OBJECTIVE-C BEST PRACTICES

IN A TEAM ENVIRONMENT

by Rolin Nelson

Presented at JaxMUG March 2013

GOALS

- Introduce / review Objective-C core features
- Review recent additions to Objective-C
- Discuss and propose Objective-C Styling that will lead to better collaboration among team

TOPICS

- Objective-C History
- Objective-C Primer and Recent Features
- Objective-C Styling Best Practices
- Xcode Configuration
- Summary

INTRO

- Rolin Nelson
- Software Engineer with 23 years of overall experience
- Performed Middleware integration prior to mobile
- 5 years of mobile experience
- Developed native and hybrid iOS apps
- Developed in a large environment

HISTORY OF OBJECTIVE-C

- Created by Brad Cox and Tom love in the early1980s
- Backward compatible to C
- In 1988, the NeXT company licensed Objective-C
- NeXT developed the AppKit (NSTextField, NSView, NS...)
- And the Foundation Kit (NSObject, NSString, etc.) libraries
- The GCC compiler was used to compile

HISTORY OF OBJECTIVE-C

- Licensing limitations did not allow sharing of enhancements
- A GNU project was started to work on a free implementation
- The new implementation of Cocoa(GNUStep) was based on the OpenStep standard
- Shortly after acquiring NeXT (in1996), Apple used OpenStep in its new Mac OS.

HISTORY OF OBJECTIVE-C

- In 2007, in Mac OS X 10.5 a 2.0 version of Objective-C was released
- Version 2.0 contain many breakthrough enhancements

VERSION 2.0

- Garbage Collection (Only on Mac OS X)
- Syntax Enhancements
- Runtime Performance Improvements
- 64-Bit Support

VERSION 2.0

- Properties
- Dot Syntax
- Fast Enumeration
- Optional Protocol

OBJECTIVE-C PRIMER

SYNTAX

- Objective-C is a thin layer on top of C
- It is a strict superset of C
- You can compile any C program with an Objective-C compiler
- Objective-C derives it object-oriented syntax from the Smalltalk language
- All other syntax are identical to C

OBJECTIVE-C PRIMER
MESSAGE SENDING

- [obj method:argument];
- The Objective-C model of object-oriented is based on message passing to object instances
- In Objective-C one doesn’t simply call a method; one sends a message
- This allows the method called to be resolved at runtime (aka. Dynamic Typing)

OBJECTIVE-C PRIMER
MESSAGE SENDING

- The side effect is message-passing has no type checking
- When a message is sent to an object, an exception will be thrown if the object does not implement a method that respond

OBJECTIVE-C PRIMER
INTERFACE IMPLEMENTATION

- Objective-C requires the interface and implementation of class be in separate code blocks
- By convention, the interface is placed in a header file with a .h suffix
- The implementation is placed in a code file with .m suffix
- Objective-C++, the suffix is .mm

OBJECTIVE-C PRIMER
INTERFACE IMPLEMENTATION

Example of class interface file

```
@interface className : superClassName {
 // instance variables
}
 // class methods
+ classMethod1;
+ (return_type)classMethod2;
+ (return_type)classMethod3:(param1_type)param1_varName;

// instance methods
- (return_type)instanceMethod1:(param1_type)param1_varName :(param2_type)param2_varName;
- (return_type)instanceMethod2WithParameter:(param1_type)param1_varName param2_callName:
(param2_type)param2_varName;
@end
```

OBJECTIVE-C PRIMER

INTERFACE IMPLEMENTATION

Example of class implementation file

```
@implementation className
+ (return_type)classMethod
{
 // implementation
}
- (return_type)instanceMethod
{
 // implementation
}
@end
```

OBJECTIVE-C PRIMER
OBJECT INSTANTIATION

- Objective-C objects can be created by allocating an instance and then initializing it

- Both steps are required for the object to be fully functional

OBJECTIVE-C PRIMER
OBJECT INSTANTIATION

Example of instantiation with the default, no-parameter initializer

```
MyObject *o = [[MyObject alloc] init];
```

OBJECTIVE-C PRIMER
OBJECTIVE-C PRIMER

OBJECT INSTANTIATION

```
Example of instantiation with a customer initializer

MyObject *o = [[MyObject alloc] initWithString:myString];
```

OBJECTIVE-C PRIMER
OBJECT INSTANTIATION

```
Example of instantiation with the default, no-parameter initializer (short form)

MyObject *o = [MyObject new];
```

OBJECTIVE-C PRIMER

OBJECT INSTANTIATION

Example of default initializer in implementation file

```
- (id)init {
    self = [super init];
    if (self) {
        // perform initialization of object here
    }
    return self;
}
```

Example of custom initializer in implementation file

```
- (id)initWithString:(NSString *)aString {
    self = [super init];
    if (self) {
        // perform initialization of object here
        self.title = aString;
    }
    return self;
}
```

OBJECTIVE-C PRIMER
PROTOCOL

- Multiple inheritance of specification
- Very similar to an interface in Java and C#
- Protocols can include both instance/class methods
- Protocols can include properties
- Informal/Formal Protocol

OBJECTIVE-C PRIMER

PROTOCOL

Example of protocol declaration

```
@protocol ABCDataSource <NSObject>
- (NSUInteger)numberOfSegments;
- (NSString *)titleForSegmentAtIndex:(NSUInter)segmentIndex;
@end
```

Example of protocol adoption

```
@interface MyClass : NSObject <ABCDataSource>
...
@end

@interface MyClass : NSObject <ABCDataSource, AnotherProtocol>
...
@end
```

OBJECTIVE-C PRIMER
DYNAMIC TYPING

- An object can be sent a message that is not specified in its interface
- This flexible allows an object to capture/forward messages
- This pattern is known as message forwarding or delegation

OBJECTIVE-C PRIMER
DYNAMIC TYPING

Examples of dynamic typing

```
Foo may be of any class.
- (void)setMyValue:(id)foo;


Foo may be an instance of any class that conforms to the NSCopying protocol.
- (void)setMyValue:(id<NSCopying>)foo;


Foo must be an instance of the NSNumber class.
- (void)setMyValue:(NSNumber *)foo;


Foo must be an instance of the NSNumber class and must conform to the
NSCopying protocol
- (void)setMyValue:(NSNumber<NSCopying> *)foo;
```

OBJECTIVE-C PRIMER
DYNAMIC TYPING

```
@interface MyClass : NSObject <SomeProtocol>
...

- (void)setMyValue:(id)foo
{
    if ([foo isKindOfClass:[NSNumber class]]) {
        // handle
    }
    else if ([foo isKindOfClass:[NSString class]]) {
        // handle
    }
}

...

@end
```

OBJECTIVE-C PRIMER
FORWARDING

- Objective-C permits the sending of any message to an object
- A runtime error ***`doesNotRecognizeSelector:`*** will be generated when an object receives an unrecognized message
- Objects may override the default behavior by implementing the ***`forwardInvocation:`*** method in NSObject.

OBJECTIVE-C PRIMER

FORWARDING

```
@implementation MyClass : NSObject <SomeProtocol>
...

- (void)forwardInvocation:(NSInvocation)invocation
{
    SEL aSelector = [invocation selector]
    if ([otherObject respondsToSelector:aSelector]) {
        [invocation invokeWithTarget:otherObject];
    }
    else {
        [super forwardInvocation:invocation];
    }
}

...

@end
```

OBJECTIVE-C PRIMER
CATEGORIES

- Objective-C was designed with the ability to maintain large code base
- Categories allow large code base to be broken into smaller pieces, known as Categories
- The methods within a category are added to a class at run-time

OBJECTIVE-C PRIMER
CATEGORIES

- Categories permit the programmer to add methods to an exiting class
- The added methods are indistinguishable from the existing methods
- The added methods have full access to instance and private variables
- Categories can even override existing methods. (Bug fixes)

OBJECTIVE-C PRIMER
CATEGORIES

Example of category declaration

```
...

@interface ExistingClass (MyAdditions)
- (NSUInteger)nameOfNewMethodAdded;
@end
```

Example of category implementation

```
@implementation ExistingClass (MyAdditions)
...

- (NSUInteger)nameOfNewMethodAdded {
    NSUInteger someValue
    ...
    return someValue + existingValue;
}

@end
```

OBJECTIVE-C PRIMER
GARBAGE COLLECTION

- Garbage Collection was added to Objective-C in Mac OS X 10.5 (Leopard)
- It was deprecated in Mac OS X 10.8 in favor of ARC (Automatic Reference Counting)
- It never existed in iOS, due to the performance

OBJECTIVE-C PRIMER
PROPERTIES

- Properties are public instance variables
- Properties may be defined with storage modifiers
- In non-ARC environment, modifiers (assign, copy or retain)
- In ARC environment, modifiers (weak or strong) instead of retain

OBJECTIVE-C PRIMER
PROPERTIES

- Additionally, properties may declare "readonly"
- Properties may be declared with "nonatomic"
- Nonatomic removes the wrapping lock used to access the variable value (faster without lock)
- The lock does not guarantee ordering (only fully set or read values)
- Default access is "atomic"

OBJECTIVE-C PRIMER

PROPERTIES

```
@interface Person : NSObject {

@public
    NSString *name;
@private
    int age;
}

@property(copy) NSString *name;
@property(readonly) int age;

- (id)initWithAge:(int)age;

@end
```

OBJECTIVE-C PRIMER

PROPERTIES

```
@implementation Person
@synthesize name=_name;

- (id)initWithAge:(int)initAge {
    self = [super init];
    if (self) {
        age = initAge; // NOTE: direct instance variable assignment, not property setter
    }
    return self;
}

- (int)age {
    return age;
}
@end
```

OBJECTIVE-C PRIMER
PROPERTIES AUTOSYNTHESIS

- When using Xcode 4.4 or newer with clang 3.1 (Apple LLVM compiler 4.0)
- Properties are implicitly synthesized unless explicitly declared

```
...

@property(copy) NSString *name;
...

@implementation Person
@synthesize name=_name; // This line is no longer necessary
...

@end
```

OBJECTIVE-C PRIMER
PROPERTIES AUTOSYNTHESIS

- Auto synthesis is not performed for properties defined in a protocol

```
...

@protocol someProtocol <NSObject>
@property (nonatomic, strong) NSString *name;
@end
...
```

OBJECTIVE-C PRIMER
PROPERTIES

- The @dynamic keyword may be used to delay the addition of the setter/getter or autosynthesis
- Backing instance variables are created by the property variables without ever being declared in the class interface
- Very useful from private properties

OBJECTIVE-C PRIMER
FAST ENUMERATION

- Instead of using the NSEnumeration class or indexes use fast enumeration
- Fast enumeration provides better performance
- It does pointer arithmetic to traverse a collection

OBJECTIVE-C PRIMER
FAST ENUMERATION

```
...

// Using NSEnumerator
NSEnumerator *enumerator = [thePeople objectEnumerator];
Person *p;

while ((p = [enumerator nextObject]) != nil) {
 NSLog(@"%@ is %i years old.", [p name], [p age]);
}

// Using indexes
for (int i = 0; i < [thePeople count]; i++) {
 Person *p = [thePeople objectAtIndex:i];
 NSLog(@"%@ is %i years old.", [p name], [p age]);
}

// Using fast enumeration
for (Person *p in thePeople) {
 NSLog(@"%@ is %i years old.", [p name], [p age]);
}

...
```

LATEST OBJECTIVE-C FEATURES

FEATURES

- ARC - Automatic Reference Counting
- Literal Object Creation
- Subscripting Collections

LATEST OBJECTIVE-C FEATURES

ARC - AUTOMATIC REFERENCE COUNTING

- Code to maintain reference counts are inserted in the appropriate places during compilation time
- More efficient that garbage collection since a separate thread is not required to manage the retain counts.

LATEST OBJECTIVE-C FEATURES
LITERAL OBJECT CREATION

- Previously, only string objects could be created literally
- `NSString *aString = @"This is a new string";`
- Now, arrays, dictionaries and numbers can be created
- `NSArray *anArray = @[anObject];`
- `NSDictionary *aDictionary = @{@"key":anObject};`
- `NSNumber *aNumber = @(anInt);`

LATEST OBJECTIVE-C FEATURES
SUBSCRIPTING COLLECTIONS

```
...

// Example without subscripting:

id object1 = [someArray objectAtIndex:0];
id object2 = [someDictionary objectForKey:@"key"];
[someMutableArray replaceObjectAtIndex:0 withObject:object3];
[someMutableDictionary setObject:object4 forKey:@"key"];

// Example with subscripting:

id object1 = someArray[0];
id object2 = someDictionary[@"key"];
someMutableArray[0] = object3;
someMutableDictionary[@"key"] = object4;

...
```

OBJECTIVE-C STYLING
GOALS

- NOT to present an ideal style representation
- A style that will make it easy to refactor
- Other members are able to modify each others code
- Make it easy for team members to code review others code
- Team members should agree and compromise on styles
- A plan and document should be made going forward

OBJECTIVE-C STYLING
SKEPTICAL

- Most programmers respond very negatively to this idea
- They believe this is a complete waste of time
- They are too busy
- Besides, who will pay for the additional cost?
- Code already exist, and it is not broken, why should I rework?

OBJECTIVE-C STYLING

WHY

- In a large organization where multiple developer may modify the same source file to code different fixes, style is very important
- Existing sources files should be lazily updated
- That is, source should only be brought to standard if modified during code fix
- If some sort of code merging(manual or auto) procedure is used, it will be simpler if the style was standardized
- In the long run, this will save cost with cleaner code

OBJECTIVE-C STYLING

SPACING

- When a keyword is preceded by an opening parenthesis, there should be a space between the keyword and parenthesis.
- When a non-keyword is preceded by an opening parenthesis, there should NOT be a space between the keyword and parenthesis.

Recommended	Not Recommended
`if (YES)`	`if(YES)`
`myFunc(`	`myFunc (`

OBJECTIVE-C STYLING
SPACING

- Use 2 newlines between major definitions (i.e. classes, protocols).
- Use a newline between code paragraphs (i.e. methods).
- Use 4 spaces instead of tabs for indentation.
- No whitespace after colons and before expressions for method parameters.
- Place a single space after // for comments (Not required for commenting out code).

OBJECTIVE-C STYLING
SPACING

- No space before comma, however, leave one space after a comma.
- For pointer types, always put a space between the type and the asterisk.
- When type casting, put a space after the closing parenthesis.

```
    NSString *foo = @"Hello";
    NSString *abc = (NSString *) [obj func];}
```

OBJECTIVE-C STYLING
CURLY BRACES

- Opening curly braces should always be presented on the same line as the construct to which they belong -- NOT wrapped underneath.

Recommended	Not Recommended

`if (YES) {`
`    [obj func];`
`}`
	`if (YES)`
	`{`
	`    [obj func];`
	`}`

OBJECTIVE-C STYLING

IF/ELSE BLOCKS

- If statements should always use curly braces to enclose their contents, even when there is only a single statement.
- If statements should always be written using multiple lines, even when there is only a single enclosed statement.
- Similar rules applies for while, do..while and for loops.

Recommended	Not Recommended
`if (YES) {` `[obj func];` `}`	`if (YES)` `[obj func];`
	`if (YES) [obj func];`

OBJECTIVE-C STYLING

IF/ELSE BLOCKS

- A single space should always be placed between the "if" keyword and the opening parenthesis.
- A single space should always be placed between the closing parenthesis and the opening curly brace.

Recommended	Not Recommended
`if (YES) {`	`if (YES){`
`    [obj func];`	`    [obj func];`
`}`	`}`

OBJECTIVE-C STYLING
IF/ELSE BLOCKS

- No space between the opening/closing parentheses and the conditional expression.

Recommended	Not Recommended
`if (YES) {`	`if ( YES ) {`
`    [obj func];`	`    [obj func];`
`}`	`}`

OBJECTIVE-C STYLING
IF/ELSE BLOCKS

- More complex if..else block should be formatted as follows.

Recommended	Not Recommended

```
if ([obj isThisTrue:1]) {
    [obj func1];
}
else if ([obj isThisTrue:2]) {
    [obj func2];
}
else {
    [obj func3];
}
```
	```
	if ([obj isThisTrue:1])
	{
	    [obj func1];
	}
	else if ([obj isThisTrue:2])
	{
	    [obj func2];
	}
	else
	{
	    [obj func3];
	}
	```

OBJECTIVE-C STYLING
IF/ELSE BLOCKS COMMENTS

- Comments for an if-statement as a whole should be placed directly above the if part.
- Comments for each condition should be placed at the top of the respective code blocks.
- Include a blank line if the comment does not apply exclusively to the first code paragraph within the block.

OBJECTIVE-C STYLING
IF/ELSE BLOCKS COMMENTS

- Comments for an if-statement as a whole should be placed directly above the if part.
- Comments for each condition should be placed at the top of the respective code blocks.
- Include a blank line if the comment does not apply exclusively to the first code paragraph within the block.

OBJECTIVE-C STYLING

IF/ELSE BLOCKS COMMENTS

```
...


// Routing to the correct handler.
if (input == kKeyboard) {
    // If the input is the keyboard, do something.

    [obj func1];
}
else if (input == kMouse) {
    // If the input is the mouse, do something else.

    [obj func2];
}
else {
    // If the input is something else, log error.

    NSLog(@"Error!");
}


...
```

OBJECTIVE-C STYLING
WRAPPING LONG LINES

- Line lengths should have a hard limit of 120 characters
- When wrapping, try to keep the most specific things grouped into single lines.
- Nest wrapping sections if necessary.
- If possible, align parameters on colon.
- Otherwise, align on the left edge of the parameters.

OBJECTIVE-C STYLING
WRAPPING LONG LINES

- If possible, align parameters on colon.

```
...

[thisIsMyVeryDescriptiveInstanceName thisIsAFunc:@”one”
                                             two:@”two”
                                           three:@”three”];
...
```

OBJECTIVE-C STYLING
WRAPPING LONG LINES

- Aligning on the left edge of the parameters.

```
...

[shortVar thisIsAFunc:@”one”
          longerSecondParamThatMakesColonAlignmentImpractical:@”two”
          three:@”three”];
...
```

OBJECTIVE-C STYLING
WRAPPING LONG LINES

- Wrap by indenting 4 spaces because of longer parameter.

```
...

[thisIsMyVeryDescriptiveInstanceName
    thisIsAFunc:@”one”
    two:@”two”
    three:@”this three is a much longer string, wrapped differently”];

...
```

OBJECTIVE-C STYLING
WRAPPING LONG LINES

- Aligning on the opening parenthesis

```
...

func(1 + (1.0f / 100.0f),
     @”hello”,
     [NSString stringWithFormat:@”hello %@! this is a longer string”,
                                @”world”);

for (int index = [obj getStartingIndexOfSomething];
     index < numItems;
     index++) {
    [obj func];
}

...
```

OBJECTIVE-C STYLING
WRAPPING LONG LINES

- Indent with an extra 4 spaces to avoid visual ambiguity

```
...

if ([obj isThisTrue] ||
        [obj iDontThinkThisWillReturnTrueButItMight] ||
        [obj hmmWhatWillThisDo] && [obj something]) {
    [obj func];
}

...
```

OBJECTIVE-C STYLING

METHODS

- Include space between the method type (+/-) indicator and first character of the method name

- Don't put spaces after the colons for method parameters

```
...

- (void)funcThatDoesSomething:(int)value1 foo:(NSString *)value2

...
```

OBJECTIVE-C STYLING
METHODS

- Keep methods short and specific.
- Break up longer methods or methods with multiple logical concepts.
- In general, avoid multiple return statements.
- Add a comment // MULTIPLE RETURNS at the top of method when multiple returns can't avoided.
- A comment is not necessary for if-statement early exit return.

OBJECTIVE-C STYLING

METHODS

- if-statement early exit return

```
...

- (float)funcDoesSomething:(int)x {
    if (x == 0) {
        return 0.0f;
    }

    float val = 5.0f / x;
    NSLog(@”val: %f”, val);
    return val;
}

...
```

OBJECTIVE-C STYLING

CLASSES

- Don’t indent access modifiers (public, protected, private)
- List sections of access modifiers in the following order: public, protected and private.
- Use a newline between sections of different access modifiers
- Pad the properties and prototypes sections, each with a newline

OBJECTIVE-C STYLING

CLASSES

- List properties before prototypes
- Alphabetize each section (per access level, properties and method prototypes).
- If there are other declarations (i.e. extern, static) separate them from the class declaration with 2 lines.

OBJECTIVE-C STYLING

CLASSES

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

extern int zero;
extern int foo;


@interface SomeClass (NSObject) {
@protected
    int one;
    NSString *two;

@private
    int three;
}

@property (nonatomic, assign) int one;

- (void)doSomething;

@end
```

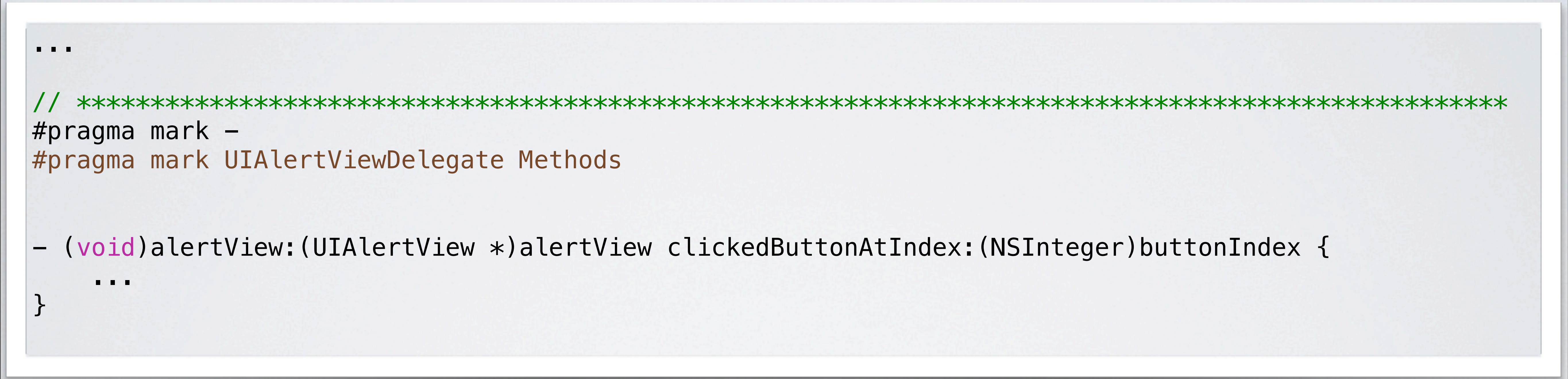
OBJECTIVE-C STYLING

CLASSES

- Use #pragma mark to organize methods into related sections.
- Alphabetize methods within related sections.
- Don’t override methods only to provide same code as the default implementation.

OBJECTIVE-C STYLING

CLASSES



```
...

// **************************************************************************************************
#pragma mark -
#pragma mark UIAlertViewDelegate Methods


- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
    ...
}
```

OBJECTIVE-C STYLING
VARIABLES

- Always use good variables, methods and class names.
- Don’t use abbreviations or acronyms except where the represented object is very commonly known by its abbreviation (i.e. ssn, fax, id).

OBJECTIVE-C STYLING
TODOS

- Use 3 levels of todos to help improve development.
- One can’t tackle all issues in a large and complicated system, simultaneously.

OBJECTIVE-C STYLING

TODOS

- Use the following for todo items that should be taken care of in the future, but are relatively low priority.

```
...

// TODO(username): This is my thing to do.

...
```

OBJECTIVE-C STYLING
TODOS

- Use the following for todo items that are very important to get done soon.
- Warning messages will ensure that people are aware of these items.

```
...
#warning TODO(username): This is my very important thing to do.
...
```

OBJECTIVE-C STYLING
TODOS

- Use the following for todo items to complete before you even build again.
- Useful when refactoring or making changes across a wide span of code.
- The build will fail, so don't checkin into source code.

```
...

#error Label optional

...
```

OBJECTIVE-C STYLING
DEPRECATION

- Use deprecation to phase out methods you wish to remove in future releases.

```
...

- (void)funcThatDoesSomething:(int)value1 DEPRECATED_ATTRIBUTE;

...
```

OBJECTIVE-C STYLING
GENERAL CODING

- Avoid conditional statement like to x == YES, x == NO, x == nil, etc.
- Use x or !x instead.
- Be deliberate about adding methods to classes scope.
- If in doubt, add them as private to keep scope as tight as possible.

OBJECTIVE-C STYLING
GENERAL CODING

- Use **NSAssert** to actively enforce cases that should be programmatically impossible.

- Comment the important stuff in your code.

- **NSAsserts** can be disabled for Release by defining **NS_BLOCK_ASSERTIONS=1** in the “**Other C Flags**” compiler options.

```
...

    NSAssert(someValueIsTrue, @”Something is very wrong”);

...
```

OBJECTIVE-C STYLING
SOURCE FILES

- .h files should only contain information vital to the public interface.
- Refrain from importing a ton of stuff in a .h header file.
- Use forward declarations as necessary (with the exception of major libraries such as UIKit and Foundation).
- An import is needed for inheritance and protocol implementations.

OBJECTIVE-C STYLING

SOURCE FILES

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

#import "SomeProtocolThisClassImplements.h"


@class SomeClassINeedForAFieldDecl;
@protocol SomeProtocolAFieldImplements;


@interface MyClass : NSObject <SomeProtocolThisClassImplements> {
    SomeClassINeedForAFieldDecl *myField;
    id <SomeProtocolAFieldImplements> *anotherField;
}

@end
```

OBJECTIVE-C STYLING
SOURCE FILES

- .m files should start with the private interface definition.
- Use className() construct vs className(private).
- ClassName() lets you define private properties.

```
@interface MyClass ()

@property (nonatomic, retain) id myPrivateField;

- (void)somePrivateMethod:(int)param;

@end
```

OBJECTIVE-C STYLING
SOURCE FILES .M

Sections of .m

1. File comments (creation date, copyright info)
2. Corresponding header import
3. Other alphabetized header imports
4. Static declarations
5. Private interface declaration
6. Implementation declaration

OBJECTIVE-C STYLING

SOURCE FILES .M

```
//
// This is a file comment with basic copyright info.
//

#import "MyClass.h"

#import "AnotherClass.h"
#import "DifferentClass.h"
#import "YetAnotherClass.h"


static const int kConstantDecl = 5;


// *************************************************************************************************
#pragma mark -
#pragma mark Private Declaration


@interface MyClass ()

@property (nonatomic, retain) NSString *aPrivateProperty;
@property (nonatomic, retain) NSString *differentPrivateProperty;

- (void)doesSomething;
- (void)processesStuff;

@end

...
```

OBJECTIVE-C STYLING
SOURCE FILES .M

```
...

// *************************************************************************************************
#pragma mark -
#pragma mark Implementation


@implementation MyClass


@synthesize aPrivateProperty;
@synthesize differentPrivateProperty;



...



@end
```

OBJECTIVE-C STYLING
SOURCE FILES .M

- Delegate and subclass method override sections should be listed under pragma marks

- Listing of pragmas may be alphabetized by the name of the protocol or class being overridden

- Pragma name should include "**Methods**" or "**Overrides**" as a suffix

- Pragma examples "**UIAlertViewDelegate Methods**" or "**UIViewController Overrides**"

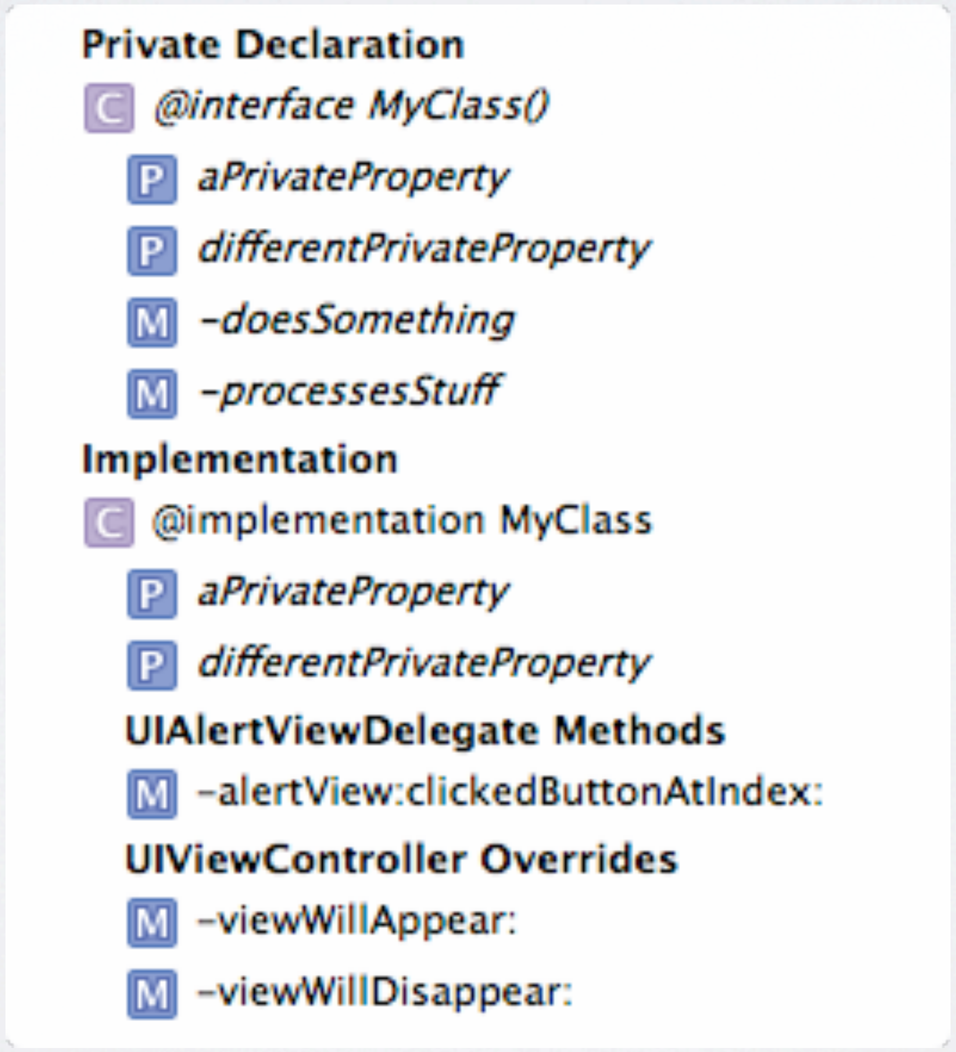
OBJECTIVE-C STYLING

SOURCE FILES PRAGMA

```
...

// *************************************************************************************************
#pragma mark -
#pragma mark UIAlertViewDelegate Methods


- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
    ...
}


// *************************************************************************************************
#pragma mark -
#pragma mark UIViewController Overrides


- (void)viewWillAppear:(BOOL)animated {
    ...
}


- (void)viewWillDisappear:(BOOL)animated {
    ...
}

...
```

OBJECTIVE-C STYLING
SOURCE FILES PRAGMA

- In large projects, pragmas allow you to have a bird's eye view of the code.



Private Declaration
C *@interface MyClass()*
  P *aPrivateProperty*
  P *differentPrivateProperty*
  M *-doesSomething*
  M *-processesStuff*

Implementation
C @implementation MyClass
  P *aPrivateProperty*
  P *differentPrivateProperty*
  UIAlertViewDelegate Methods
  M -alertView:clickedButtonAtIndex:
  UIViewController Overrides
  M -viewWillAppear:
  M -viewWillDisappear:

OBJECTIVE-C STYLING
XCODE PREFERENCES

- Under **Text Editing**, enable “Page guide”
- Specify between 100 and120 as column width
- Under **Indentation**, select “Prefer indent using: Spaces”
- Specify Tab width and Indent width as 4 spaces
- Configure Syntax-aware indenting to your liking

OBJECTIVE-C STYLING
SUMMARY

- We've reviewed useful Objective-C features
- We've gone over some best coding practices
- We've discussed Xcode settings to assist with the standard
- We've discussed how the standard is not ideal but a compromise between the parties
- The standard should be maintained in a document

OBJECTIVE-C STYLING

Questions ?