Dotsquares™

2017

iOS Coding Guideline

VERSION 1.0

DOTSQUARES

India Office: Dotsquares Ltd.: 6-kha-9, Jawahar Nagar, Jaipur. Rajasthan, India - 302004
UK Office: Unit 2 Albourne Court, Henfield Road, Albourne, West Sussex, UK
USA Office: 14781 Memorial Dr., Suite # 1956, Houston, TX 77079, USA
www.dotsquares.com

Contents

Objective

The objective of this document is to summarize **Code Guidelines** for iOS developers to view and follow the basic checklist while writing code.

This document provides general guidelines for both developer & reviewer and coding standards to follow while writing the code.

General Guideline for developers

1. Architecture

- Follow MVC architecture and prefer Swift language over Objective C.
- Lightweight your controller and heavy your services, Avoid writing excessive code in controller, instead perform the business logic and data operation work in service classes
- Avoid using multiple if condition in a code block, instead divide that in multiple functions.
- Avoid adding more stress on classes by assigning more responsibilities on them, this means that every class in your code should only have one job to do and everything in this class should be related with single purpose.
- Always try to design your classes in such a way that the new functionality can be added only when new requirements are generated and it should not be altered until you find any bugs into it.
- Follow DRY principle means write once and runs everywhere
- Keep code simple and straight
- Avoid using 3rd party libraries to prevent data stealing & upgrade issues.

2. Documentation

- All methods should be commented in a clear language- as if it is not clear to the reader, it will also unclear to the user.
- All source code should contain @author for all the authors.
- @version should be included as required.
- All class, variable and method modifiers should be examined for correctness.
- Complex algorithms should be explained with references.
  For example- document the reference that identifies the equation, formula, or pattern. In all cases, examine the algorithm and determine if it can be simplified.
- Code that depends on non-obvious behaviour in external frameworks is documented with reference to external documentation.
- Units of measurement are documented for numeric values.
- Incomplete code is marked with //TODO or //FIXME markers.

3. Unit Testing

- Unit tests must cover error conditions and invalid parameter cases.
- Check for possible null pointers are always checked before use.
- Array indices are always checked to avoid ArrayIndexOfBounds exceptions.
- Ensure that the code fixes the issue, or implements the requirement, and that the unit test confirms it. If the unit test confirms a fix for issue, add the issue number to the documentation.

4. Error Handling

- Invalid parameter values are handled properly early in methods (Fast Fail).
- NullPointerException conditions from method invocations are checked.
- Consider using a general error handler to handle known error conditions.
- Avoid using RuntimeException, or sub-classes to avoid making code changes to implement correct error handling.
- Don't pass the buck! Don't create classes which throw Exception rather than dealing with exception condition.

5. Thread Safety

- Use proper methods like - delegates/protocol to avoid circular dependency.
- Locks must be acquired and released in the right order to prevent deadlocks, even in error handling code.
- Use structs over classes.

6. Performance

- Objects are duplicated only when necessary. If you must duplicate objects, consider implementing Clone and decide if deep cloning is necessary.
- No busy-wait loops instead of proper thread synchronisation methods. For example, avoid `while(true){ ... sleep(10);...}`
- Avoid large objects in memory, or using String to hold large documents which should be handled with better tools. For example, don't read a large XML document into a String, or DOM.
- Do not leave debugging code in production code.

7. Warning, Memory Leaks & App thinning

- Developers need to take care about the warning in the application, if they found any warning then it should be corrected.
- Developers should check the application once in a week with instrument tools such as energy log, leaks, zombies etc. to avoid memory leaks.
- To reduce the app size, avoid using custom fonts, use vector/pdf images where possible, use XIB instead of large size splash PNG image, remove i386 architecture, use ".wav" files for audio and enable BITCODE while publishing the app on app store.

Coding Standards and Best practices

1. Class Names

Class names are always capitalised.

Objective-C doesn't have namespaces, so prefix your class names with initials. This avoids "namespace collision," which is a situation where two pieces of code have the same name but do different things. Classes created by Cocoa Dev Central should be prefixed with "CDC".

If you subclass a standard Cocoa class, it's good idea to combine your prefix with the superclass name, such as CDCTableView.

2. Variable Names

Variable names start with lower-case letters, but are internally capitalised wherever a new word appears:

Correct

```
//Objective C
NSString * streetAddress  = @"Raja Park";
NSString * cityName       = @"Jaipur";
NSString * countyName     = @"India";
NSString * hostName;
NSNumber * ipAddress;
NSArray  * accounts;

//SWIFT
var streetAddress: String = "Raja Park"
var cityName: String = "Jaipur"
var countyName: String = "India"
var hostName: String = ""
var ipAddress: NSNumber?
var accounts = [Any]()
```

Incorrect

```
//Objective C
NSString       * HST_NM;      // all caps and too terse
NSNumber       * theip;       // a word or abbreviation?
NSMutableArray * nsma;        // completely ambiguous

//SWIFT
var HST_NM: String = "" // all caps and too terse
var theip: NSNumber? // a word or abbreviation?
var nsma = [Any]() // completely ambiguous
```

Variables cannot start with a number, no spaces, and no special characters other than underscores.

Apple discourages using an underscore as a prefix for a private instance variable.

```
//Objective C
NSString * name;// correct!
NSString * _name; // _incorrect_

//SWIFT
var name: String = "" // correct!
var _name: String = "" // _incorrect_
```

Variable Names: Indicating Type

In terms of real-world practice, a variable name usually does not indicate the type if it is something common like NSString, NSArray, NSNumber or BOOL.

Correct

```
//Objective C
NSString       * accountName;
NSMutableArray * mailboxes;
NSArray        * defaultHeaders;
BOOL userInputWasUpdated;

//Swift
```

```
var accountName: String = ""
var mailboxes = [Any]()
var defaultHeaders = [Any]()
var userInputWasUpdated: Bool = false
```

OK, But Not Ideal

```
//Objective C
NSString     * accountNameString;
NSMutableArray * mailboxArray;
NSArray        * defaultHeadersArray;
BOOL userInputWasUpdatedBOOL;


//Swift
var accountNameString: String = ""
var mailboxArray = [Any]()
var defaultHeadersArray = [Any]()
var userInputWasUpdatedBOOL: Bool = false
```

If a variable is not one of these types, the name should reflect it. In addition, there are certain classes that you only need one instance of them. In this case, just name the variable based on the class name. File manager is a good example of this.

When to Indicate Type

```
//Objective C
UIImage *previewPaneImage;  // self-explanatory
UIProgressView *uploadIndicator;   // shows progress for uploads
NSFileManager *fileManager; // only one of these, basic name ok

//Swift
var previewPaneImage: UIImage? // self-explanatory
var uploadIndicator: UIProgressView? // shows progress for uploads
var fileManager: FileManager? // only one of these, basic name ok
```

3. Method Names

Methods are perhaps the most important topic we can talk about and most object-oriented languages use syntax.

While these methods names are easy to write the first time, the actual behaviour is not clear. This is much more of a problem amidst massive amounts of surrounding code.

Cocoa programmers think from the end, choosing a method name based on how it will look in actual use. Let's say I want to write an in-memory file object written to disk.

Example:

```
OBJECTIVE C

-(void)saveObjectForKey:(NSString *)key withValue:(NSString *)value{
    //Do something with key & value
}
[self saveObjectForKey:@"userInfo" withValue:@"DS"];

SWIFT

func saveObjectFor(key:String, withValue:String){
  //Do something with key & value
}
self.saveObjectFor(key: "userInfo", withValue: "DS")
```

Method Names: Accessor

In contrast to many other languages, Objective-C discourages use of the "get" prefix on simple accessor. Instance variables and methods can have the same name, so use this to your advantage:

Correct

```
//Objective C
-(NSString *) name;
-(UIColor *) color;
name = [object name];
color = [object color];

//SWIFT
func name() -> String;
func color() -> UIColor;
name = object.name()
color = object.name()
```

Incorrect

```
//Objective C
-(NSString *) getName;
-(UIColor *) getColor;
name = [object getName];
color = [object getColor];

//Swift
func getName () -> String;
func getColor () -> UIColor;
name = object.getName ()
color = object.getColor ()
```

The "get" prefix is, however, used in situations when you're returning a value indirectly via a memory address.

4. Code Organization

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

Example:

```
// Objective c
#pragma mark - IBActions
 -(IBAction)submitData:(id)sender {}
#pragma mark - Public
 -(void)publicMethod {}

// Swift

// MARK: - IBActions
 @IBAction func submitData(_ sender:Any?)
 {}
// MARK: - Public
 @IBAction func publicMethod(){}
```

5. Spacing

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
 }

//Not Preferred:
 if (!error)
    return success;
```

6. Error Handling

When methods return an error parameter by reference, code MUST switch on the return value and MUST NOT switch on the error variable.

Example:

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

//Not Preferred:
NSError *error;
[self trySomethingWithError:&error];
if (error) {
  // Handle Error
}
```

In Swift, use do-catch statement to handle errors by running a block of code. If an error is thrown by the code in the do clause then it is matched against the catch clauses to determine which one of them can handle the error.

```
//Syntax
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}

//Example
do {
     let data = try Data(contentsOf: url)
} catch let error {
     print(error)
}
```

7. Comments

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. This is compulsory to define comments over the classes, methods and boolean variables.

The mechanism for a single line comment is marked by prefixing the line with // (double-slash).

Example:

```
// This is a comment line. It is for human use only and is ignored by the Objective-C/Swift compiler.
```

The start and end of lines of comments are marked by the /* and */ markers respectively. Everything immediately after the /* and before the */ is considered to be a comment, regardless of where the markers appear on a line.

Example:

```
/* Commented out July 20 while testing improved version */
```

8. Booleans

Values MUST NOT be compared directly to YES, because YES is defined as 1, and a BOOL in Objective-C is a CHAR type that is 8 bits long (so a value of 11111110 will return NO if compared to YES).

```
//Preferred: For an object pointer:
if (!someObject) {
}
if (someObject == nil) {
}
//For a BOOL value:
if (!someObject) { }
if (!someNumber.boolValue){ }
if (someNumber.boolValue == NO){ }

//Not Preferred:
if (isAwesome == YES) //Never do this.
```

9. Singleton

Singleton objects SHOULD use a thread-safe pattern for creating their shared instance. This will prevent possible and sometimes frequent crashes.

```
// Objective C
+(instancetype)sharedInstance {
      static id sharedInstance = nil; static dispatch_once_tonceToken;
      dispatch_once(&onceToken, ^{
         sharedInstance = [[[self class] alloc] init];
      });
      return sharedInstance;
   }

//Swift
class UploadManager: NSObject {
   static let shared: UploadManager = { UploadManager() } ()
}
```

10. Imports

If there is more than one import statement, statements MUST be grouped together. Groups MAY be commented.

Note: For modules use the @import syntax.

```
OBJECTIVE C

// Frameworks
@import QuartzCore;
// Models
#import "UserModel.h"
// Views
#import "DSButton.h"

SWIFT

// Frameworks
import QuartzCore;
```

11. Protocols

In a delegate or data source protocol, the first parameter to each method should be the object sending the message.

This helps disambiguate in cases when an object is the delegate for multiple similarly typed objects, and it helps clarify intent to readers of a class implementing these delegate methods.

Example:

```
//Objective C

// Preferred:
-(void)tableView:(UITableView*)tableView didSelectRowAtIndexPath: (NSIndexPath*)indexPath;

//Not Preferred:
-(void)didSelectTableRowAtIndexPath:(NSIndexPath*)indexPath;


//Swift
// Preferred:
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)

//Not Preferred:
func didSelectRowAt(_ indexPath: IndexPath)
```

12. Data Storage

iOS provide various way to store data locally, like .plist, UserDefaults, Database and as per the need these can be used to store data.

1. NSUserDefaults or UserDefaults provides a programmatic interface to store key-value pairs persistently across invocations of app on a given device. It have certain methods, which return data with particular type without manually casting it.

With this you can save objects from the following class types:

- NSData / Data

- NSString / String
- NSNumber / Number
- NSDate / Date
- NSArray / Array
- NSDictionary / Dictionary

Example:

```
// Save data in defaults
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
[defaults setObject:@"Dotsquares" forKey:@"name"];
[defaults synchronize];
// read data from defaults
[defaults objectForKey:@"name"]; //@"Dotsquares"

// delete data from defaults
[defaults removeObjectForKey:@"name"];
[defaults synchronize];


// Swift 3
// Save data in defaults
let defaults:UserDefaults = UserDefaults.standard
defaults.setValue("Dotsquares", forKeyPath: "name")
defaults.synchronize()

// read data from defaults
defaults.object(forKey: "name")//@"Dotsquares"

// delete data from defaults
defaults.removeObject(forKey: "name")
defaults.synchronize()

//Access particular type value from defaults
defaults.string(forKey: key)
defaults.integer(forKey: key)
```

2. plist, a property list is a structured data representation used by Cocoa and Core Foundation as a convenient way to store, organize, and access standard types of data. Property lists are based on an abstraction for expressing simple hierarchies of data. The items of data in a property list are of a limited number of types. Some types are for primitive values and others are for containers of values.

The primitive types are strings, numbers, binary data, dates, and Boolean values. The containers are arrays—indexed collections of values—and dictionaries—collections of values each identified by a key. The containers can contain other containers as well as the primitive types. Thus, you might have an array of dictionaries, and each dictionary might contain other arrays and dictionaries, as well as the primitive types. A root property-list object is at the top of this hierarchy, and in almost all cases is a dictionary or an array.

3. Database: We can use SQLite & Core Data for save complex & relational data.

a. **SQLite:** SQLite is the most used database engine in the world and its open source as well. It implements a transactional SQL database engine with no configuration and no server required. The reasons for the great popularity of SQLite are its
   - Independence from a server
   - Zero-configuration
   - Safe access from multiple processes and threads

- Stores data in tables with one or more columns that contain a specific type of data

b. **Core Data:** Core Data is a framework that store or retrieve data in database in an object-oriented way. Core Data is actually backed by SQLite database & it is a wrapper around SQLite. It focuses more on objects than the traditional table database methods. With Core Data, you are actually storing contents of an object, which is represented by a class in Objective-C. Although they are fundamentally different, Core data
   - Uses more memory than SQLite
   - Uses more storage space than SQLite
   - Faster in fetching records than SQLite

Sqlite v/s Core Data

Sqlite	Core Data
Primary function is storing and fetching data	Primary function is graph management (although reading and writing to disk is an important supporting feature)
Operates on data stored on disk (or minimally and incrementally loaded)	Operates on objects stored in memory (although they can be lazily loaded from disk)
Stores "dumb" data	Works with fully-fledged objects that self-manage a lot of their behaviour and can be sub-classed and customized for further behaviours
Can be transactional, thread-safe, multi-user	Non-transactional, single threaded, single user (unless you create an entire abstraction around Core Data which provides these things)
Can drop tables and edit data without loading into memory	Only operates in memory
Perpetually saved to disk (and often crash resilient)	Requires a save process
Can be slow to create millions of new rows	Can create millions of new objects in-memory very quickly (although saving these objects will be slow)
Offers data constraints like "unique" keys	Leaves data constraints to the business logic side of the program

13. Background Thread

If the operations, we're performing take long time or too many resources then it will leads to very poor user experience. These kind of operation we need to perform in background thread by using the main UI thread not blocked & it will give better user experience of the app.

Once we done with background thread & we have to update the UI then again we need to move back to main UI thread. Below is the example of downloading an image from an URL & show the downloaded image in to the image view.

Example:

```
//Downloading an image from the URL
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0),
^(void) {
        NSURL *url = [[NSURL alloc] initWithString:@"http://www.dotsquares.com/assets/wp-
content/themes/dotsquares/images/dotsquares-web-design.png"];
        NSData *data0 = [NSData dataWithContentsOfURL:url];
        UIImage *image = [UIImage imageWithData:data0];
        //Updating the image view after downloading the image
        dispatch_sync(dispatch_get_main_queue(), ^(void) {
            imageView.image = image;
        });
    });

// In swift 3 Downloading an image from the URL
DispatchQueue.global(qos: .background).async {
            guard let url = URL(string: "http://www.dotsquares.com/assets/wp-
content/themes/dotsquares/images/dotsquares-web-design.png") else{
                return    }
            do
            {
                let data = try Data(contentsOf: url)
                let image = UIImage(data:data)
              //Updating the image view after downloading the image
                DispatchQueue.main.async {
                    imageView.image = image;
                }
            } catch {
                print(error)
            }
        }
```

14. User Filter (Swift Only)

In swift, filter method helps to search collection and returns an array that contains elements that meet a condition. This is a good alternate of doing the same with for loop iteration.

```
// return an array of videos those are not uploaded yet
let unUploadVideos = videos.filter { (video) -> Bool in
    video.uploadStatus == 0;
}
```