Google Objective-C Style Guide

Revision 2.21

Mike Pinkerton
Greg Miller
Dave MacLachlan

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ▽ . You may toggle all summaries with the big arrow button:

▽ Toggle all summaries

Table of Contents

Important Note

Displaying Hidden Details in this Guide

link   ▽ This style guide contains many details that are initially hidden from view. They are marked by the triangle icon, which you see here on your left. Click it now. You should see "Hooray" appear below.

Hooray! Now you know you can expand points to get more details. Alternatively, there's an "expand all" at the top of this document.

Background

Objective-C is a very 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 new applications on Mac OS X and the iPhone.

Cocoa is one of the main application frameworks on Mac OS X. It is a collection of Objective-C classes that provide for rapid development of full-featured Mac OS X applications.

Apple has already written a very good, and widely accepted, coding guide for Objective-C. Google has also written a similar guide for C++. This Objective-C guide aims to be a very natural combination of Apple's and Google's general recommendations. So, before reading this guide, please make sure you've read:

- [Apple's Cocoa Coding Guidelines](#)
- [Google's Open Source C++ Style Guide](#)

Note that all things that are banned in Google's C++ guide are also banned in Objective-C++, unless explicitly noted in this document.

The purpose of this document is to describe the Objective-C (and Objective-C++) coding guidelines and practices that should be used for all Mac OS X code. Many of 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.

Google has already released open-source code that conforms to these guidelines as part of the [Google Toolbox for Mac project](#) (abbreviated GTM throughout this document). Code meant to be shared across different projects is a good candidate to be included in this repository.

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 [The Objective-C Programming Language](#).

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, etc.

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

```
//  GTMFoo.h
//  FooProject
//
//  Created by Greg Miller on 6/13/08.
//  Copyright 2008 Google, Inc. All rights reserved.
//

#import <Foundation/Foundation.h>

// A sample class demonstrating good Objective-C style. All interfaces,
// categories, and protocols (read: all top-level declarations in a header)
// MUST be commented. Comments must also be adjacent to the object they're
// documenting.
//
// (no blank line between this comment and the interface)
@interface GTMFoo : NSObject {
 @private
  NSString *foo_;
  NSString *bar_;
}

// Returns an autoreleased instance of GMFoo. See -initWithString: for details
// about the argument.
+ (id)fooWithString:(NSString *)string;
```

```
    // Designated initializer.  |string| will be copied and assigned to
    |foo_|.
    - (id)initWithString:(NSString *)string;

    // Gets and sets the string for |foo_|.
    - (NSString *)foo;
    - (void)setFoo:(NSString *)newFoo;

    // Does some work on |blah| and returns YES if the work was completed
    // successfuly, and NO otherwise.
    - (BOOL)doWorkWithString:(NSString *)blah;

    @end
```

An example source file, demonstating the correct commenting and spacing for the `@implementation` of an interface. It also includes the reference implementations for important methods like getters and setters, `init`, and `dealloc`.

```
    //
    //  GTMFoo.m
    //  FooProject
    //
    //  Created by Greg Miller on 6/13/08.
    //  Copyright 2008 Google, Inc. All rights reserved.
    //

    #import "GTMFoo.h"


    @implementation GTMFoo

    + (id)fooWithString:(NSString *)string {
      return [[[self alloc] initWithString:string] autorelease];
    }

    // Must always override super's designated initializer.
    - (id)init {
      return [self initWithString:nil];
    }

    - (id)initWithString:(NSString *)string {
      if ((self = [super init])) {
        foo_ = [string copy];
        bar_ = [[NSString alloc] initWithFormat:@"hi %d", 3];
      }
      return self;
    }

    - (void)dealloc {
      [foo_ release];
      [bar_ release];
      [super dealloc];
    }

    - (NSString *)foo {
      return foo_;
    }

    - (void)setFoo:(NSString *)newFoo {
      [foo_ autorelease];
      foo_ = [newFoo copy];
    }

    - (BOOL)doWorkWithString:(NSString *)blah {
      // ...
      return NO;
    }

    @end
```

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.

Line Length

▽ Each line of text in your code should be at most 80 characters long.

Even though Objective-C tends to be a more verbose language than C++, to aid in the interoperability with that guide, we have decided to keep the limit at 80 columns as well. It's easier to live with than you might expect.

We recognize that this rule is controversial, but so much existing code already adheres to it, and we feel that consistency is important.

You can make violations easier to spot in Xcode by going to *Xcode > Preferences > Text Editing > Show page guide*.

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:

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

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

If you have too many parameters to fit on one line, giving each its own line is preferred. If multiple lines are used, align each using the colon before the parameter.

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

When the first keyword is shorter than the others, indent the later lines by at least four spaces. You can do this by making keywords line up vertically, not aligning colons:

```
- (void)short:(GTMFoo *)theFoo
    longKeyword:(NSRect)theRect
    evenLongerKeyword:(float)theInterval {
  ...
}
```

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:

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

or have one argument per line, with colons aligned:

```
[myObject doFooWith:arg1
```

```
            name:arg2
           error:arg3];
```

Don't use any of these styles:

```
[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 keyword lengths make it impossible to align colons and still have four leading spaces, indent later lines by four spaces and align keywords after the first one, instead of aligning the colons.

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

@public and @private

▽ The `@public` and `@private` access modifiers should be indented by 1 space.

This is similar to `public`, `private`, and `protected` in C++.

```
@interface MyClass : NSObject {
 @public
  ...
 @private
  ...
}
@end
```

Exceptions

▽ Format exceptions with each @ label on its own line and a space between the @ label and the opening brace ({), as well as between the `@catch` and the caught object declaration.

If you must use Obj-C exceptions, format them as follows. However, see [Avoid Throwing Exceptions](#) for reasons why you should not be using exceptions.

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

Protocols

▽ There should not be a space between the type identifier and the name of the protocol encased in angle brackets.

This applies to class declarations, instance variables, and method delcarations. For example:

```
@interface MyProtocoledClass : NSObject<NSWindowDelegate> {
 @private
  id<MyFancyDelegate> delegate_;
}
```

```
        - (void)setDelegate:(id<MyFancyDelegate>)aDelegate;
        @end
```

Naming

Naming rules are very important in maintainable code. 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 comments unnecessary.

When writing pure Objective-C code, we mostly follow standard [Objective-C naming rules](). These naming guidelines may differ significantly from those outlined in the C++ style guide. For example, Google's C++ style guide recommends the use of underscores between words in variable names, whereas this guide recommends the use of intercaps, which is standard in the Objective-C community.

Any class, category, method, or variable name may use all capitals for [initialisms]() within the name. This follows Apple's standard of using all capitals within a name for initialisms such as URL, TIFF, and EXIF.

When writing Objective-C++, however, things are not so cut and dry. Many projects need to implement cross-platform C++ APIs with some Objective-C or Cocoa, or bridge between a C++ back-end and a native Cocoa front-end. This leads to situations where the two guides are directly at odds.

Our solution is that the style follows that of the method/function being implemented. If you're in an `@implementation` block, use the Objective-C naming rules. If you're implementing a method for a C++ `class`, use the C++ naming rules. This avoids the situation where instance variable and local variable naming rules are mixed within a single function, which would be a serious detriment to readability.

File Names

link

▽ 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:

`.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

File names for categories should include the name of the class being extended, e.g. `GTMNSString+Utils.h` or `GTMNSTextView+Autocomplete.h`.

Objective-C++

link

▽ Within a source file, Objective-C++ follows the style of the function/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. If you're in an `@implementation` block, use the Objective-C naming rules. If you're implementing a method for a C++ `class`, use the C++ naming rules.

```
// 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 stringWithInt:instanceVar_];
   NSLog(@"%@", tempString);
 }
 @end

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

Class Names

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

In *application-level* code, prefixes on class names should generally be avoided. Having every single
class with same prefix impairs readability for no benefit. When designing code to be shared across
multiple applications, prefixes are acceptable and recommended (e.g. `GTMSendMessage`).

Category Names

link ▽ Category names should start with a 2 or 3 character prefix identifying the category as part of a
project or open for general use. The category name should incorporate the name of the class
it's extending.

For example, if we want to create a category on `NSString` for parsing, we would put the category
in a file named `GTMNSString+Parsing.h`, and the category itself would be named
`GTMStringParsingAdditions` (yes, we know the file name and the category name do not
match, but this file could have many separate categories related to parsing). Methods in that
category should share the prefix (`gtm_myCategoryMethodOnAString:`) in order to prevent
collisions in Objective-C which only has a single namespace. If the code isn't meant to be shared
and/or doesn't run in a different address-space, the method naming isn't quite as important.

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

Objective-C Method Names

link ▽ Method names should start as lowercase and then use mixed case. Each named parameter
should also start as lowercase.

The method name should read like a sentence if possible, meaning you should choose parameter
names that flow with the method name. (e.g. `convertPoint:fromRect:` or

`replaceCharactersInRange:withString:`). See [Apple's Guide to Naming Methods](#) for more details.

Accessor methods should be named the same as the variable they're "getting", but they should *not* be prefixed with the word "get". For example:

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

```objectivec
- (id)delegate;     // GOOD
```

This is for Objective-C methods only. C++ method names and functions continue to follow the rules set in the C++ style guide.

Variable Names

▽ Variables names start with a lowercase and use mixed case to delimit words. Class member variables have trailing underscores. For example: *myLocalVariable*, *myInstanceVariable_*. Members used for KVO/KVC bindings may begin with a leading underscore *iff* use of Objective-C 2.0's `@property` isn't allowed.

Common Variable Names

Do *not* use Hungarian notation for syntactic attributes, such as the static type of a variable (int or pointer). Give as descriptive a name as possible, within reason. 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:

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

```objectivec
int numErrors;
int numCompletedConnections;
tickets = [[NSMutableArray alloc] init];
userInfo = [someObject object];
port = [network port];
```

Instance Variables

Instance variables are mixed case and should be suffixed with a trailing underscore, e.g. *usernameTextField_*. However, we permit an exception when binding to a member variable using KVO/KVC and Objective-C 2.0 cannot be used (due to OS release constraints). In this case, it is acceptable to prefix the variable with an underscore, per Apple's accepted practices for key/value naming. If Objective-C 2.0 can be used, `@property` and `@synthesize` provide a solution that conforms to the naming guidelines.

Constants

Constant names (#defines, enums, const local variables, etc.) should start with a lowercase *k* and then use mixed case to delimit words, i.e. *kInvalidHandle*, *kWritePerm*.

Comments

Though a pain to write, they are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very 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.

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!

Remember that all of the rules and conventions listed in the C++ Style Guide are in effect here, with a few additional points, below.

File Comments

▽ Start each file with a copyright notice, followed by a description of the contents of the file.

Legal Notice and Author Line

Every file should contain the following items, in order:

- a copyright statement (for example, `Copyright 2008 Google Inc.`)
- a license boilerplate. Choose the appropriate boilerplate for the license used by the project (for example, Apache 2.0, BSD, LGPL, GPL)

If you make significant changes to a file that someone else originally wrote, add yourself to the author line. This can be very helpful when another contributor has questions about the file and needs to know whom to contact about it.

Declaration Comments

▽ Every interface, category, and protocol declaration should have an accompanying comment describing its purpose and how it fits into the larger picture.

```
// A delegate for NSApplication to handle notifications about app
// launch and shutdown. Owned by the main app controller.
@interface MyAppDelegate : NSObject {
  ...
}
@end
```

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 in the public interface should have a comment explaining its function, arguments, return value, and any side effects.

Document the synchronization assumptions the class makes, 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.

Implementation Comments

▽ Use vertical bars to quote variable names and symbols in comments rather than quotes or naming the symbol inline.

This helps eliminate ambiguity, especially when the symbol is a common word that might make the sentence read like it was poorly constructed. E.g. for a symbol "count":

```
// Sometimes we need |count| to be less than zero.
```

or when quoting something which already contains quotes

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

Object Ownership

▽ Make the pointer ownership model as explicit as possible when it falls outside the most

common Objective-C usage idioms.

Instance variable pointers to objects derived from NSObject are presumed to be retained, and should be documented as weak if they are not retained by the class. However, instance variables which are labeled as IBOutlets are presumed to not be retained by the class, and should be documented as strong if the class does retain them.

Where instance variables are pointers to CoreFoundation, C++, and other non-Objective-C objects, they should always be documented in comments as strong or weak. Be mindful that support for automatic C++ objects encapsulated in Objective-C objects is disabled by default, as described [here](#).

Examples of strong and weak documentation:

```
@interface MyDelegate : NSObject {
 @private
  IBOutlet NSButton* okButton_;   // normal NSControl
  IBOutlet NSMenu* myContextMenu_;   // manually-loaded menu (strong)

  AnObjcObject* doohickey_;   // my doohickey
  MyController* controller_;   // so we can send msgs back (weak, owns
me)

  // non-NSObject pointers...
  CWackyCPPClass* wacky_;   // some cross-platform object (strong)
  CFDictionaryRef* dict_;   // (strong)
}
@end
```

strong
> The object will be `retain`'d by this class

weak
> The object will be not be `retain`'d by this class (e.g. a delegate).

Cocoa and Objective-C Features

Member Variables Should Be @private

link ▽ Member variables should be declared `@private`.

```
@interface MyClass : NSObject {
 @private
  id myInstanceVariable_;
}
// public accessors, setter takes ownership
- (id)myInstanceVariable;
- (void)setMyInstanceVariable:(id)theVar;
@end
```

Identify Designated Initializer

link ▽ Comment and 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 subclass a single initializer (of potentially several) to guarantee their subclass' initializer 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.

Override Designated Initializer

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

If you fail to override the superclass' designated initializer, your initializer may not be called in all

cases, leading to subtle and very difficult to find bugs.

Initialization

▽ Don't initialize variables to `0` or `nil` in the init method; it's redundant.

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

Avoid +new

▽ Do not invoke the `NSObject` class method `new`, nor override it in a subclass. Instead, use `alloc` and `init` methods to instantiate retained objects.

Modern Objective-C code explicitly calls `alloc` and an `init` method to create and retain an object. As the `new` class method is rarely used, it makes reviewing code for correct memory management more difficult.

Keep the Public API Simple

▽ Keep your class simple; avoid "kitchen-sink" APIs. If a method doesn't need to be public, don't make it so. Use a private category to prevent cluttering the public header.

Unlike C++, Objective-C doesn't have a way to differentiate between public and private methods — everything is public. 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. For internal implementation methods, use a category defined in the implementation file as opposed to adding them to the public header.

```
// GTMFoo.m
#import "GTMFoo.h"

@interface GTMFoo (PrivateDelegateHandling)
- (NSString *)doSomethingWithDelegate;   // Declare private method
@end

@implementation GTMFoo(PrivateDelegateHandling)
...
- (NSString *)doSomethingWithDelegate {
  // Implement this method
}
...
@end
```

Before Objective-C 2.0, if you declare a method in the private `@interface`, but forget to implement it in the main `@implementation`, the compiler will *not* object. (This is because you don't implement these private methods in a separate category.) The solution is to put the functions within an `@implementation` that specifies the category.

If you are using Objective-C 2.0, you should instead declare your private category using a [class extension](), for example:

```
@interface GMFoo () { ... }
```

which will guarantee that the declared methods are implemented in the `@implementation` section by issuing a compiler warning if they are not.

Again, "private" methods are not really private. You could 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.

Finally, Objective-C categories are a great way to segment a large `@implementation` section into

more understandable chunks and to add new, application-specific functionality to the most appropriate class. For example, instead of adding "middle truncation" code to a random object in your app, make a new category on `NSString`).

#import and #include

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

Choose between `#import` and `#include` based on the language of the header that you are including.

- When including a header that uses Objective-C or Objective-C++, use `#import`.
- When including a standard C or C++ header, use `#include`. The header should provide its own #define guard.

Some Objective-C headers lack `#define` guards, and expect to be included only by `#import`. As Objective-C headers may only be included in Objective-C source files and other Objective-C headers, using `#import` across the board is appropriate.

Standard C and C++ headers without any Objective-C in them can expect to be included by ordinary C and C++ files. Since there is no `#import` in standard C or C++, such files will be included by `#include` in those cases. Using `#include` for them in Objective-C source files as well means that these headers will always be included with the same semantics.

This rule helps avoid inadvertent errors in cross-platform projects. A Mac developer introducing a new C or C++ header might forget to add `#define` guards, which would not cause problems on the Mac if the new header were included with `#import`, but would break builds on other platforms where `#include` is used. Being consistent by using `#include` on all platforms means that compilation is more likely to succeed everywhere or fail everywhere, and avoids the frustration of files working only on some platforms.

```
#import <Cocoa/Cocoa.h>
#include <CoreFoundation/CoreFoundation.h>
#import "GTMFoo.h"
#include "base/basictypes.h"
```

Use Root Frameworks

▽ Include root frameworks over 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` rather than `#include` for Objective-C frameworks.

```
#import <Foundation/Foundation.h>       // good
```

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

Prefer To autorelease At Time of Creation

▽ When creating new temporary objects, `autorelease` them on the same line as you create them rather than a separate `release` later in the same method.

While ever so slightly slower, this prevents someone from accidentally removing the `release` or inserting a `return` before it and introducing a memory leak. E.g.:

```
// AVOID (unless you have a compelling performance reason)
MyController* controller = [[MyController alloc] init];
// ... code here that might return ...
```

```
  [controller release];
```

```
  // BETTER
  MyController* controller = [[[MyController alloc] init] autorelease];
```

Autorelease Then Retain

▽ Assignment of objects follows the `autorelease` then `retain` pattern.

When assigning a new object to a variable, one must first release the old object to avoid a memory
leak. There are several "correct" ways to handle this. We've chosen the "autorelease then retain"
approach because it's less prone to error. Be aware in tight loops it can fill up the autorelease pool,
and may be slightly less efficient, but we feel the tradeoffs are acceptable.

```
  - (void)setFoo:(GMFoo *)aFoo {
    [foo_ autorelease];   // Won't dealloc if |foo_| == |aFoo|
    foo_ = [aFoo retain];
  }
```

dealloc should process instance variables in declaration order

▽ `dealloc` should process instance variables in the same order the `@interface` declares them,
so it is easier for a reviewer to verify.

A code reviewer checking a new or revised `dealloc` implementation needs to make sure that every
retained instance variable gets released.

To simplify reviewing `dealloc`, order the code so that the retained instance variables get released
in the same order that they are declared in the `@interface`. If `dealloc` invokes other methods
that release instance variables, add comments describing what instance variables those methods
handle.

Setters copy NSStrings

▽ Setters taking an `NSString`, should always `copy` the string it accepts.

Never just `retain` the string. This avoids the caller changing it under you without your knowledge.
Don't assume that because you're accepting an `NSString` that it's not actually an
`NSMutableString`.

```
  - (void)setFoo:(NSString *)aFoo {
    [foo_ autorelease];
    foo_ = [aFoo copy];
  }
```

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.

Do not use the `NS_DURING`, `NS_HANDLER`, `NS_ENDHANDLER`, `NS_VALUERETURN` and
`NS_VOIDRETURN` macros unless you are writing code that needs to run on Mac OS X 10.2 or
before.

Also be aware when writing Objective-C++ code that stack based objects are not cleaned up when
you throw an Objective-C exception. Example:

```
class exceptiontest {
 public:
   exceptiontest() { NSLog(@"Created"); }
   ~exceptiontest() { NSLog(@"Destroyed"); }
};

void foo() {
   exceptiontest a;
   NSException *exception = [NSException exceptionWithName:@"foo"
                                                   reason:@"bar"
                                                 userInfo:nil];
   @throw exception;
}

int main(int argc, char *argv[]) {
   GMAutoreleasePool pool;
   @try {
     foo();
   }
   @catch(NSException *ex) {
     NSLog(@"exception raised");
   }
   return 0;
}
```

will give you:

```
2006-09-28 12:34:29.244 exceptiontest[23661] Created
2006-09-28 12:34:29.244 exceptiontest[23661] exception raised
```

Note that the destructor for *a* never got called. This is a major concern for stack based smartptrs such as `shared_ptr` and `linked_ptr`, as well as any STL objects that you may want to use. Therefore it pains us to say that if you must use exceptions in your Objective-C++ code, use C++ exceptions whenever possible. You should never re-throw an Objective-C exception, nor are stack based C++ objects (such as `std::string`, `std::vector` etc.) allowed in the body of any `@try`, `@catch`, or `@finally` blocks.

nil Checks

▽ Use `nil` checks for logic flow only.

Use `nil` checks for logic flow of the application, not for crash prevention. Sending a message to a `nil` object is handled by the Objective-C runtime. If the method has no return result, you're good to go. However if there is one, there may be differences based on runtime architecture, return size, and OS X version (see [Apple's documentation](#) for specifics).

Note that this is very different from checking C/C++ 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.

BOOL Pitfalls

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

`BOOL` is defined as an unsigned char in Objective-C which means that it can 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` which, depending on the value of the last byte of the integral result, could 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). You cannot safely interchange `BOOL` and `Boolean` so treat `Booleans` as a general integral value as discussed above. Only 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.

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

```
- (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, the first point above demonstrates that return values may not always be what you expect.

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

```
BOOL great = [foo isGreat];
if (great)
  // ...be great!
```

Properties

link

▽ Properties in general are allowed with the following caveat: properties are an Objective-C 2.0 feature which will limit your code to running on the iPhone and Mac OS X 10.5 (Leopard) and higher. Dot notation is allowed only for access to a declared @property.

Naming

A property's associated instance variable's name must conform to the trailing _ requirement. The property's name should be the same as its associated instance variable without the trailing _.

Use the @synthesize directive to rename the property correctly.

```
@interface MyClass : NSObject {
 @private
  NSString *name_;
}
@property(copy, nonatomic) NSString *name;
@end

@implementation MyClass
@synthesize name = name_;
@end
```

Location

A property's declaration must come immediately after the instance variable block of a class interface. A property's definition must come immediately after the @implementation block in a class definition. They are indented at the same level as the @interface or @implementation statements that they are enclosed in.

```
@interface MyClass : NSObject {
 @private
  NSString *name_;
}
@property(copy, nonatomic) NSString *name;
@end
```

```
@implementation MyClass
@synthesize name = name_;
- (id)init {
...
}
@end
```

Use Copy Attribute For Strings

NSString properties should always be declared with the `copy` attribute.

This logically follows from the requirement that setters for NSStrings always must use `copy` instead of `retain`.

Never synthesize CFType properties

CFTypes should always have the `@dynamic` implementation directive.

Since CFTypes can't have the `retain` property attribute, the developer must handle retaining and releasing the value themselves. In the rare case that you do actually want assignment it is better to make that completely clear by actually implementing the setter and getter and commenting why that is the case.

List out all implementation directives

Use implementation directives for all properties even if they are `@dynamic` by default.

Even though `@dynamic` is default, explicitly list it out with all of the other property implementation directives making it clear how every property in a class is handled at a single glance.

```
@interface MyClass : NSObject
@property(readonly) NSString *name;
@end

@implementation MyClass
.
.
.
- (NSString*)name {
  return @"foo";
}
@end
```

```
@interface MyClass : NSObject
@property(readonly) NSString *name;
@end

@implementation MyClass
@dynamic name;
.
.
.
- (NSString*)name {
  return @"foo";
}
@end
```

Atomicity

Be aware of the overhead of properties. By default, all synthesized setters and getters are atomic. This gives each set and get calls a substantial amount of synchronization overhead. Declare your properties `nonatomic` unless you require atomicity.

Dot notation

Dot notation is idiomatic style for Objective-C 2.0. It may be used when doing simple operations to get and set a `@property` of an object, but should not be used to invoke other object behavior.

```
NSString *oldName = myObject.name;
myObject.name = @"Alice";
```

```
NSArray *array = [[NSArray arrayWithObject:@"hello"] retain];

NSUInteger numberOfItems = array.count;    // not a property
array.release;                             // not a property
```

Cocoa Patterns

Delegate Pattern

link   ▽ Delegate objects should not be retained.

A class that implements the delegate pattern should:

1. Have an instance variable named *delegate_* to reference the delegate.
2. Thus, the accessor methods should be named `delegate` and `setDelegate:`.
3. The *delegate_* object should not be retained.

Model/View/Controller

link   ▽ Separate the model from the view. Separate the controller from the view and the model. Use
`@protocol`s for callback APIs.

- Separate model from view: don't build assumptions about the presentation into the model or data source. Keep the interface between the data source and the presentation abstract. Don't give the model knowledge of its view. (A good rule of thumb is to ask yourself if it's possible to have multiple presentations, with different states, on a single instance of your data source.)
- Separate controller from view and model: don't put all of the "business logic" into view-related classes; this makes the code very unusable. Make controller classes to host this code, but ensure that the controller classes don't make too many assumptions about the presentation.
- Define callback APIs with `@protocol`, using `@optional` if not all the methods are required. (Exception: when using Objective-C 1.0, `@optional` isn't available, so use a category to define an "informal protocol".)

Revision 2.21

Mike Pinkerton
Greg Miller
Dave MacLachlan