Objective-C and Objective-C++ Coding Standards

Contents

Synopsis

This document is for the Objective-C and Objective-C++ coding standards at Appcelerator. As with other coding standards documents, the primary goal is clean, readable code, which is comprable to common existing conventions.

Basis for this document

We attempt to follow the Google coding standards and Apple Cocoa guidelines. The remainder of this document reiterates information from these where appropriate, and otherwise provides exceptions to their standards.

We use "Modern Objective-C", the features of which are described in these documents.

- Programming with Objective-C
- Adopting Modern Objective-C
- Objective-C Feature Availability Index
- Transitioning to ARC Release Notes
- Blocks Programming Topics

Appcelerator C/C++ standards

You are expected to follow the C/C++ coding guidelines when writing Objective-C except where explicitly specified. These standards take precedence over any generic rules listed in the style guidelines above, although we have our own exceptions.

However, for consistency, any pure-C functions you write in Objective-C source files are to follow the Objective-C rules with C exceptions.

Standards

The following are the standard set of spacing, formatting, and naming conventions we expect for Objective-C(++) code.

`import` vs. `include`

Always @import (not `#import`) Objective-C headers, and `#include` C (or C++) headers.

Class Naming

- Objective-C classes are to be named with:
  - The prefix `Ti`, or another project-appropriate prefix
  - Camelcase

Example

```
@interface TiExampleClass : NSObject {
// ivars
}

// properties

// methods
```

The `@interface` directive should not be indented, and neither should `@property` or method declarations.

Protocols

Protocols follow the same naming conventions as classes, with the following exceptions:

- Protocols which reference a *behavior type* should end with a gerund (-ing).
- Protocols which describe a *set of actions* should describe the functional property of these collective actions.
- Protocols which are a *delegate* should end with the word `Delegate`.

Example

```
@protocol TiScrolling; // Gerund; behavior type is "this object scrolls"
@protocol TiFocusable; // Action set; describes actions related to "focusing" and
"TiFocusing" seems inappropraite ("this object focuses" vs. "this object performs
actions related to focusing")
@protocol TiScrollViewDelegate; // Delegate
```

Protocols must always include the `@required` directive explicitly.

Category naming

Header files which define an interface for a category only should be named `<base class>+<category>.h`.

- Categories on existing classes should be named appropriately, with the category describing the set of extensions.
- Categories which are intended to describe a private API within an implementation file should be the empty category `()`.

ivars

Prefer private properties to ivars. If you do have a valid use case for an ivar, then declare them in the @implementation block (not the @interface block).

Instance variables for a class should be intended one tabstop.

Instance variables should be named in camelcase, and are not required to follow any other specific naming convention.

`@public`, `@protected`, and `@private`

Use of access specifiers is discouraged (use publicly-declared and private-category `@property` instead).

`@property` and `@synthesize`

Use the default synthesis property of ivars. You should rarely need @synthesize.

Methods

- Methods should be named in camelcase, with the first character lowercase. Method names **must never** begin with an underscore.
- The leading method specifier (+ or -) should not be followed by a space, and neither should the return type.
- Selector (and argument) names should not have a space after their `:` character, or the type.
- If method declarations, definitions, or calls are spread across multiple lines, their `:` characters should be aligned rather than spaced on tabstops.
- The opening brace of a method should be on its own line for implementations.

Example

```
+(void)x:(int)y
{
}

-(void)veryLongMethodName:(NSObject*)veryLongArgumentName
                     arg2:(NSObject*)anotherArg
                     arg3:(NSObject*)moreArg
{
}
```

`init`

Every class must have one, and only one, designated initializer that is identified as such in a comment. The following is an example of well-written designated initializer:

Example

```
// Designated initializer.
-(instancetype)init
{
 self = [super init];
    if (self) {
        // initialization code goes here...
    }
    return self;
}
```

Note the single braces. You may wish to turn off the "initializer not fully bracketed" clang warning in Xcode as a result.

Blocks

- Block variables should never be a raw type; they should always have a `typedef`associated with them and that name used as the variable type.

    - **EXCEPTION:** The `void (^varname)(void)` block type does not require a typedef, although there are plenty of existing convenience typedefs for this block type which should be used when appropriate.
- Blocks should have their opening brace on the same line as their `^`, and their closing brace on its own line, indented with the surrounding scope.
- Blocks have their contents indented one tabstop from the surrounding scope.
- The `void ^(void)` block type should always be written as {{ ^{ ... } }}.
- `__block` storage specifier objects should be used with care. Remember that if a `__block` variable goes out of scope when a block tries to access it, there can be unpredictable and bad results.

Example

```
typedef int ^(intBlock)(int);

intBlock foo = ^(int foo) {
    return 2*foo;
};
```

Fast enumeration (`for x in y`)

Prefer fast enumeration loops to other looping constructs where possible. Note that if `y` is a method call, the result of it should be pre-cached. Do not write fast enumeration loops which would modify `y` (whether an object or a method call) as a side-effect of the loop contents.

File names

The following file names are acceptable for Objective-C:

- `.h` (headers)
- `.m` (implementation files)
- `.mm` (Objective-C++ - use with care, see below)
- `.pch` (precompiled header)

`@implementation` ordering

Methods should be ordered in `@implementation` in the following way:

- `@synthesize` directives
- Designated initializer(s), ending with `init`
- `#pragma mark Private` - Only required for implementations with a private category
- Methods declared in private category
- `#pragma mark Public` - Only required for implementations with a private category
- Methods declared in `@interface`
- `#pragma mark Protocol @protocol-name` - Only required for classes which implement a protocol
- Methods for `@protocol`, `@required` first, then `@optional`

The protocol implementation sections may be repeated as necessary.

`nil` and `NULL`

- Do not mix `nil` and `NULL`. `NULL` should only be used for C-style pointers, and `nil` for all Objective-C object (and `id`) types.
- It is illegal to use a statement such as {{ if (objcObject) { ... } }}. Instead directly compare to `nil`, **only where required**. Remember that it is actually faster to send a message to `nil` than to perform the `cmp/jmp` instructions from an `if` **and** make a method call. This is especially true on RISC architectures like ARM.

`BOOL` types

`BOOL` types should only be assigned to from `YES`, `NO`, or an explicit boolean operation. Do not mix `BOOL` types with C++ `boolean` or the C macros `TRUE` and `FALSE` - doing so may lead to subtle comparator errors for truth.

Exceptions to the C standard

There are a number of exceptions to the C standard, to make our Objective-C code more compatible with existing source and follow standard conventions. Any C code which is written within an Objective-C source file (`.m`) must also follow these conventions, for readability purposes.

Comments

- Classes, methods, and properties are to be documented as part of their `@interface`, not `@implementation`.
- Anything intended to be accessible through a public API of any kind should be tagged with comments suitable for appledoc generation; see appledoc for format info. You may wish to `brew install appledoc` as well.

Order of declarations

Rather than `namespace-contents`, the basic block for an Objective-C header is `objc-contents`:

- `@interface`
- `#pragma mark "@interface-name Properties"` (or "Properties" for headers with one `@interface`)
- `@property`
- `#pragma mark "@interface-name Methods"` (or "Methods" for headers with one `@interface`)
- Class methods
- Instance methods

The following is the order of declarations for an Objective-C or Objective-C++ header:

- Copyright notice
- `#import` headers (system, 3rd party, project)
- `#include` headers (system, 3rd party, project)
- macros
- `const` variables
- `enum`
- `typedef`
- `@protocol` declarations
- C-style function declarations
- `objc-contents`
- `namespace-contents` (for declared namespaces in Objective-C++ headers only)

Braces

Rather than spacing a brace on a newline in C, in Objective-C there are some cases in which an opening brace is placed on the same line as the preceding statement, with a space before it:

- Blocks (see above)
- Flow control (`if`/`while`/`for`/`do...while`/`switch...case`)

Variables

All variables are named in camel-case and should not contain punctuation.

Exceptions to the C++ standard

There are no exceptions to the C++ standard at this time.

Other Rules

3rd party libraries

As with all other source, the style in 3rd party libraries should be consistent with the style there rather than any Appcelerator coding standards. This holds true even for extensions we write to them.

Deprecated classes and methods

Avoid the usage of deprecated methods from standard frameworks where an alternative is available, unless it breaks backwards compatibility with a version of software that we support.

`@compatibility_alias`

Do not use the `@compatibility_alias` directive unless it is explicitly required due to a conflict between external libraries to a project, or multiple internal versions required by different 3rd party libraries.

`pragma mark`

Use `#pragma mark` liberally to annotate sections of your code where necessary, in addition to the rules spelled out above.