
  int an_instance_var_;
};

// file: mac_implementation.mm
#include "cross_platform_header.h"

/** A typical Objective-C class, using Objective-C naming. */
@interface MyDelegate : NSObject {
 @private
  int _instanceVar;
  CrossPlatformAPI* _backEndObject;
}

- (void)respondToSomething:(id)something;

@end

@implementation MyDelegate

- (void)respondToSomething:(id)something {
  // bridge from Cocoa through our C++ backend
  _instanceVar = _backEndObject->DoSomethingPlatformSpecific();
  NSString* tempString = [NSString stringWithFormat:@"%d", _instanceVar];
  NSLog(@"%@", tempString);
}

@end
```

```
/** The platform-specific implementation of the C++ class, using C++ naming. */
int CrossPlatformAPI::DoSomethingPlatformSpecific() {
  NSString* temp_string = [NSString stringWithFormat:@"%d", an_instance_var_];
  NSLog(@"%@", temp_string);
  return [temp_string intValue];
}
```

Projects may opt to use an 80 column line length limit for consistency with Google's C++ style guide.

Spacing and Formatting

Spaces vs. Tabs

Use only spaces, and indent 2 spaces at a time. We use spaces for indentation. Do not use tabs in your code.

You should set your editor to emit spaces when you hit the tab key, and to trim trailing spaces on lines.

Line Length

The maximum line length for Objective-C files is 100 columns.

You can make violations easier to spot by enabling *Preferences > Text Editing > Page guide at column: 100* in Xcode.

Method Declarations and Definitions

One space should be used between the `-` or `+` and the return type, and no spacing in the parameter list except between parameters.

Methods should look like this:

```
// GOOD:

- (void)doSomethingWithString:(NSString *)theString {
  ...
}
```

The spacing before the asterisk is optional. When adding new code, be consistent with the surrounding file's style.

If a method declaration does not fit on a single line, put each parameter on its own line. All lines except the first should be indented at least four spaces. Colons before parameters should be aligned on all lines. If the colon before the parameter on the first line of a method declaration is positioned such that colon alignment would cause indentation on a subsequent line to be less than four spaces, then colon alignment is only required for all lines except the first.

```
// GOOD:

- (void)doSomethingWithFoo:(GTMFoo *)theFoo
                      rect:(NSRect)theRect
                  interval:(float)theInterval {
  ...
}

- (void)shortKeyword:(GTMFoo *)theFoo
         longerKeyword:(NSRect)theRect
    someEvenLongerKeyword:(float)theInterval
                    error:(NSError **)theError {
  ...
}

- (id<UIAdaptivePresentationControllerDelegate>)
    adaptivePresentationControllerDelegateForViewController:(UIViewController *)viewC

- (void)presentWithAdaptivePresentationControllerDelegate:
    (id<UIAdaptivePresentationControllerDelegate>)delegate;
```

Function Declarations and Definitions

Prefer putting the return type on the same line as the function name and append all parameters on the same line if they will fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.

```
// GOOD:

NSString *GTMVersionString(int majorVersion, minorVersion) {
  ...
}

void GTMSerializeDictionaryToFileOnDispatchQueue(
    NSDictionary<NSString *, NSString *> *dictionary,
    NSString *filename,
    dispatch_queue_t queue) {
```

```
  ...
}
```

Function declarations and definitions should also satisfy the following conditions:

- The opening parenthesis must always be on the same line as the function name.
- If you cannot fit the return type and the function name on a single line, break between them and do not indent the function name.
- There should never be a space before the opening parenthesis.
- There should never be a space between function parentheses and parameters.
- The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
- The close curly brace is either on the last line by itself or on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be aligned if possible.
- Function scopes should be indented 2 spaces.
- Wrapped parameters should have a 4 space indent.

Conditionals

Include a space after `if`, `while`, `for`, and `switch`, and around comparison operators.

```
// GOOD:

for (int i = 0; i < 5; ++i) {
}

while (test) {};
```

Braces may be omitted when a loop body or conditional statement fits on a single line.

```
// GOOD:

if (hasSillyName) LaughOutLoud();

for (int i = 0; i < 10; i++) {
  BlowTheHorn();
}
```

```
// AVOID:

if (hasSillyName)
  LaughOutLoud();               // AVOID.

for (int i = 0; i < 10; i++)
  BlowTheHorn();                // AVOID.
```

If an `if` clause has an `else` clause, both clauses should use braces.

```
// GOOD:

if (hasBaz) {
  foo();
} else {  // The else goes on the same line as the closing brace.
  bar();
}



// AVOID:

if (hasBaz) foo();
else bar();         // AVOID.

if (hasBaz) {
  foo();
} else bar();       // AVOID.
```

Intentional fall-through to the next case should be documented with a comment unless the case has no intervening code before the next case.

```
// GOOD:

switch (i) {
  case 1:
    ...
    break;
  case 2:
    j++;
    // Falls through.
  case 3: {
    int k;
    ...
    break;
  }
```

```
    case 4:
    case 5:
    case 6: break;
}
```

Expressions

Use a space around binary operators and assignments. Omit a space for a unary operator. Do not add spaces inside parentheses.

```
// GOOD:

x = 0;
v = w * x + y / z;
v = -y * (x + z);
```

Factors in an expression may omit spaces.

```
// GOOD:

v = w*x + y/z;
```

Method Invocations

Method invocations should be formatted much like method declarations.

When there's a choice of formatting styles, follow the convention already used in a given source file. Invocations should have all arguments on one line:

```
// GOOD:

[myObject doFooWith:arg1 name:arg2 error:arg3];
```

or have one argument per line, with colons aligned:

```
// GOOD:

[myObject doFooWith:arg1
               name:arg2
              error:arg3];
```

Don't use any of these styles:

```
// AVOID:

[myObject doFooWith:arg1 name:arg2  // some lines with >1 arg
             error:arg3];

[myObject doFooWith:arg1
              name:arg2 error:arg3];

[myObject doFooWith:arg1
          name:arg2  // aligning keywords instead of colons
          error:arg3];
```

As with declarations and definitions, when the first keyword is shorter than the others, indent the later lines by at least four spaces, maintaining colon alignment:

```
// GOOD:

[myObj short:arg1
          longKeyword:arg2
    evenLongerKeyword:arg3
                error:arg4];
```

Invocations containing multiple inlined blocks may have their parameter names left-aligned at a four space indent.

Function Calls

Function calls should include as many parameters as fit on each line, except where shorter lines are needed for clarity or documentation of the parameters.

Continuation lines for function parameters may be indented to align with the opening parenthesis, or may have a four-space indent.

```
// GOOD:

CFArrayRef array = CFArrayCreate(kCFAllocatorDefault, objects, numberOfObjects,
                                 &kCFTypeArrayCallBacks);

NSString *string = NSLocalizedStringWithDefaultValue(@"FEET", @"DistanceTable",
    resourceBundle,  @"%@ feet", @"Distance for multiple feet");

UpdateTally(scores[x] * y + bases[x],  // Score heuristic.
```

```
            x, y, z);

TransformImage(image,
               x1, x2, x3,
               y1, y2, y3,
               z1, z2, z3);
```

Use local variables with descriptive names to shorten function calls and reduce nesting of calls.

```
// GOOD:

double scoreHeuristic = scores[x] * y + bases[x];
UpdateTally(scoreHeuristic, x, y, z);
```

Exceptions

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:

@try {
  foo();
} @catch (NSException *ex) {
  bar(ex);
} @finally {
  baz();
}
```

Function Length

Prefer small and focused functions.

Long functions and methods are occasionally appropriate, so no hard limit is placed on function length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

When updating legacy code, consider also breaking long functions into smaller and more manageable pieces.

Vertical Whitespace

Use vertical whitespace sparingly.

To allow more code to be easily viewed on a screen, avoid putting blank lines just inside the braces of functions.

Limit blank lines to one or two between functions and between logical groups of code.

Objective-C Style Exceptions

Indicating style exceptions

Lines of code that are not expected to adhere to these style recommendations require `// NOLINT` at the end of the line or `// NOLINTNEXTLINE` at the end of the previous line. Sometimes it is required that parts of Objective-C code must ignore these style recommendations (for example code may be machine generated or code constructs are such that its not possible to style correctly).

A `// NOLINT` comment on that line or `// NOLINTNEXTLINE` on the previous line can be used to indicate to the reader that code is intentionally ignoring style guidelines. In addition these annotations can also be picked up by automated tools such as linters and handle code correctly. Note that there is a single space between `//` and `NOLINT*`.