Contents

Visual Appearance

Hardly any software is maintained for its whole life by the original author. Writing code in unified way helps engineers to understand it more quickly.

Class names are not prefixed usually

Prefixes are used only for classes that are to be shared between applications.

Protocol names indicates behavior

Most protocols group related methods that aren't associated with any class in particular. This type of protocol should be named so that the protocol won't be confused with a class. A common convention is to use a gerund ("...ing") form:

- NSLocking Good
- NSLock Poor (seems like a name for a class)

Some protocols group a number of unrelated methods (rather than create several separate small protocols). These protocols tend to be associated with a class that is the principal expression of the protocol. In these cases, the convention is to give the protocol the same name as the class. An example of this sort of protocol is the NSObject protocol.

Header file follows a structure

Header should be kept as simple as possible. Declarations appears in it in the following order:

1. Imports
2. Forward class declarations
3. Enumeration types
4. Constants
5. Notification names and its user info dictionary keys
6. Delegate protocol
7. Data source protocol
8. Class interface

The class interface has a fixed structure as well:

1. Properties
2. Data source property
3. Delegate property
4. Class methods
5. Instance methods

Please pay attention to number of separating lines, it is also a rule.

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

@class ExampleClass;


typedef NS_ENUM(NSInteger, Enumeration) {
    EnumerationInvalid,
    EnumerationA,
};

extern CGFloat const ExampleClassDefaultHeight;

extern NSString * const ExampleClassWillPerformActionNotification;
extern NSString * const ExampleClassDidPerformActionNotification;
extern NSString * const ExampleClassActionNameKey;


@protocol ExampleClassDelegate <NSObject>

- (void)exampleClass:(ExampleClass *)exampleClass didPerformAction:(Action *)action;

@optional

- (void)exampleClass:(ExampleClass *)exampleClass willPerformAction:(Action *)action;

@end


@protocol ExampleClassDataSource <NSObject>

- (NSInteger)exampleClassNumberOfActions:(ExampleClass *)exampleClass;

@end


@interface ExampleClass : NSObject

@property (strong, nonatomic) NSURL * initialProperty;
@property (weak, nonatomic) id<ExampleClassDataSource> dataSource;
@property (weak, nonatomic) id<ExampleClassDelegate> delegate;

+ (id)exampleClassWithInitialProperty:(NSURL *)initialProperty;
- (id)initWithInitialProperty:(NSURL *)initialProperty;
- (void)performAction;

@end
```

Implementation file is divided by pragma marks

Methods in implementation file appear in the same order they are declared in the header. They are grouped by pragma marks similar to following:

```
#pragma mark - Public Properties
#pragma mark - Public Class Methods
#pragma mark - Public Instance Methods
#pragma mark - IBActions
#pragma mark - Overridden
#pragma mark - Private Properties
#pragma mark - Private Class Methods
#pragma mark - Private Instance Methods
#pragma mark - Protocols
#pragma mark - Notifications
```

When greater granularity is needed:

```
#pragma mark - Overridden (UIView)
#pragma mark - Overridden (UIContainerViewControllerCallbacks)
#pragma mark - Overridden (UIViewControllerRotation)
```

Tip: Create a code snipped to help you use the same pragma marks through all implementation files.

Property attributes are kept in order

Attributes are kept in the same order through all property declarations.

```
[assign | weak | strong | copy] + [nonatomic | atomic] + [readonly | readwrite] + [getter = ]
```

None of the attributes can be omitted with the exception of readwrite.

```
@property (assign, nonatomic) CGFloat height;
@property (strong, nonatomic) UIColor * color;
@property (copy, nonatomic) NSString * name;
@property (weak, nonatomic) id <UITableViewDelegate> delegate;
```

Getters for boolean properties should be renamed as follows.

```
@property (assign, nonatomic, getter = isVisible) BOOL visible;
```

Protocols and constants are prefixed with class name

Protocols, notification names, enumeration types and other constants that refer to particular class are prefixed with the name of that class.

```
typedef NS_ENUM(NSInteger, UITableViewStyle) {
    UITableViewStylePlain,
    UITableViewStyleGrouped
};

UIKIT_EXTERN NSString *const UITableViewIndexSearch;

UIKIT_EXTERN const CGFloat UITableViewAutomaticDimension;

@protocol UITableViewDelegate<NSObject, UIScrollViewDelegate>
...
```

IBOutlets are declared privately

Outlets are defined as weak properties at the top of class extension, divided by one line from other properties.

```
@interface PanelViewController ()

@property (weak, nonatomic) IBOutlet UIButton * infoButton;
@property (weak, nonatomic) IBOutlet UIButton * closeButton;
@property (weak, nonatomic) IBOutlet UILabel * descriptionLabel;

@property (weak, nonatomic) UIView * overlayView;

@end
```

Classes may have additional header file for subclasses

Private methods and properties are never exposed. To provide subclass access to them, they have to be declared in ForSubclassEyesOnly category. It should be placed in separate header file, named in the following manner: [Class Name] + Subclass.h.

> **Tip:** UIGestureRecognizerSubclass.h is a good example of that approach.

Methods return early, return often

Nesting makes code harder to read.

```
- (BOOL)loginUser:(NSString *)user withPassword:(NSString *)password
{
    if (user.length >= 6) {
        if ((password.length >= 8) {
            // actual logging code
        }
    }
    return NO;
}
```

Getting invalid cases out of he way first will keep the working code with one level of indentation, and ensure that all parameters are valid. This paradigm is called The Golden Path.

```
- (BOOL)loginUser:(NSString *)user withPassword:(NSString *)password
{
    if (user.length < 6) return NO;
    if (password.length < 8) return NO;

    // actual logging code
}
```

This approach has one more advantage, it is easier to add error handling later on.

```
- (BOOL)loginUser:(NSString *)user withPassword:(NSString *)password error:(NSError **)error
{
    if (user.length < 6) {
        if (error != NULL)
            *error = [NSError errorWithDomain:ExampleDomain
                                         code:1001
                                     userInfo:@{NSLocalizedDescriptionKey: @"User name must have..."}];
        return NO;
    }

    if (password.length < 8) {
        if (error != NULL)
            *error = [NSError errorWithDomain:ExampleDomain
                                         code:1002
                                     userInfo:@{NSLocalizedDescriptionKey: @"Password must have..."}];
        return NO;
    }

    // logging in code
}
```

Method are not bisected with conditionals.

Following method structure is not acceptable.

```
- (void)method
{
    if (self.valueX == 10) {
        // perform some actions
    }
    else {
        // perform some other actions
    }
}
```

Bisection can be removed by returning in the if statement.

```
- (void)method
{
    if (self.valueX == 10) {
        // perform some actions
        return;
    }

    // perform some other actions
}
```

In one case bisection is allowed.

```
- (void)method
{
    if (self.valueX == 10) {
        // perform some actions
    }
    else {
        // perform some other actions
    }

    // perform some actions no matter what
}
```