The official raywenderlich.com Objective-C style guide.

This style guide outlines the coding conventions for raywenderlich.com.

Introduction

The reason we made this style guide was so that we could keep the code in our books, tutorials, and starter kits nice and consistent - even though we have many different authors working on the books.

This style guide is different from other Objective-C style guides you may see, because the focus is centered on readability for print and the web. Many of the decisions were made with an eye toward conserving space for print, easy legibility, and tutorial writing.

Credits

The creation of this style guide was a collaborative effort from various raywenderlich.com team members under the direction of Nicholas Waynik. The team includes: Soheil Moayedi Azarpour, Ricardo Rendon Cepeda, Tony Dahbura, Colin Eberhardt, Matt Galloway, Greg Heo, Matthijs Hollemans, Christopher LaPollo, Saul Mora, Andy Pereira, Mic Pringle, Pietro Rea, Cesare Rocchi, Marin Todorov, Nicholas Waynik, and Ray Wenderlich

We would like to thank the creators of the New York Times and Robots & Pencils' Objective-C Style Guides. These two style guides provided a solid starting point for this guide to be created and based upon.

Background

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

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

Table of Contents

Language

US English should be used.

Preferred:

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

Not Preferred:

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

Code Organization

Use `#pragma mark -` to categorize methods in functional groupings and protocol/delegate implementations following this general structure.

```
#pragma mark - Lifecycle

- (instancetype)init {}
```

```
- (void)dealloc {}
- (void)viewDidLoad {}
- (void)viewWillAppear:(BOOL)animated {}
- (void)didReceiveMemoryWarning {}

#pragma mark - Custom Accessors

- (void)setCustomProperty:(id)value {}
- (id)customProperty {}

#pragma mark - IBActions

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

#pragma mark - Public

- (void)publicMethod {}

#pragma mark - Private

- (void)privateMethod {}

#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 {}
```

Spacing

- Indent using 2 spaces (this conserves space in print and makes line wrapping less likely). Never indent with tabs. Be sure to set this preference in Xcode.
- 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.

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 avoided. There are cases where a method signature may have >= 3 colons and colon-aligning makes the code more readable. Please do **NOT** however colon align methods containing blocks because Xcode's indenting makes it illegible.

Preferred:

```
// blocks are easily readable
[UIView animateWithDuration:1.0 animations:^{
  // something
} completion:^(BOOL finished) {
  // something
}];
```

Not Preferred:

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

Comments

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

Block comments 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.*

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 three letter prefix should always be used for class names and constants, however may be omitted for Core Data entity names. For any official raywenderlich.com books, starter kits, or tutorials, the prefix 'RWT' should be used.

Constants should be camel-case with all words capitalized and prefixed by the related class name for clarity.

Preferred:

```
static NSTimeInterval const
RWTTutorialViewControllerNavigationFadeAnimationDuration = 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 (strong, nonatomic) NSString *descriptiveVariableName;
```

Not Preferred:

```
id varnm;
```

Underscores

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.

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.

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.
```

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.

Preferred:

```
@interface RWTTutorial : NSObject

@property (strong, nonatomic) NSString *tutorialName;

@end
```

Not Preferred:

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

Property Attributes

Property attributes should be explicitly listed, and will help new programmers when reading the code. The order of properties should be storage then atomicity, which is consistent with automatically generated code when connecting UI elements from Interface Builder.

Preferred:

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

Not Preferred:

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

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.

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;
```

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 numberWithInteger:10018];
```

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 `#define`s unless explicitly being used as a macro.

Preferred:

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

static CGFloat const RWTImageThumbnailHeight = 50.0;
```

Not Preferred:

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

#define thumbnailHeight 2
```

Enumerated Types

When using `enum`s, 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, RWTLeftMenuTopItemType) {
  RWTLeftMenuTopItemMain,
  RWTLeftMenuTopItemShows,
  RWTLeftMenuTopItemSchedule
};
```

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

```
typedef NS_ENUM(NSInteger, RWTGlobalConstants) {
  RWTPinSizeMin = 1,
  RWTPinSizeMax = 5,
  RWTPinCountMin = 100,
  RWTPinCountMax = 500,
};
```

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

Not Preferred:

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

Case Statements

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

```
RWTLeftMenuTopItemType menuType = RWTLeftMenuTopItemMain;
```

```
switch (menuType) {
  case RWTLeftMenuTopItemMain:
    // ...
    break;
  case RWTLeftMenuTopItemShows:
    // ...
    break;
  case RWTLeftMenuTopItemSchedule:
    // ...
    break;
}
```

Private Properties

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

@property (strong, nonatomic) GADBannerView *googleAdView;
@property (strong, nonatomic) ADBannerView *iAdView;
@property (strong, nonatomic) UIWebView *adXWebView;

@end
```

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.

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 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.

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;
```

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;
```

Init Methods

Init methods should follow the convention provided by Apple's generated code template. 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 on instancetype.

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:(RWTAirplaneType)type;
@end
```

More information on instancetype can be found on NSHipster.com.

CGRect Functions

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

> All functions described in this reference that take CGRect data structures as inputs

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

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 };
```

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
  }
}
```

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).

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.

Line Breaks

Line breaks are an important topic since this style guide is focused for print and online readability.

For example:

```
self.productsRequest = [[SKProductsRequest alloc]
initWithProductIdentifiers:productIdentifiers];
```

A long line of code like this should be carried on to the second line adhering to this style guide's Spacing section (two spaces).

```
self.productsRequest = [[SKProductsRequest alloc]
  initWithProductIdentifiers:productIdentifiers];
```

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:

```
:)
```

Xcode project

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

When possible, always turn on "Treat Warnings as Errors" in the target's Build Settings and enable as many additional warnings as possible. If you need to ignore a specific warning, use Clang's pragma feature.

Other Objective-C Style Guides

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

- Robots & Pencils
- New York Times
- Google

- GitHub
- Adium
- Sam Soffes
- CocoaDevCentral
- Luke Redpath
- Marcus Zarra