Objective C coding Standard.
2011-03-02

Table of Contents

0. Braces style.

We use only the BSD-Allman style to format our "C" conformant code

1. Spaces vs. Tabs

Use only spaces, and indent 3 spaces at a time.

2. Method Declarations and Definitions

One space MUST be used between the round bracket symbols – "**(**" and "**)**" – for the input parameter declarations. The *output parameter type* includes no spaces.

```
-(NSString*)name;
-(void)setName:( NSString* )name_;
-(int)size;
-(void)setSize:( int )size_;
```

The method name IS NEVER separated by the spaces from the input parameter type and the colon sign.

```
-(void)setName:( NSString* )name_;
```

Complex method names SHOULD be aligned according by the colon sign :

```
-(void)clipController:( RPClipController* )clip_controller_
        didLoadClips:( NSArray* ) clips_
               error:( NSError* )error_;
```

3. Method Invocations

3.1 Usual methods

Method invocations should be formatted much like method declarations. The **square brackets** MUST be **separated** from the **invocation** text with **one space**.

Actual parameters must be **separated** from the **colon sign** with **one space**

Invocations should have all arguments on one line:

```
[ myObject doFooWith: arg1_ name: arg_name_ error: error_arg_ ];
```

or have one argument per line, with colons aligned:

```
[ myObject doFooWith: arg1
                name: name
               error: error ];
```

Don't use any of these styles:

```
[myObject doFooWith:arg1 name:arg2  // some lines with >1 arg
          error:arg3];

[myObject doFooWith:arg1
           name:arg2 error:arg3];

[myObject doFooWith:arg1
        name:arg2  // aligning keywords instead of colons
        error:arg3];
```

3.2 Methods with the unknown parameters count

Their parameters MUST be organized according to the structures they represent. (in the way they will be parsed).

Each new item row MUST begin with the coma sign

```
NSDictionary* english_russian_ = [ NSDictionary dictionaryWithObjects:
                                  @"cat", "@            "
                                , @"apple", @"              "
                                , nil ];
```

Don't use any of these styles:

```
NSDictionary* english_russian_ = // commas are at the end of the row
  [ NSDictionary dictionaryWithObjects: // not at the beginning
     @"cat", "@           " ,
     @"apple", @"             " ,
     nil ];

NSDictionary* english_russian_ =  // item rows are broken
  [ NSDictionary dictionaryWithObjects:
     @"cat", "@           " ,   @"apple"
   , @"            "
   , nil ];
```

4. Operators – If, for

After "**if**" and "**for**" operators there MUST be a space before the bracket. Actually, the brackets MUST be surrounded with spaces.

```
if ( a_ > b_ )
{
  return a;
}

for ( int i_ = 0; i_ < count_; ++i_ )
{
  ;
}
```

Don't use any of these styles:

```
if ( a_ > b_)  //no space before enclosing bracket

for (int i_ = 0; i_ < count_; ++i_ ) // no space after opening bracket

if( a_ > b_ ) //the logical statement is not separated from the “if”
```

Use compiler guarding techniques in the logical statements :

```
if ( 5 == variable_ )  // good

if ( variable_ == 5 ) // avoid. Possible error :
if ( variable_ = 5 ) // Unwanted assignment, wrong result
```

Do not inline multiple operators in a single code line

```
if ( !self = [ super init ] ) // hardly readable
   return nil;

self = [ super init ];
if ( !self )
{
   return nil;
}
```

5. @public and @private

These modifiers MUST NOT be idented. The declarations below it MUST be one time idented

```
@interface A : NSObject
{
@private
   NSUInteger _pages_count;
   NSString* _title;
}
```

Don't use any of these styles:

```
@interface A : NSObject  // The declarations are not idented
{
@private
NSUInteger _pages_count;
NSString* _title;
}


@interface A : NSObject  // the keyword MUST NOT be idented
{
   @private
      NSUInteger _pages_count;
      NSString* _title;
}
```

6. Blocks Memory Management Rules

a) properties attributes

Blocks properties MUST always use "copy" attribute .

```
@interface A : NSObject
{
  …
}

typedef BOOL (^PredicateBlock) ( id object_ );

@property ( nonatomic, copy ) PredicateBlock predicate;
```

Don't use any of these attributes:

```
@property ( nonatomic, assign )  PredicateBlock predicate;

@property ( nonatomic, retain )  PredicateBlock predicate;
```

b) blocks retain context

Blocks MUST not use "super" in own function.

```
-(void)reloadData
{
   [ super reloadData ];
}

-(void)someFunc
{
   void (^block_) ( void ) = ^{ [ self reloadData ]; };
}
```

Such code may cause a crash at delayed blocks calls:

```
-(void)someFunc
{
   void (^block_) ( void ) = ^{ [ super reloadData ]; };

   [ block_ performAfterDelay: 1.0 ];
}
```

7. Naming conventions

1. Variables

ALL variable names are written in the **underscore notation**. Meaning that the parts of the name are separated with the underscore symbol and contain ONLY small letters.

Code element	Formatting description	Example
Local variables	MUST end with the underscore sign	`NSUInteger pages_count_ = 5;`

Instance variables	MUST start with the underscore sign	`@interface ABook : NSObject`
		`{`
		`@private`
		`NSUInteger _pages_count;`
		`}`

private global variables	MUST end with the underscore sign (same as local variables)	`static NSUInteger max_pages_count_ = 5;`

global variables with public usage	according to the apple standard	MyConstants.h
		`extern NSUInteger RNGlobalVariable;`
		`extern NSString* const RNAnotherGlobalVariable;`
		--- 	
		MyConstants.m
		`NSUInteger RNGlobalVariable = 5;`
		`NSString* const RNAnotherGlobalVariable = @“Hello”;`

If we have several constant of integral type better unite them into one enum, each constant name should start with enum name prefix. Coma separator MUST be used before every single constant of the enum.

```
typedef enum
{
  RNColorWhite
  , RNColorBlack
  , RNColorRed
} RNColorType;
```

2. Properties

ALL property names MUST be written in the Camel case notation

Properties declarations MUST be ordered in the same way as instance variables are.

The optional property qualifiers SHOULD be omitted. However, you can write them

explicitly if you want to.

The qualifiers list MUST be separated by spaces from the @property keyword and the property type.

.h
```
@property ( nonatomic, assign ) NSUInteger pagesCount;
```

.m
```
@synthesize pagesCount = _pages_count;
```

Don't use any of these styles:

```
// no space after keyword
@property( nonatomic, assign ) NSUInteger pagesCount;

// no space before type
@property ( nonatomic, assign )NSUInteger pagesCount;

// no space before the brace
@property ( nonatomic, assign) NSUInteger pagesCount;

// no space after the brace
@property (nonatomic, assign ) NSUInteger pagesCount;
```

3. Primitive types

Avoid using raw C types. Use typedefs form the Foundation framework instead. Do not use int instead of bool. Do not mix signed and unsigned types.

```
// GOOD
BOOL operation_result_ = YES;
@property ( nonatomic, assign ) NSUInteger pagesCount;
// BAD
bool operation_result_ = YES;
@property ( nonatomic, assign ) int pagesCount;
//Ugly
int operation_result_ = 1;
```

4. File names

File names MUST reflect the name of the class implementation that they contain -- including case. Follow the convention that your project uses.
File extensions should be as follows:

`.h`	C/C++/Objective-C header file
`.m`	Objective-C implementation file
`.mm`	Objective-C++ implementation file
`.cpp or .cc`	Pure C++ implementation file
`.c`	C implementation file

File names for categories MUST start with the name of the class being extended, e.g.
`GTMNSString+Utils.h` or
`GTMNSTextView+Autocomplete.h`

One file MUST contain ONLY ONE **interface** or **protocol** declaration.
However, a file MAY contain **multiple categories** of a **single interface**.

5. Class Names

Class names (along with category and protocol names) should start as uppercase and use Camel case notation to delimit words.

Each class MUST have a prefix, made up from the project or library name shortcut (e.g. `GTMSendMessage`).

6. Method names

The method name should read like a sentence if possible, meaning you should choose parameter names that flow with the method name. (e.g. `convertPoint:fromRect:` or `replaceCharactersInRange:withString:`). See Apple's Guide to Naming Methods for more details.

Accessor methods should be named the same as the variable they're "getting", but they should *not* be prefixed with the word "get". For example:

```
- (id)getDelegate;  // AVOID
- (id)delegate;    // GOOD
```

This is for Objective-C methods only.

Comments

The **implementation source files** SHOULD NOT contain any **comments**. The code SHOULD be self-documented (as simple and as clean as possible).

Any comments will indicate that fixes are required for this file.

Header files, MAY contain some commented stuff.
For example, it can be

- an expected NSArray (or other collection) item type.
- Signature of the selector parameter for the method
- Limitations for the parameters
- Designated initializer
- etc.

However, comments MUST be avoided. Use them ONLY in emergency cases.

Cocoa and Objective-C Features

1. Member Variables Should Be @private

Do not use them under ARC. Really! If you really need to do so, please put them to the @**implementation** section. Clang compiler works with such declarations just fine.

Member variables should be declared `@private`.

```
@implementation MyClass
{
@private
   UIView* _my_view;
   UIViewControl* _parent_view_controller;

   id _my_instance_variable;
}
// public accessors, setter takes ownership
-(id)myInstanceVariable;
-(void)setMyInstanceVariable: ( id )theVar;
@end
```

To **access** the instance variables you MUST use **properties** rather than ivars.

Order of instance variables is important. The declaration order is :

1. view variables
2. controller variables
3. model variables

1.1 There is only situation when you may want to leave @public ivars

In this case you deal with some legacy code that uses POD (Plain Old Data) structures in old-school C style. However, you should consider advantages of Objective-C++ and constant links.

1.2 Still declaring ivars under ARC? No need.

That's right! The compiler will do the right job for you having just property declarations and @**synthesize** statements.

Note : If you still prefer declaring ivars under ARC, please make sure they have same attributes as corresponding properties.

```
@interface Todo : NSObject
{
@private
  __weak NSString* _text
}

@property (nonatomic, strong) NSString *text; // the ivar was weak !!!!
@end
```

2. Identify Designated Initializer

Comment and clearly identify your designated initializer.
It is important for those who might be subclassing your class that the designated initializer be clearly identified. That way, they only need to subclass a single initializer (of potentially several) to guarantee their subclass' initializer is called. It also helps those debugging your class in the future understand the flow of initialization code if they need to step through it.

If you replace a superclass designated initializer with your own one, you MUST provide an implementation for the superclass designated initializer.
If it is impossible to implement a superclass designated initializer (some very custom initialization is required), please create a stub implementation using NSAssert.

3. Initialization

Don't initialize variables to `0` or `nil` in the init method; it's redundant.
All memory for a newly allocated object is initialized to 0 (except for *isa*), so don't clutter up the `init` method by re-initializing variables to 0 or `nil`.

Do not use convenience constructors under ARC. This awesome technology makes them utterly useless. Implement initializers instead.

3.1 Initializing UIView Subclasses

While subclassing UIViews, you MUST implement the following initialization methods :

- -(id)init
- -(id)initWithFrame:(CGRect)frame_
- -(void)awakeFromNib

Don't forget to call [super awakeFromNib];

4. #import and #include

`#import` Objective-C/Objective-C++ headers, and `#include` C/C++ headers. Choose between `#import` and `#include` based on the language of the header that you are including.

When including headers from the library (third-party or our own one), please use triangle brackets **<>** instead of double quotes **“”**

```
#import <ESCommon/ESUI/StripeView/ESStripeView.h>  // good
#import <JSON/SBJson.h>  // good
#import “JSON/SBJson.h”  // avoid
```

!!! NEVER use recursive include feature of the Xcode. !!!

```
#import <ESCommon/ESUI/StripeView/ESStripeView.h>  // good
#import <ESStripeView.h>  //avoid – missing base path
ESCommon/ESUI/StripeView/
```

Header files MUST be included in the following order (from least general headers to most general ones) :

1. The corresponding header for the implementation file.
2. View headers (including their delegates) from the same project.
3. View controller headers (including their delegates) from the same project.
4. Busyness logic headers from the same project
5. View headers (including their delegates) from our libraries.
6. View controller headers (including their delegates) from our libraries .

7. Busyness logic headers from our libraries
8. View headers (including their delegates) from third-party libraries.
9. View controller headers (including their delegates) from third-party libraries .
10. Busyness logic headers from third-party libraries
11. Standard libraries and frameworks
12. Objective-C runtime interface headers.

In the **header files** forward declarations of classes and protocols (**@class,** @**protoco**l) SHOULD be used **instead** of **#import/include** statements wherever possible.

Inside the library the **triangle brackets** MUST be used **in the headers** instead of the quotes **in the #include and #import** directives.

4.1 Pre-compiled Headers

In the main project you must put the following things to the pre-compiled header file :

1. Root headers for Apple frameworks.
2. Root headers for third-party and common libraries (in case such root headers exist and is supported by the library development team).
3. Headers of used individual classes and protocols from common and third-party libraries.

For more details on the pre-compiled headers, please consider the following article

5. Include Root Frameworks Over Individual Files.

While it may seem tempting to include individual system headers from a framework such as Cocoa or Foundation, in fact it's less work on the compiler if you include the top-level root framework. The root framework is generally pre-compiled and can be loaded much more quickly. In addition, remember to use `#import` rather than `#include` for Objective-C frameworks.

```
#import <Foundation/Foundation.h>     // good
#import <Foundation/NSArray.h>        // avoid
#import <Foundation/NSString.h>
...
```

The same thing applies to the header files from the third-party libraries and our own custom libraries.

!!! Note !!! : these headers must be included to the pre-compiled header file first.

6. dealloc Should Process Instance Variables in Declaration Order

`dealloc` MUST process **instance variables** in the **same order** the `@interface` **declares** them, so it is easier for a reviewer to verify.
A code reviewer checking a new or revised `dealloc` implementation needs to make sure that every retained instance variable gets released.

To simplify reviewing `dealloc`, order the code so that the retained instance variables get released in the same order that they are declared in the `@interface`. If `dealloc` invokes other methods that release instance variables, add comments describing what instance variables those methods handle.

The **dealloc** implementation MUST **go just after** the properties **@synthesize** statements.

Note : most likely you won't even have to implement **dealloc** it under ARC.

7. Avoid Throwing Exceptions

Don't `@throw` Objective-C exceptions, but you should be prepared to catch them from third-party or OS calls. Make sure to catch all C++ exceptions. Even those, implicitly produced by STL, boost and other libraries.

8. BOOL Pitfalls

Be careful when converting general integral values to `BOOL`. Avoid comparing directly with `YES`.

`BOOL` is defined as an unsigned char in Objective-C which means that it can have values other than `YES` (1) and `NO` (0). Do not cast or convert general integral values directly to `BOOL`. Common mistakes include casting or converting an array's size, a pointer value, or the result of a bitwise logic operation to a `BOOL` which, depending on the value of the last byte of the integral result, could still result in a `NO` value. When converting a general integral value to a `BOOL` use ternery operators to return a `YES` or `NO` value.

You can safely interchange and convert `BOOL`, `_Bool` and `bool` (see C++ Std 4.7.4, 4.12 and C99 Std 6.3.1.2). You cannot safely interchange `BOOL` and `Boolean` so treat `Booleans` as a general integral value as discussed above. Only use `BOOL` in Objective C method signatures.

Using logical operators (`&&`, `||` and `!`) with `BOOL` is also valid and will return values that can be safely converted to `BOOL` without the need for a ternery

operator.

```
- (BOOL)isBold
{
  return [self fontTraits] & NSFontBoldTrait;
}
- (BOOL)isValid
{
  return [self stringValue];
}
- (BOOL)isBold
{
  return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
}
- (BOOL)isValid
{
  return [self stringValue] != nil;
}
- (BOOL)isEnabled
{
  return [self isValid] && [self isBold];
}
```

Also, don't directly compare `BOOL` variables directly with `YES`. Not only is it harder to read for those well-versed in C, the first point above demonstrates that return values may not always be what you expect.

```
BOOL great = [foo isGreat];
if (great == YES)
  // ...be great!

BOOL great = [foo isGreat];
if (great)
  // ...be great!
```

Properties

1. List out all implementation directives

Use implementation directives for all properties even if they are `@dynamic` by default.

Even though `@dynamic` is default, explicitly list it out with all of the other property implementation directives making it clear how every property in a class is handled at a single glance.

```
@interface MyClass : NSObject
@property(readonly) NSString *name;
@end

@implementation MyClass
.
.
.
- (NSString*)name {
  return @"foo";
}
@end
@interface MyClass : NSObject
@property(readonly) NSString *name;
@end

@implementation MyClass
@dynamic name;
.
.
.
- (NSString*)name {
  return @"foo";
}
@end
```

2. Atomicity

Be aware of the overhead of properties. By default, all synthesized setters and getters are atomic. This gives each set and get calls a substantial amount of synchronization overhead. Declare your properties `nonatomic` unless you require atomicity.

3. Dot notation

Dot notation is idiomatic style for Objective-C 2.0. It may be used when doing simple operations to get and set a `@property` of an object, but should not be used to invoke other object behavior.

```
NSString *oldName = myObject.name;
myObject.name = @"Alice";
NSArray *array = [[NSArray arrayWithObject:@"hello"]
retain];

NSUInteger numberOfItems = array.count;  // not a property
array.release;                           // not a property
```

Using properties within the class implementation file, ALWAYS use the "**self**" keyword

```
-(void)doWorkWithFoo
{
   [ self.foo doWork ];
}

-(void)doWorkWithFoo
{
   [ foo doWork ];  // no "self" keyword
}
```

4. Deinitialization

NEVER set properties to **nil** in **dealloc**.
Release instance variables explicitly instead.

```
-(void)dealloc
{
   [ _foo release ];
    [ super dealloc ];
}

-(void)dealloc
{
   self.foo = nil;  // side effect may take place
   [ super dealloc ];
}
```

Repository Structure

Filesystem layout

The root of the SVN should contain the standard directories :

- **trunk**
- **branches**
- **tags**

Inside of the trunk (versioned branch/tag directory) there SHOULD be the following directories :

- **app** – for main project sources
- **lib** – shared code, maintained by our team. Libraries, containing the **model** and **network protocol bindings** should also be stored here.
- **lib-third-party** – for libraries and frameworks used by our applications
- **test** – for unit tests of all kinds (*Sen Testing Kit*, *Google toolbox*, *GHUnit*).
- **tools** – ready-to-use binaries and or source code which are used
- **scripts** – scripts for continuous integration in *bash*/*ruby*/*python*/*<your favorite scripting language>*.
- **certificates** – apple provisioning and developer profiles must be stored here.
- **doc** – for various documentation (estimates, design drafts, UML diagrams, etc.)

There MAY be some additional directories :

- **frameworks** – pre-build third-party libraries and those maintained by our team. It should contain no *.xcodeproj entries. Instead only deployed libraries must be placed there. All frameworks and universal libraries, developed by our team must be deployed into this directory.
- **deployment** – this directory should not be under version control. All applications must be deployed into it by our CI scripts. *Libraries* and *frameworks* MAY be redeployed from the *frameworks* directory for convenience purposes only.

The src directory contains all applications of . Each of them stores some Xcode generated files. The source tree MUST look in the **same** way both in the **filesystem** and in the **Xcode**. This is applied to the resources (images, localizations, CoreData model files, *.xib UI design files).

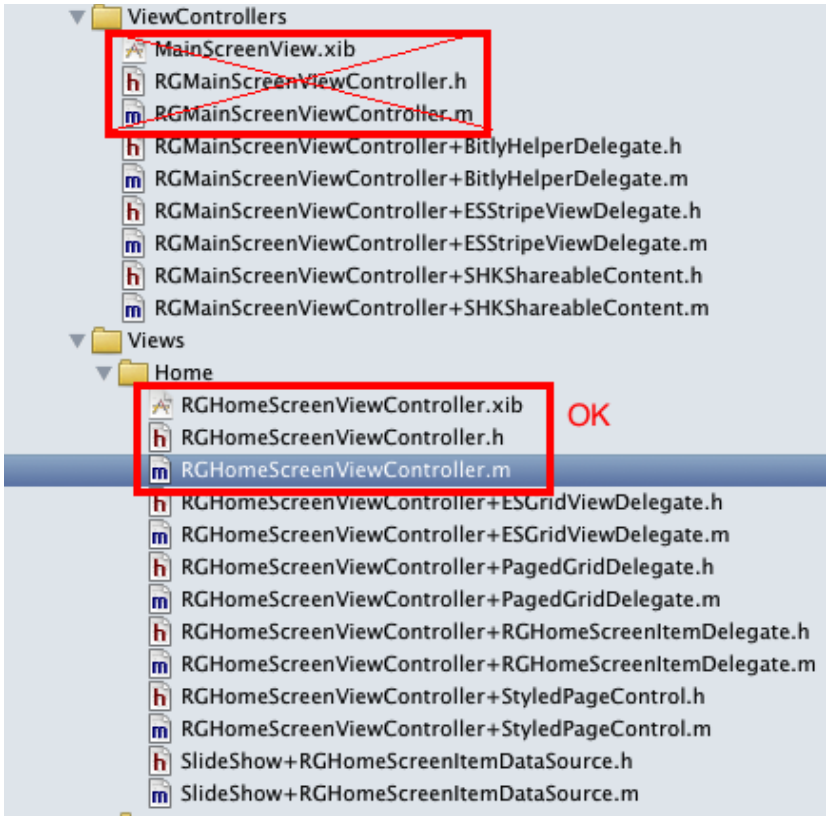
Source, implementation files and folders SHOULD be set up in the alphabetical order within the xCode.

Projects MUST be packaged according to the rule "**One Xcode project – one library**".
Clint Harris has a good tutorial to get familiar with the static libraries for iPhone.

Nib Files Naming

In case the library uses some resources (CoreData models, Interface Builder xib files, images, localized strings, etc.), they MUST be included into the library project. This project MUST have a separate Bundle target for those files. The description of required actions can be found at StackOverflow

A nib file MUST have the same name as its FilesOwner does.



If you are subclassing UIView and use nib files for user interface development, you MUST place those nib files in the same directory with the code.
The same implies for the CoreData models.

Library projects that incorporate custom ***UI controls*** MUST have a target of a ***"bundle" type*** with their nib files, localized strings and other resources.
In the main application and the library code we USUALLY suppose that those bundles are copied to the [NSBundle mainBundle].

Interface and Protocol Declarations

Interfaces and protocols MUST conform the rule : "**One interface – one unit**". Meaning, the unit is a pair of the header and implementation files.

In the **header files** forward declarations of classes and protocols (**@class, @protocol**) SHOULD be used **instead** of **#import/include** statements wherever possible.

Within the @interface / @protocol block the instance variables MUST be declared in the way, described above (page 14, section 4. #import and #include).

If the class does not contain any instance variable declarations, the curly braces block MUST be omitted.

A property's declaration must come immediately after the instance variable block of a class interface. A property's definition must come immediately after the @implementation block in a class definition. They are indented at the same level as the @interface or @implementation statements that they are enclosed in.

After the property declarations the initializers and convenience constructors. Then usual methods should go.

Summarizing above statements, we get the following declaration order :

1. Essential includes
2. Forward declarations of required interfaces and protocols
3. Interface declaration and instance variables block
4. properties block
5. initializers
6. convenience constructors
7. usual methods

NEVER declare properties in your protocols. Protocols should contain only instance and class methods

The same order should be in the implementation file. The only addition is the **dealloc** implementation **between** the **@synthesize** statements and **initializers**.

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

@protocol ESStripeViewDelegate;

@interface ESStripeView : UIView
{
@private
   UIScrollView* _scroll_view;

   CGRect _previous_frame;

   UILabel* _warning_label;

   NSUInteger _active_element;

   NSMutableArray* _reusable_elements;
   NSMutableDictionary* _elements_by_index;

   CGFloat _left_inset;
   CGFloat _right_inset;

   id< ESStripeViewDelegate > _delegate;
}

@property ( nonatomic, retain, readonly ) UIScrollView* scrollView;
@property ( nonatomic, retain, readonly ) UILabel* warningLabel;
@property ( nonatomic, retain, readonly ) UIView* activeElementView;
@property ( nonatomic, assign, readonly ) NSUInteger activeElement;

@property ( nonatomic, assign ) IBOutlet id< ESStripeViewDelegate > delegate;

-(id)initWithFrame:( CGRect )frame_
              delegate:( id< ESStripeViewDelegate > )delegate_;

+(id)stripeViewWithFrame: ( CGRect )frame_
                        delegate:( id< ESStripeViewDelegate > )delegate_;

-(void)reloadData;
-(void)relayoutElements;
-(UIView*)dequeueReusableElement;
-(UIView*)elementAtIndex:( NSUInteger )index_;
-(NSArray*)visibleElements;

-(void)removeElementWithIndex:( NSUInteger )index_
                                  animated:( BOOL )animated_;

-(void)insertElementAtIndex:( NSUInteger )index_
                  animated:( BOOL )animated_;

-(void)exchangeElementAtIndex:( NSUInteger )first_index_
          withElementAtIndex:( NSUInteger )second_index_;

-(void)slideForward;
-(void)slideToIndex:( NSInteger )index_ animated:(BOOL)animated_;
-(void)slideToIndex:( NSInteger )index_;

@end
```

Objective-C Blocks

Objective-C block variables should be aligned to their declaration. Just like function declarations are. For example :

```
id block_ = ^void()
{
  CGRect newFrame = self.imageView.frame;
  newFrame.origin.y = newFrame.origin.y + 150.0;
  self.imageView.frame = newFrame;
  self.imageView.alpha = 0.2;
};
```

Inlined Objective-C Blocks

We strongly recommend using local variables of the block type wherever possible. If you don't do this, block body should start directly under the selector colon separator.

```
  [UIView animateWithDuration:1.5
                   animations: ^void()
                               {
                               // your code here
                               }
                   completion: ^void(BOOL finished)
                               {
                                // Your code here.
                               } ];
```

This keeps block code close to its declaration. The only disadvantage is that your code becomes too wide.

Some may think alignment to the place selector parts begin is a good idea.

```
  [UIView animateWithDuration:1.5
                   animations: ^void()
                     {
                         // your code here
                    }
                   completion: ^void(BOOL finished)
                     {
                      // Your code here.
                    } ];
```

However, you won't always be that lucky to have selector parts of equal length. Yes, both “animations” and “completion” have 10 letters! In this case you may end up with code piece

that looks like a mess:

```
[UIView animateWithDuration:1.5
            customAnimations: ^void()
             {
                 // your code here
             }
                completion: ^void(BOOL finished)
                  {
                    // Your code here.
                } ];
```

Note : We strongly recommend **not** to use inlined block variables. Local variables will make your code a lot more clear than any body alignment style.

Xcode usage

NEVER use recursive include feature of the Xcode.
Unless you're dealing with really complex legacy code and this is your last resort.

Install latest xCode SDKs to the default directory (**/Developer**). Older SDKs should be reinstalled to some other directories if necessary.
Luckily, you won't have any of these problems with xCode4 under Lion.
Anyway, **xcode-select** command will help you to set up your environment in the way you like.

Edit project settings rather than target settings. This applies to the headers, libraries search path, etc. Do not repeat yourselves !!!

Targets may contain specific preprocessor directives, build architectures and target OS version.

Think of targets as about different representation of a single thing. For example, a library project may have one target to a produce static binary, one for a dynamic binary and one more for a framework. On the contrary, you should not put your dependencies to targets.