

Coding Conventions for iOS Objectiv-C

Prepared by:

Nakcheon Jung

Table of Contents

1	Introduction.....	- 4 -
2	General Naming Conventions and Special Rules	- 4 -
2.1	Language	- 4 -
2.2	Code Organization	- 4 -
2.3	Adopting Modern Objective-C	- 5 -
2.4	Boilerplate code	- 5 -
2.5	Spacing	- 7 -
2.6	Comments.....	- 9 -
2.7	Naming.....	- 9 -
2.7.1	Underscores at property	- 10 -
2.7.2	Underscores at method	- 10 -
2.7.3	Underscores for private method	- 11 -
2.7.1	Action and selector method	- 11 -
2.8	Declarations	- 11 -
2.9	Methods	- 13 -
2.10	Variables.....	- 13 -
2.11	Property Attributes	- 14 -
2.1	Class Property.....	- 15 -
2.2	Dot-Notation Syntax	- 15 -
2.3	Literals.....	- 15 -
2.4	Constants	- 16 -
2.5	Enumerated Types.....	- 16 -
2.6	Case Statements.....	- 17 -
2.7	Private Properties.....	- 18 -
2.8	Booleans	- 18 -
2.9	Conditionals	- 19 -
2.9.1	Ternary Operator	- 19 -
2.10	Init Methods	- 20 -
2.11	Class Constructor Methods.....	- 20 -
2.12	CGRect Functions.....	- 20 -

2.13	Golden Path	- 21 -
2.14	Error handling	- 21 -
2.15	Singletons	- 22 -
2.16	GCD Blocks	- 22 -
2.17	Smiley Face	- 22 -
3	Referecnes	- 23 -
3.1	The official raywenderlich.com Objective-C style guide.	- 23 -
3.2	Style guide & coding conventions for Objective-C projects.....	- 23 -

1 Introduction

The aims of this document is to describe about the coding conventions on the iOS develop environment. So that we could keep the code nice and consistent.

2 General Naming Conventions and Special Rules

2.1 Language

US English is preferred.

Preferred:

```
UIColor *myColor = [UIColor whiteColor];
```

Not Preferred:

```
UIColor *myColour = [UIColor whiteColor];
```

2.2 Code Organization

- Use #pragma mark - to categorize methods in functional groupings and protocol/delegate implementations following this general structure. And the order of categorized pragma mark sections is also important to find functions. It helps to use less search.

```
#pragma mark - Lifecycle
- (instancetype)init {}
- (void)dealloc {}
- (void)viewDidLoad {}
- (void)viewWillAppear:(BOOL)animated {}
- (void)didReceiveMemoryWarning {}

#pragma mark - NSCopying
- (id)copyWithZone:(NSZone *)zone {}

#pragma mark - NSObject
- (NSString *)description {}

#pragma mark - properties
- (void)setCustomProperty:(id)value {}
- (id)customProperty {}

#pragma mark - Public
- (void)publicMethod {}

#pragma mark - Private
```

```

- (void)privateMethod {}

#pragma mark - Actions

- (IBAction)submitData:(id)sender {}

#pragma mark - Custom Protocol conformance

#pragma mark - UITextFieldDelegate
#pragma mark - UITableViewDataSource
#pragma mark - UITableViewDelegate

```

2.3 Adopting Modern Objective-C

- Adapting Modern Objective-C is essential, please refer [the apple's document](#).
- After done programming please run the converting function provided by xCode. You can find the menu from 'xCode->Edit->Convert->To Modern Objective-C Syntax...'. For more information how the function works, please refer to [the link](#).

2.4 Boilerplate code

- Leaving code as comment for future use is not preferred. Especially when code organization code Introduced section [2.2](#) is copied and pasted, then unused '#pragma mark - Lifecycle' is commented the source is looks ugly.
- Do not leave code as comment.
- Do not leave unused 'pragma mark -' section as comment
- For better understanding of what is boilerplate code, please refer [the link](#).

Preferred:

```

#pragma mark - Lifecycle

- (instancetype)init
{
    // do something...
}

- (instancetype)initWithVehicleIndex:(NSInteger)index
{
    // do something...
}

- (void)dealloc
{
    // do something...
}

#pragma mark - Custom Accessor

- (NSInteger)getVehicleIndex
{
    // do something...
}

- (void)setVehicleIndex:(NSInteger)vehicleIndex
{
    // do something...
}

```

Not Preferred:

```

#pragma mark - Lifecycle
- (instancetype)init
{
    // do something...
}

// - (instancetype)initWithVehicleIndex:(NSInteger)index
//{
//    // do something...
//}

- (void)dealloc
{
    // do something...
}

#pragma mark - Custom Accessor
- (NSInteger)getVehicleIndex
{
    // do something...
}

- (void)setVehicleIndex:(NSInteger)vehicleIndex
{
    // do something...
}

// #pragma mark - Protocol conformance
// #pragma mark - UITextFieldDelegate
// #pragma mark - UITableViewDataSource
// #pragma mark - UITableViewDelegate
//
// #pragma mark - NSCopying
//
// - (id)copyWithZone:(NSZone *)zone {}
//
// #pragma mark - NSObject
//
// - (NSString *)description {}

```

- Do not leave issue number and fixed dated for fixing bugs. It does not help to improve code quality.

Preferred:

```

// When back to the foreground, if polling stauts was "P" resend 1 times
if (self.bOneMoreCheckRemoteCommandStatus) {
    // fixed codes are here...
}

```

Not Preferred:

```

// HDMC-578: When back to the foreground, if polling stauts was "P" resend 1 times: Owen 20160418
if (self.bOneMoreCheckRemoteCommandStatus) {
    // fixed codes are here...
}

```

- Do not write "break" after "return" in switch statements.

Preferred:

```
switch (rowIndex) {  
    case 0: // Change Title Table View  
        cellIdentifier = @"ChangeTitleTableViewCell";  
        break;  
    default:  
        cellIdentifier = @"ChangeTitleTableViewCell";  
        break;  
}
```

Not Preferred:

```
switch (rowIndex) {  
    case 0: // Change Title Table View  
        cellIdentifier = @"ChangeTitleTableViewCell";  
        return; // boiler plate code!  
        break;  
    default:  
        cellIdentifier = @"ChangeTitleTableViewCell";  
        break;  
}
```

2.5 Spacing

- Indent using 4 spaces (this conserves space in print and makes line wrapping less likely). Never indent with tabs. Be sure to use default settings in xCode.
- Method implementation always starts with new line.

Preferred:

```
- (void)doSomething  
{  
    // do something  
}
```

Not Preferred:

```
- (void)doSomething {  
    // do something  
}
```

- Method braces and other braces (if/else/switch/while etc.) always open on the same line as the statement but close on a new line. And 'else' keyword always starts with new line.
- Braces are should always be placed in conditional keywords (if/else/switch/while). Even if statement block consists of one line using braces is preferred.

Preferred:

```
if (user.isHappy) {  
    // do something  
}  
else {  
    // do something else  
}
```

Not Preferred:

```
if (user.isHappy)  
{  
    // do something  
}  
else {  
    // do something else  
}
```

- There should be exactly one blank line between methods to aid in visual clarity and organization. Whitespace within methods should separate functionality, but often there should probably be new methods.
- Prefer using auto-synthesis. But if necessary, `@synthesize` and `@dynamic` should each be declared on new lines in the implementation.
- Colon-aligning method invocation should often be used. There are cases where a method signature may have < 2 colons and not using colon-aligning makes the code more readable. Please do colon align methods containing blocks.
- For Colon-aligning method there's practical criterion, not using colon-aligning is better if block signature starts in 80 columns. If not Colon-aligning is better for reading.

Preferred:

```
// blocks are easily readable (block signature starts in 80 columns)  
[UIView animateWithDuration:1.0 animations:^(  
    // something  
} completion:^(BOOL finished) {  
    // something  
});
```

```
// Colon aligned: blocks are easily readable (block signature starts at over 80 columns)  
UIAlertAction *setENLngAction = [UIAlertAction actionWithTitle:[LanguageMgr sharedInstance] stringWithID:@"IDS_ENGLISH"  
                                                                    style:UIAlertActionStyleDefault  
                                                                    handler:^(UIAlertAction * action) {  
                                                                    // something  
}];
```

Not Preferred:

```
// colon-aligning makes the block hard to read  
[UIView animateWithDuration:1.0  
    animations:^(  
        // something  
    }  
    completion:^(BOOL finished) {  
        // something  
    }  
];
```


2.6 Comments

When they are needed, comments should be used to explain **why** a particular piece of code does something. Any comments that are used must be kept up-to-date or deleted. Please do not comment for future use codes because there are many cases code for future use are not used or mostly modified at last.

Block comments should generally be avoided, as code should be as self-documenting as possible, with only the need for intermittent, few-line explanations. **Exception: This does not apply to those comments used to generate documentation.**

2.7 Naming

- Apple naming conventions should be adhered to wherever possible, especially those related to [memory management rules \(NARC\)](#).
- Long, descriptive method and variable names are good.

Preferred:

```
UIButton *settingsButton;
```

Not Preferred:

```
UIButton *setBut;
```

- A letter prefix should always be used for class names and constants, however may be omitted for Core Data entity names. For any official iOS products such as books, starter kits, or tutorials, the prefix 'IX' for class and 'IX' for constants should be used.
- Enumeration constants naming should follow the rule described in here.
- Global type constants should have prefix "kIX". In-class global constants, global enumeration types are also included in here.
- Constants should be camel-case with all words capitalized and prefixed by the related class name for clarity.

Preferred:

```
static NSTimeInterval const kIXTutorialViewControllerNavigationFadeAnimationDuration = 0.3;
```

Not Preferred:

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

- Properties should be camel-case with the leading word being lowercase. Use auto-synthesis for properties rather than manual `@synthesize` statements unless you have good reason.

Preferred:

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

Not Preferred:

```
id varnm;
```

2.7.1 Underscores at property

When using properties, instance variables should always be accessed and mutated using `'self.'`. This means that all properties will be visually distinct, as they will all be prefaced with `'self.'`.

An exception to this: inside initializers, the backing instance variable (i.e. `_variableName`) should be used directly to avoid any potential side effects of the getters/setters.

Local variables should not contain underscores.

2.7.2 Underscores at method

When declare method only for use in classes as a private function, should use prefix `"_"`. This helps distinguish private methods on the implementation file among diverse kind of methods like, public, delegate, inherited etc. Also all private methods should be grouped altogether in `"#pragma mark - Private"` section.

Preferred:

```
#pragma mark - Private
- (BOOL)_calculateSomething
{
    // do something...
}

- (BOOL)_initializeSetting
{
    // do something...
}

- (BOOL)_initializeUi
{
    // do something...
}
```

Not Preferred:

```
#pragma mark - Private
- (BOOL)calculateSomething
{
    // do something...
}

- (BOOL)initializeSetting
{
    // do something...
}

- (BOOL)initializeUi
{
    // do something...
}
```

2.7.3 Underscores for private method

Should not use “get” and “set” prefix at any methods. Those are reserved for dynamic properties. If using “get/set” prefix is inevitable, design it as a property. Using another keyword is fine such as “calculate”, “adjust”, “move” or something verb.

Preferred:

- (BOOL)_calculateSomething
- (BOOL)_adjustLayout

Not Preferred:

- (BOOL)_getSomething
- (BOOL)_setLayout

2.7.1 Action and selector method

Use prefix “**action**” to distinguish other methods that have different usage. Also all private methods should be grouped altogether in “#pragma mark - Action” section.

Preferred:

- (IBAction)actionBackButtonPressed:(id)sender
- (void)actionKeyboardWasShown:(NSNotification *)notification;

Not Preferred:

- (IBAction)backButtonPressed:(id)sender
- (void)keyboardWasShown:(NSNotification *)notification;

2.8 Declarations

- Never declare an ivar unless you need to change its type from its declared property.

Preferred:

```
@interface IXTutorial : NSObject
@property (nonatomic, copy) NSString *tutorialName;
@end
```

Not Preferred:

```
@interface IXTutorial : NSObject {
    NSString *tutorialName;
}
```

- Prefer exposing an immutable type for a property if it being mutable is an implementation detail. This is a valid reason to declare an ivar for a property.

Preferred:

```
// in header file

@property (nonatomic, strong, nullable, getter=getVehicleStatus) NSDictionary* dicVehicleStatus;

// in .m file file

@interface IXInternalManager ()
@property (nonatomic, strong, nullable) NSMutableDictionary* dicVehicleStatusInfo;
@end

#pragma mark - Custom Accessors

@implementation IXInternalManager
- (NSDictionary *)getVehicleStatus
{
    return [NSDictionary dictionaryWithDictionary:self.dicVehicleStatusInfo];
}
@end
```

Not Preferred:

```
// in header file

-(void)setT:(NSString *)text i:(UIImage *)image;

@property (nonatomic, strong, nullable) NSMutableDictionary* dicVehicleStatus;
```

- Always declare memory-management semantics even on readonly properties. e.g., 'weak', 'strong', 'assign.
- Declare properties 'readonly' if they are only set once in -init.
- Don't use '@synthesize' unless the compiler requires it. Note that optional properties in protocols must be explicitly synthesized in order to exist.
- Declare properties 'copy' if they return immutable objects and aren't ever mutated in the implementation. strong should only be used when exposing a mutable object, or an object that does not conform to <NSCopying>
- Avoid 'weak' properties whenever possible. A long-lived weak reference is usually a code smell that should be refactored out.
- Don't put a space between an object type and the protocol it conforms to.

Preferred:

```
@property (attributes) id<Protocol> object;
@property (nonatomic, strong) NSObject<Protocol> *object;
```

Not Preferred:

```
@property (attributes) id <Protocol> object;
@property (nonatomic, strong) NSObject <Protocol> *object;
```

- C function declarations should have no space before the opening parenthesis, and should be namespaced just like a class.

Preferred:

```
void IxAwesomeFunction(BOOL hasSomeArgs);
```

Not Preferred:

```
void IxAwesomeFunction (BOOL hasSomeArgs);
```

2.9 Methods

In method signatures, there should be a space after the method type (-/+ symbol). There should be a space between the method segments (matching Apple's style). Always include a keyword and be descriptive with the word before the argument which describes the argument.

- The usage of the word "and" is reserved. It should not be used for multiple parameters as illustrated in the `initWithWidth:height:` example below.
- Do not use "and" and "with" at the signature name. By the way "with" can be used in method name.

Preferred:

```
- (void)setExampleText:(NSString *)text image:(UIImage *)image;
- (void)sendAction:(SEL)aSelector to:(id)anObject forAllCells:(BOOL)flag;
- (id)viewWithTag:(NSInteger)tag;
- (instancetype)initWithWidth:(CGFloat)width height:(CGFloat)height;
```

Not Preferred:

```
-(void)setT:(NSString *)text i:(UIImage *)image;
- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;
- (id>taggedView:(NSInteger)tag;
- (instancetype)initWithWidth:(CGFloat)width andHeight:(CGFloat)height;
- (instancetype)initWith:(int)width and:(int)height; // Never do this.
```

2.10 Variables

Variables should be named as descriptively as possible. Single letter variable names should be avoided except in `for()` loops.

Asterisks indicating pointers belong with the variable, e.g., `'NSString *text'` not `'NSString* text'` or `'NSString * text'`, except in the case of constants.

Private properties should be used in place of instance variables whenever possible. Although using instance variables is a valid way of doing things, by agreeing to prefer properties our code will be more consistent.

- Direct access to instance variables that 'back' properties should be avoided except in initializer methods (init, initWithCoder:, etc...), dealloc methods and within custom setters and getters. For more information on using Accessor Methods in Initializer Methods and dealloc, see [here \(Don't Use Accessor Methods in Initializer Methods and dealloc\)](#).
- **NSString** should have **copy** property attribute.

Preferred:

```
@property (nonatomic, copy, nullable) NSString *tutorialName;
```

Not Preferred:

```
@property (nonatomic, strong, nullable) NSString *tutorialName;
```

2.11 Property Attributes

Property attributes should be explicitly listed, and will help new programmers when reading the code.

- The order of property attributes is as follows: class, atomicity, memory management, nullability, getter and setter. And do not omit all attribute type. Not all programmers know which attributes is default.

Preferred:

```
@property (nonatomic, weak, nullable) IBOutlet UIView *containerView;  
@property (nonatomic, copy, nullable) NSString *tutorialName;
```

Not Preferred:

```
@property (nonatomic, weak) IBOutlet UIView *containerView;  
@property (nonatomic) NSString *tutorialName;
```

Properties with mutable counterparts (e.g. NSString) should prefer **copy** instead of **strong**. Why? Even if you declared a property as NSString somebody might pass in an instance of an NSMutableString and then change it without you noticing that.

Preferred:

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

Not Preferred:

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

2.1 Class Property

Class property is preferred rather than as method.

Preferred:

```
@property(class, nonatomic, readonly) UIApplication *sharedApplication
```

Not Preferred:

```
+ (UIApplication *)sharedApplication;
```

2.2 Dot-Notation Syntax

Dot syntax is purely a convenient wrapper around accessor method calls. When you use dot syntax, the property is still accessed or changed using getter and setter methods. Read more [here](#)

Dot-notation should always be used for accessing and mutating properties, as it makes code more concise. Bracket notation is preferred in all other instances.

Match with property and method definition. If callee class defines as property the caller should use dot notation not as method, for method definition should be called using "[]" method call notation.

Preferred:

```
NSInteger arrayCount = self.array.count;  
view.backgroundColor = UIColor.orangeColor;  
UIApplication.sharedApplication.delegate;
```

Not Preferred:

```
NSInteger arrayCount = [self.array count];  
[view setBackgroundColor:[UIColor orangeColor]];  
[UIApplication sharedApplication].delegate;
```

2.3 Literals

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

Preferred:

```
NSArray *names = @[@"Brian", @"Matt", @"Chris", @"Alex", @"Steve", @"Paul"];
```

```
NSDictionary *productManagers = @{@"iPhone": @"Kate", @"iPad": @"Kamal", @"Mobile Web": @"Bill"};
NSNumber *shouldUseLiterals = @YES;
NSNumber *buildingStreetNumber = @10018;
```

Not Preferred:

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

2.4 Constants

Constants are preferred over in-line string literals or numbers, as they allow for easy reproduction of commonly used variables and can be quickly changed without the need for find and replace. Constants should be declared as static constants and not #defines unless explicitly being used as a macro.

Preferred:

```
static NSString * const kIXAboutViewControllerCompanyName = @"RayWenderlich.com";

static CGFloat const kIXImageThumbnailHeight = 50.0;
```

Not Preferred:

```
#define CompanyName @"RayWenderlich.com"

#define thumbnailHeight 2
```

2.5 Enumerated Types

When using enums, it is recommended to use the new fixed underlying type specification because it has stronger type checking and code completion. The SDK now includes a macro to facilitate and encourage use of fixed underlying types: `NS_ENUM()`

For Example:

```
typedef NS_ENUM(NSInteger, IXLeftMenuTopItemType) {
    IXLeftMenuTopItemMain,
    IXLeftMenuTopItemShows,
    IXLeftMenuTopItemSchedule
};
```

You can also make explicit value assignments (showing older k-style constant definition):


```
typedef NS_ENUM(NSInteger, IXGlobalConstants) {
    IXPinSizeMin = 1,
    IXPinSizeMax = 5,
    IXPinCountMin = 100,
    IXPinCountMax = 500,
};
```

Older k-style constant definitions should be **avoided** unless writing CoreFoundation C code (unlikely).

Not Preferred:

```
enum GlobalConstants {
    kMaxPinSize = 5,
    kMaxPinCount = 500,
};
```

2.6 Case Statements

Braces are not required for case statements, unless enforced by the compiler.

When a case contains more than one line, braces should be added.

```
switch (condition) {
    case 1:
        // ...
        break;
    case 2:
    {
        // ...
        // Multi-line example using braces
    }
        break;
    case 3:
        // ...
        break;
    default:
        // ...
        break;
}
```

There are times when the same code can be used for multiple cases, and a fall-through should be used. A fall-through is the removal of the 'break' statement for a case thus allowing the flow of execution to pass to the next case value. A fall-through should be commented for coding clarity.

```
switch (condition) {
    case 1:
        // ** fall-through! **
    case 2:
        // code executed for values 1 and 2
        break;
    default:
        // ...
        break;
}
```

When using an enumerated type for a switch, 'default' is not needed. For example:

```
IXLeftMenuTopItemType menuType = IXLeftMenuTopItemMain;
```

```
switch (menuType) {  
    case IXLeftMenuTopItemMain:  
        // ...  
        break;  
    case IXLeftMenuTopItemShows:  
        // ...  
        break;  
    case IXLeftMenuTopItemSchedule:  
        // ...  
        break;  
}
```

Not Preferred:

```
switch (menuType) {  
    case IXLeftMenuTopItemMain:  
        // ...  
        break;  
    case IXLeftMenuTopItemShows:  
        // ...  
        break;  
    case IXLeftMenuTopItemSchedule:  
        // ...  
        break;  
    default:  
        // not need!  
}
```

2.7 Private Properties

Private properties should be declared in class extensions (anonymous categories) in the implementation file of a class. Named categories (such as

IXPrivate or private) should never be used unless extending another class. The Anonymous category can be shared/exposed for testing using the +Private.h file naming convention.

For Example:

```
@interface IXDetailViewController ()  
  
@property (nonatomic, strong) GADBannerView *googleAdView;  
@property (nonatomic, strong) ADBannerView *iAdView;  
@property (nonatomic, strong) UIWebView *adXWebView;  
  
@end
```

2.8 Booleans

Objective-C uses YES and NO. Therefore true and false should only be used for CoreFoundation, C or C++ code. Since nil resolves to NO it is unnecessary to compare it in conditions. Never compare something directly to YES, because YES is defined to 1 and a BOOL can be up to 8 bits.

This allows for more consistency across files and greater visual clarity. **Recommend only.**

Preferred:

```
if (someObject) {}  
if (![anotherObject boolValue]) {}
```

Not Preferred:

```
if (someObject == nil) {}  
if ([anotherObject boolValue] == NO) {}  
if (isAwesome == YES) {} // Never do this.  
if (isAwesome == true) {} // Never do this.  
if (isAwesome == false) {} // Never do this.
```

If the name of a BOOL property is expressed as an adjective, the property can omit the “is” prefix but specifies the conventional name for the get accessor, for example:

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

Text and example taken from the [Cocoa Naming Guidelines](#).

2.9 Conditionals

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

Preferred:

```
if (!error) {  
    return success;  
}
```

Not Preferred:

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

or

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

or

```
if (!error)  
{  
    return success;  
}
```

2.9.1 Ternary Operator

The Ternary operator, `?:`, should only be used when it increases clarity or code neatness. A single condition is usually all that should be evaluated. Evaluating multiple conditions is usually more understandable as an `if` statement, or refactored into instance variables. In general, the best use of the ternary operator is during assignment of a variable and deciding which value to use.

Non-boolean variables should be compared against something, and parentheses are added for improved readability. If the variable being compared is a boolean type, then no parentheses are needed.

Preferred:

```
NSInteger value = 5;
result = (value != 0) ? x : y;

BOOL isHorizontal = YES;
result = isHorizontal ? x : y;
```

Not Preferred:

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

2.10 Init Methods

Init methods implementation body brace should start with new line. A return type of 'instancetype' should also be used instead of 'id'.

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

See [Class Constructor Methods](#) for link to article oninstancetype.

2.11 Class Constructor Methods

Where class constructor methods are used, these should always return type of 'instancetype' and never 'id'. This ensures the compiler correctly infers the result type.

```
@interface Airplane
+ (instancetype)airplaneWithType:(IXAirplaneType)type;
@end
```

More information oninstancetype can be found on NSHipster.com.

2.12 CGRect Functions

When accessing the x, y, width, or height of a CGRect, always use the [CGRectGeometry functions](#) instead of direct struct member access. From Apple's CGRectGeometry reference:

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

Preferred:

```
CGRect frame = self.view.frame;

CGFloat x = CGRectGetMinX(frame);
CGFloat y = CGRectGetMinY(frame);
CGFloat width = CGRectGetWidth(frame);
CGFloat height = CGRectGetHeight(frame);
CGRect frame = CGRectMake(0.0, 0.0, width, height);
```

Not Preferred:

```
CGRect frame = self.view.frame;

CGFloat x = frame.origin.x;
CGFloat y = frame.origin.y;
CGFloat width = frame.size.width;
CGFloat height = frame.size.height;
CGRect frame = (CGRect){ .origin = CGPointZero, .size = frame.size };
```

2.13 Golden Path

When coding with conditionals, the left hand margin of the code should be the "golden" or "happy" path. That is, don't nest if statements. Multiple return statements are OK.

Preferred:

```
- (void)someMethod
{
    if (![someOther boolValue]) {
        return;
    }

    //Do something important
}
```

Not Preferred:

```
- (void)someMethod
{
    if ([someOther boolValue]) {
        //Do something important
    }
}
```

2.14 Error handling

When methods return an error parameter by reference, switch on the returned value, not the error variable.

Preferred:

```

NSError *error;
if (![self trySomethingWithError:&error]) {
    // Handle Error
}

```

Not Preferred:

```

NSError *error;
[self trySomethingWithError:&error];
if (error) {
    // Handle Error
}

```

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

2.15 Singletons

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

```

+ (instancetype)sharedInstance
{
    static id sharedInstance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}

```

This will prevent [possible and sometimes prolific crashes](#).

2.16 GCD Blocks

GCD blocks should use a retain cycle safe pattern.

```

__weak SomeObjectClass *weakSelf = self;

SomeBlockType someBlock = ^{
    SomeObjectClass *strongSelf = weakSelf;
    if (!strongSelf) {
        // The original self doesn't exist anymore.
        // Ignore, notify or otherwise handle this case.
        return;
    }
    [strongSelf someMethod];
};

```

This will prevent possible and sometimes prolific crashes and retain cycle, see [here](#)

2.17 Smiley Face

Smiley faces are a very prominent style feature of the raywenderlich.com site! It is very important to have the correct smile signifying the immense amount of happiness and excitement for the coding topic. The end square bracket is used because it represents the largest smile able to be captured using ascii art. A half-hearted smile is represented if an end parenthesis is used, and thus not preferred.

Preferred:

:]

Not Preferred:

:), ^^

3 Referecnes

3.1 The official raywenderlich.com Objective-C style guide.

<https://github.com/raywenderlich/objective-c-style-guide>

3.2 Style guide & coding conventions for Objective-C projects

<https://github.com/github/objective-c-style-guide>