Coding Guidelines for Cocoa

Developer

Contents

Listings

Introduction to Coding Guidelines for Cocoa

Developing a Cocoa framework, plug-in, or other executable with a public API requires some approaches and conventions that are different from those used in application development. The primary clients of your product are developers, and it is important that they are not mystified by your programmatic interface. This is where API naming conventions come in handy, for they help you to make your interfaces consistent and clear. There are also programming techniques that are special to—or of greater importance with—frameworks, such as versioning, binary compatibility, error-handling, and memory management. This topic includes information on both Cocoa naming conventions and recommended programming practices for frameworks.

Organization of This Document

The articles contained in this topic fall into two general types. The first and larger group presents naming conventions for programmatic interfaces. These are the same conventions (with some minor exceptions) that Apple uses for its own Cocoa frameworks. These articles on naming conventions include the following:

The second group (currently with a membership of one) discusses aspects of framework programming:

Code Naming Basics

An often overlooked aspect of the design of object-oriented software libraries is the naming of classes, methods, functions, constants, and the other elements of a programmatic interface. This section discusses several of the naming conventions common to most items of a Cocoa interface.

General Principles

Clarity

- It is good to be both clear and brief as possible, but clarity shouldn't suffer because of brevity:

Code	Commentary
`insertObject: atIndex:`	Good.
`insert:at:`	Not clear; what is being inserted? what does "at" signify?
`removeObjectAtIndex:`	Good.
`removeObject:`	Good, because it removes object referred to in argument.
`remove:`	Not clear; what is being removed?

- In general, don't abbreviate names of things. Spell them out, even if they're long:

Code	Commentary
`destinationSelection`	Good.
`destSel`	Not clear.
`setBackgroundColor:`	Good.
`setBkgdColor:`	Not clear.

  You may think an abbreviation is well-known, but it might not be, especially if the developer encountering your method or function name has a different cultural and linguistic background.

- However, a handful of abbreviations are truly common and have a long history of use. You can continue to use them; see "Acceptable Abbreviations and Acronyms" (page 24).

- Avoid ambiguity in API names, such as method names that could be interpreted in more than one way.

Code	Commentary
`sendPort`	Does it send the port or return it?
`displayName`	Does it display a name or return the receiver's title in the user interface?

Consistency

- Try to use names consistently throughout the Cocoa programmatic interfaces. If you are unsure, browse the current header files or reference documentation for precedents.
- Consistency is especially important when you have a class whose methods should take advantage of polymorphism. Methods that do the same thing in different classes should have the same name.

Code	Commentary
`- (NSInteger)tag`	Defined in `NSView`, `NSCell`, `NSControl`.
`- (void)setStringValue:(NSString *)`	Defined in a number of Cocoa classes.

See also "Method Arguments" (page 16).

No Self Reference

- Names shouldn't be self-referential.

Code	Commentary
`NSString`	Okay.
`NSStringObject`	Self-referential.

- Constants that are masks (and thus can be combined in bitwise operations) are an exception to this rule, as are constants for notification names.

Code	Commentary
`NSUnderlineByWordMask`	Okay.
`NSTableViewColumnDidMoveNotification`	Okay.

Prefixes

Prefixes are an important part of names in programmatic interfaces. They differentiate functional areas of software. Usually this software comes packaged in a framework or (as is the case of Foundation and Application Kit) in closely related frameworks. Prefixes protect against collisions between symbols defined by third-party developers and those defined by Apple (as well as between symbols in Apple's own frameworks).

- A prefix has a prescribed format. It consists of two or three uppercase letters and does not use underscores or "sub prefixes." Here are some examples:

Prefix	Cocoa Framework
NS	Foundation
NS	Application Kit
AB	Address Book
IB	Interface Builder

- Use prefixes when naming classes, protocols, functions, constants, and `typedef` structures. Do *not* use prefixes when naming methods; methods exist in a name space created by the class that defines them. Also, don't use prefixes for naming the fields of a structure.

Typographic Conventions

Follow a few simple typographical conventions when naming API elements:

- For names composed of multiple words, do not use punctuation marks as parts of names or as separators (underscores, dashes, and so on); instead, capitalize the first letter of each word and run the words together (for example, `runTheWordsTogether`)—this is known as camel-casing. However, note the following qualifications:
  - For method names, start with a lowercase letter and capitalize the first letter of embedded words. Don't use prefixes.

    ```
    fileExistsAtPath:isDirectory:
    ```

    An exception to this guideline is method names that start with a well-known acronym, for example, `TIFFRepresentation` (`NSImage`).
  - For names of functions and constants, use the same prefix as for related classes and capitalize the first letter of embedded words.

```
NSRunAlertPanel
NSCellDisabled
```

- Avoid the use of the underscore character as a prefix meaning private in *method* names (using an underscore character as a prefix for an *instance variable* name is allowed). Apple reserves the use of this convention. Use by third parties could result in name-space collisions; they might unwittingly override an existing private method with one of their own, with disastrous consequences. See "Private Methods" (page 17) for suggestions on conventions to follow for private API.

Class and Protocol Names

The name of a class should contain a noun that clearly indicates what the class (or objects of the class) represent or do. The name should have an appropriate prefix (see "Prefixes" (page 8)). The Foundation and application frameworks are full of examples; a few are `NSString`, `NSDate`, `NSScanner`, `NSApplication`, `UIApplication`, `NSButton`, and `UIButton`.

Protocols should be named according to how they group behaviors:

- 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. This protocol groups methods that you can use to query any object about its position in the class hierarchy, to make it invoke specific methods, and to increment or decrement its reference count. Because the `NSObject` class provides the primary expression of these methods, the protocol is named after the class.

Header Files

How you name header files is important because the convention you use indicates what the file contains:

- **Declaring an isolated class or protocol**. If a class or protocol isn't part of a group, put its declaration in a separate file whose name is that of the declared class or protocol.

Header file	Declares
`NSLocale.h`	The `NSLocale` class.

- **Declaring related classes and protocols**. For a group of related declarations (classes, categories, and protocols), put the declarations in a file that bears the name of the primary class, category, or protocol.

Header file	Declares
`NSString.h`	`NSString` and `NSMutableString` classes.
`NSLock.h`	`NSLocking` protocol and `NSLock`, `NSConditionLock`, and `NSRecursiveLock` classes.

- **Including framework header files**. Each framework should have a header file, named after the framework, that includes all the public header files of the framework.

Header file	Framework
`Foundation.h`	`Foundation.framework`.

- **Adding API to a class in another framework**. If you declare methods in one framework that are in a category on a class in another framework, append "Additions" to the name of the original class; an example is the `NSBundleAdditions.h` header file of the Application Kit.
- **Related functions and data types**. If you have a group of related functions, constants, structures, and other data types, put them in an appropriately named header file such as `NSGraphics.h` (Application Kit).

Naming Methods

Methods are perhaps the most common element of your programming interface, so you should take particular care in how you name them. This section discusses the following aspects of method naming:

General Rules

Here are a few general guidelines to keep in mind when naming methods:

- Start the name with a lowercase letter and capitalize the first letter of embedded words. Don't use prefixes. See "Typographic Conventions" (page 8).

  There are two specific exceptions to these guidelines. You may begin a method name with a well-known acronym in uppercase (such as TIFF or PDF)), and you may use prefixes to group and identify private methods (see "Private Methods" (page 17)).

- For methods that represent actions an object takes, start the name with a verb:

  ```
  - (void)invokeWithTarget:(id)target;
  - (void)selectTabViewItem:(NSTabViewItem *)tabViewItem
  ```

  Do not use "do" or "does" as part of the name because these auxiliary verbs rarely add meaning. Also, never use adverbs or adjectives before the verb.

- If the method returns an attribute of the receiver, name the method after the attribute. The use of "get" is unnecessary, unless one or more values are returned indirectly.

`- (NSSize)cellSize;`	Right.
`- (NSSize)calcCellSize;`	Wrong.
`- (NSSize)getCellSize;`	Wrong.

  See also "Accessor Methods" (page 12).

- Use keywords before all arguments.

`- (void)sendAction:(SEL)aSelector to:(id)anObject forAllCells:(BOOL)flag;`	Right.

`- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;`	Wrong.

- Make the word before the argument describe the argument.

`- (id)viewWithTag:(NSInteger)aTag;`	Right.
`- (id)taggedView:(int)aTag;`	Wrong.

- Add new keywords to the end of an existing method when you create a method that is more specific than the inherited one.

`- (id)initWithFrame:(CGRect)frameRect;`	`NSView`, `UIView`.
`- (id)initWithFrame:(NSRect)frameRect mode:(int)aMode cellClass:(Class)factoryId numberOfRows:(int)rowsHigh numberOfColumns:(int)colsWide;`	NSMatrix, a subclass of NSView

- Don't use "and" to link keywords that are attributes of the receiver.

`- (int)runModalForDirectory:(NSString *)path file:(NSString *)name types:(NSArray *)fileTypes;`	Right.
`- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(NSArray *)fileTypes;`	Wrong.

Although "and" may sound good in this example, it causes problems as you create methods with more and more keywords.

- If the method describes two separate actions, use "and" to link them.

`- (BOOL)openFile:(NSString *)fullPath withApplication:(NSString *)appName andDeactivate:(BOOL)flag;`	`NSWorkspace`.

Accessor Methods

Accessor methods are those methods that set and return the value of a property of an object. They have certain recommended forms, depending on how the property is expressed:

- If the property is expressed as a noun, the format is:

  `- (type)noun;`

  `- (void)setNoun:(type)aNoun;`

  For example:

```
- (NSString *)title;
- (void)setTitle:(NSString *)aTitle;
```

- If the property is expressed as an adjective, the format is:

  `- (BOOL)isAdjective;`

  `- (void)setAdjective:(BOOL)flag;`

  For example:

  ```
  - (BOOL)isEditable;
  - (void)setEditable:(BOOL)flag;
  ```

- If the property is expressed as a verb, the format is:

  `- (BOOL)verbObject;`

  `- (void)setVerbObject:(BOOL)flag;`

  For example:

  ```
  - (BOOL)showsAlpha;
  - (void)setShowsAlpha:(BOOL)flag;
  ```

  The verb should be in the simple present tense.

- Don't twist a verb into an adjective by using a participle:

`- (void)setAcceptsGlyphInfo:(BOOL)flag;`	Right.
`- (BOOL)acceptsGlyphInfo;`	Right.
`- (void)setGlyphInfoAccepted:(BOOL)flag;`	Wrong.
`- (BOOL)glyphInfoAccepted;`	Wrong.

- You may use modal verbs (verbs preceded by "can", "should", "will", and so on) to clarify meaning, but don't use "do" or "does".

`- (void)setCanHide:(BOOL)flag;`	Right.
`- (BOOL)canHide;`	Right.
`- (void)setShouldCloseDocument:(BOOL)flag;`	Right.

`- (BOOL)shouldCloseDocument;`	Right.
`- (void)setDoesAcceptGlyphInfo:(BOOL)flag;`	Wrong.
`- (BOOL)doesAcceptGlyphInfo;`	Wrong.

- Use "get" only for methods that return objects and values indirectly. You should use this form for methods only when multiple items need to be returned.

`- (void)getLineDash:(float *)pattern count:(int *)count phase:(float *)phase;`	`NSBezierPath`.

In methods such as these, the implementation should accept `NULL` for these in–out parameters as an indication that the caller is not interested in one or more of the returned values.

Delegate Methods

Delegate methods (or delegation methods) are those that an object invokes in its delegate (if the delegate implements them) when certain events occur. They have a distinctive form, which apply equally to methods invoked in an object's data source:

- Start the name by identifying the class of the object that's sending the message:

```
- (BOOL)tableView:(NSTableView *)tableView shouldSelectRow:(int)row;
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;
```

The class name omits the prefix and the first letter is in lowercase.

- A colon is affixed to the class name (the argument is a reference to the delegating object) unless the method has only one argument, the sender.

```
- (BOOL)applicationOpenUntitledFile:(NSApplication *)sender;
```

- An exception to this are methods that invoked as a result of a notification being posted. In this case, the sole argument is the notification object.

```
- (void)windowDidChangeScreen:(NSNotification *)notification;
```

- Use "did" or "will" for methods that are invoked to notify the delegate that something has happened or is about to happen.

```
- (void)browserDidScroll:(NSBrowser *)sender;
- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)window;
```

- Although you can use “did” or “will” for methods that are invoked to ask the delegate to do something on behalf of another object, “should” is preferred.

```
- (BOOL)windowShouldClose:(id)sender;
```

Collection Methods

For objects that manage a collection of objects (each called an element of that collection), the convention is to have methods of the form:

```
- (void)addElement:(elementType)anObj;
- (void)removeElement:(elementType)anObj;
- (NSArray *)elements;
```

For example:

```
- (void)addLayoutManager:(NSLayoutManager *)obj;
- (void)removeLayoutManager:(NSLayoutManager *)obj;
- (NSArray *)layoutManagers;
```

The following are some qualifications and refinements to this guideline:

- If the collection is truly unordered, return an NSSet object rather than an NSArray object.
- If it’s important to insert elements into a specific location in the collection, use methods similar to the following instead of or in addition to the ones above:

```
- (void)insertLayoutManager:(NSLayoutManager *)obj atIndex:(int)index;
- (void)removeLayoutManagerAtIndex:(int)index;
```

There are a couple of implementation details to keep in mind with collection methods:

- These methods typically imply ownership of the inserted objects, so the code that adds or inserts them must retain them, and the code that removes them must also release them.
- If the inserted objects need to have a pointer back to the main object, you do this (typically) with a `set...` method that sets the back pointer but does not retain. In the case of the `insertLayoutManager:atIndex:` method, the NSLayoutManager class does this in these methods:

  ```
  - (void)setTextStorage:(NSTextStorage *)textStorage;
  - (NSTextStorage *)textStorage;
  ```

  You would normally not call `setTextStorage:` directly, but might want to override it.

Another example of the above conventions for collection methods comes from the NSWindow class:

```
- (void)addChildWindow:(NSWindow *)childWin ordered:(NSWindowOrderingMode)place;
- (void)removeChildWindow:(NSWindow *)childWin;
- (NSArray *)childWindows;

- (NSWindow *)parentWindow;
- (void)setParentWindow:(NSWindow *)window;
```

Method Arguments

There are a few general rules concerning the names of method arguments:

- As with methods, arguments start with a lowercase letter and the first letter of successive words are capitalized (for example, `removeObject:(id)anObject`).
- Don't use "pointer" or "ptr" in the name. Let the argument's type rather than its name declare whether it's a pointer.
- Avoid one- and two-letter names for arguments.
- Avoid abbreviations that save only a few letters.

Traditionally (in Cocoa), the following keywords and arguments are used together:

```
...action:(SEL)aSelector
...alignment:(int)mode
```

```
...atIndex:(int)index
...content:(NSRect)aRect
...doubleValue:(double)aDouble
...floatValue:(float)aFloat
...font:(NSFont *)fontObj
...frame:(NSRect)frameRect
...intValue:(int)anInt
...keyEquivalent:(NSString *)charCode
...length:(int)numBytes
...point:(NSPoint)aPoint
...stringValue:(NSString *)aString
...tag:(int)anInt
...target:(id)anObject
...title:(NSString *)aString
```

Private Methods

In most cases, private method names generally follow the same rules as public method names. However, a common convention is to give private methods a prefix so it is easy to distinguish them from public methods. Even with this convention, the names given to private methods can cause a peculiar type of problem. When you design a subclass of a Cocoa framework class, you cannot know if your private methods unintentionally override private framework methods that are identically named.

Names of most private methods in the Cocoa frameworks have an underscore prefix (for example, `_fooData`) to mark them as private. From this fact follow two recommendations.

- Don't use the underscore character as a prefix for your private methods. Apple reserves this convention.
- If you are subclassing a large Cocoa framework class (such as `NSView` or `UIView`) and you want to be absolutely sure that your private methods have names different from those in the superclass, you can add your own prefix to your private methods. The prefix should be as unique as possible, perhaps one based on your company or project and of the form "XX_". So if your project is called Byte Flogger, the prefix might be `BF_addObject:`

Although the advice to give private method names a prefix might seem to contradict the earlier claim that methods exist in the namespace of their class, the intent here is different: to prevent unintentional overriding of superclass private methods.

Naming Functions

Objective-C allows you to express behavior through functions as well as methods. You should use functions rather than, say, class methods, when the underlying object is always a singleton or when you are dealing with obviously functional subsystems.

Functions have some general naming rules that you should follow:

- Function names are formed like method names, but with a couple exceptions:
  - They start with the same prefix that you use for classes and constants.
  - The first letter of the word after the prefix is capitalized.
- Most function names start with verbs that describe the effect the function has:

```
NSHighlightRect
NSDeallocateObject
```

Functions that query properties have a further set of naming rules:

- If the function returns the property of its first argument, omit the verb.

  ```
  unsigned int NSEventMaskFromType(NSEventType type)
  float NSHeight(NSRect aRect)
  ```

- If the value is returned by reference, use "Get".

  ```
  const char *NSGetSizeAndAlignment(const char *typePtr, unsigned int *sizep,
   unsigned int *alignp)
  ```

- If the value returned is a boolean, the function should begin with an inflected verb.

  ```
  BOOL NSDecimalIsNotANumber(const NSDecimal *decimal)
  ```

Naming Properties and Data Types

This section describes the naming conventions for declared properties, instance variables, constants, notifications, and exceptions.

Declared Properties and Instance Variables

A declared property effectively declares accessor methods for a property, and so conventions for naming a declared property are broadly the same as those for naming accessor methods (see "Accessor Methods" (page 12)). If the property is expressed as a noun or a verb, the format is:

`@property (…) type nounOrVerb;`

For example:

```
@property (strong) NSString *title;
@property (assign) BOOL showsAlpha;
```

If the name of a declared property is expressed as an adjective, however, the property name omits the "is" prefix but specifies the conventional name for the get accessor, for example:

```
@property (assign, getter=isEditable) BOOL editable;
```

In many cases, when you use a declared property you also synthesize a corresponding instance variable.

Make sure the name of the instance variable concisely describes the attribute stored. Usually, you should not access instance variables directly; instead you should use accessor methods (you do access instance variables directly in `init` and `dealloc` methods). To help to signal this, prefix instance variable names with an underscore (_), for example:

```
@implementation MyClass {
    BOOL _showsTitle;
}
```

If you synthesize the instance variable using a declared property, specify the name of the instance variable in the `@synthesize` statement.

```
@implementation MyClass
@synthesize showsTitle=_showsTitle;
```

There are a few considerations to keep in mind when adding instance variables to a class:

- Avoid explicitly declaring public instance variables.

  Developers should concern themselves with an object's interface, not with the details of how it stores its data. You can avoid declaring instance variables explicitly by using declared properties and synthesizing the corresponding instance variable.

- If you need to declare an instance variable, explicitly declare it with either `@private` or `@protected`.

  If you expect that your class will be subclassed, and that these subclasses will require direct access to the data, use the `@protected` directive.

- If an instance variable is to be an accessible attribute of instances of the class, make sure you write accessor methods for it (when possible, use declared properties).

Constants

The rules for constants vary according to how the constant is created.

Enumerated constants

- Use enumerations for groups of related constants that have integer values.
- Enumerated constants *and* the typedef under which they are grouped follow the naming conventions for functions (see "Naming Functions" (page 18)). The following example comes from `NSMatrix.h`:

  ```
  typedef enum _NSMatrixMode {
      NSRadioModeMatrix           = 0,
      NSHighlightModeMatrix       = 1,
      NSListModeMatrix            = 2,
      NSTrackModeMatrix           = 3
  } NSMatrixMode;
  ```

  Note that the `typedef` tag (`_NSMatrixMode` in the above example) is unnecessary.

- You can create unnamed enumerations for things like bit masks, for example:

```
enum {
    NSBorderlessWindowMask      = 0,
    NSTitledWindowMask          = 1 << 0,
    NSClosableWindowMask        = 1 << 1,
    NSMiniaturizableWindowMask  = 1 << 2,
    NSResizableWindowMask       = 1 << 3

};
```

Constants created with const

- Use `const` to create constants for floating point values. You can use `const` to create an integer constant if the constant is unrelated to other constants; otherwise, use enumeration.
- The format for `const` constants is exemplified by the following declaration:

```
const float NSLightGray;
```

As with enumerated constants, the naming conventions are the same as for functions (see "Naming Functions" (page 18)).

Other types of constants

- In general, don't use the `#define` preprocessor command to create constants. For integer constants, use enumerations, and for floating point constants use the `const` qualifier, as described above.
- Use uppercase letters for symbols that the preprocessor evaluates in determining whether a block of code will be processed. For example:

```
#ifdef DEBUG
```

- Note that macros defined by the compiler have leading and trailing double underscore characters. For example:

```
__MACH__
```

- Define constants for strings used for such purposes as notification names and dictionary keys. By using string constants, you are ensuring that the compiler verifies the proper value is specified (that is, it performs spell checking). The Cocoa frameworks provide many examples of string constants, such as:

  ```
  APPKIT_EXTERN NSString *NSPrintCopies;
  ```

  The actual NSString value is assigned to the constant in an implementation file. (Note that the `APPKIT_EXTERN` macro evaluates to `extern` for Objective-C.)

Notifications and Exceptions

The names for notifications and exceptions follow similar rules. But both have their own recommended usage patterns.

Notifications

If a class has a delegate, most of its notifications will probably be received by the delegate through a defined delegate method. The names of these notifications should reflect the corresponding delegate method. For example, a delegate of the global `NSApplication` object is automatically registered to receive an `applicationDidBecomeActive:` message whenever the application posts an `NSApplicationDidBecomeActiveNotification`.

Notifications are identified by global `NSString` objects whose names are composed in this way:

```
[Name of associated class] + [Did | Will] + [UniquePartOfName] + Notification
```

For example:

```
NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification
```

Exceptions

Although you are free to use exceptions (that is, the mechanisms offered by the `NSException` class and related functions) for any purpose you choose, Cocoa reserves exceptions for programming errors such an array index being out of bounds. Cocoa does *not* use exceptions to handle regular, expected error conditions. For these cases, use returned values such as `nil`, `NULL`, `NO`, or error codes. For more details, see *Error Handling Programming Guide*.

Exceptions are identified by global `NSString` objects whose names are composed in this way:

```
[Prefix] + [UniquePartOfName] + Exception
```

The unique part of the name should run constituent words together and capitalize the first letter of each word. Here are some examples:

```
NSColorListIOException
NSColorListNotEditableException
NSDraggingException
NSFontUnavailableException
NSIllegalSelectorException
```

Acceptable Abbreviations and Acronyms

In general, you shouldn't abbreviate names when you design your programmatic interface (see "General Principles" (page 6)). However, the abbreviations listed below are either well established or have been used in the past, and so you may continue to use them. There are a couple of additional things to note about abbreviations:

- Abbreviations that duplicate forms long used in the standard C library—for example, "`alloc`" and "`getc`"—are permitted.
- You may use abbreviations more freely in argument names (for example, "`imageRep`", "`col`" (for "column"), "`obj`", and "`otherWin`").

Abbreviation	Meaning and comments
alloc	Allocate.
alt	Alternate.
app	Application. For example, NSApp the global application object. However, "application" is spelled out in delegate methods, notifications, and so on.
calc	Calculate.
dealloc	Deallocate.
func	Function.
horiz	Horizontal.
info	Information.
init	Initialize (for methods that initialize new objects).
int	Integer (in the context of a C `int`—for an `NSInteger` value, use `integer`).
max	Maximum.
min	Minimum.
msg	Message.

Abbreviation	Meaning and comments
nib	Interface Builder archive.
pboard	Pasteboard (but only in constants).
rect	Rectangle.
Rep	Representation (used in class name such as `NSBitmapImageRep`).
temp	Temporary.
vert	Vertical.

You may use abbreviations and acronyms that are common in the computer industry in place of the words they represent. Here are some of the better-known acronyms:

ASCII

PDF

XML

HTML

URL

RTF

HTTP

TIFF

JPG

PNG

GIF

LZW

ROM

RGB

CMYK

MIDI

FTP

Tips and Techniques for Framework Developers

Developers of frameworks have to be more careful than other developers in how they write their code. Many client applications could link in their framework and, because of this wide exposure, any deficiencies in the framework might be magnified throughout a system. The following items discuss programming techniques you can adopt to ensure the efficiency and integrity of your framework.

> **Note:** Some of these techniques are not limited to frameworks. You can productively apply them in application development.

Initialization

The following suggestions and recommendations cover framework initialization.

Class Initialization

The `initialize` class method gives you a place to have some code executed once, lazily, before any other method of the class is invoked. It is typically used to set the version numbers of classes (see "Versioning and Compatibility" (page 28)).

The runtime sends `initialize` to each class in an inheritance chain, even if it hasn't implemented it; thus it might invoke a class's `initialize` method more than once (if, for example, a subclass hasn't implemented it). Typically you want the initialization code to be executed only once. One way to ensure this happens is to use `dispatch_once()`:

```
+ (void)initialize {
    static dispatch_once_t onceToken = 0;
    dispatch_once(&onceToken, ^{
        // the initializing code
    }
}
```

Note: Because the runtime sends initialize to every class, it's possible that `initialize` will be called in the context of a subclass—if the subclass doesn't implement `initialize`, then the invocation will fall through to the superclass. If you specifically need to perform initialization within the context of the relevant class, you can perform the following check rather than using `dispatch_once()`:

```
if (self == [NSFoo class]) {
    // the initializing code
}
```

You should never invoke the `initialize` method explicitly. If you need to trigger the initialization, invoke some harmless method, for example:

```
[NSImage self];
```

Designated Initializers

A designated initializer is an `init` method of a class that invokes an `init` method of the superclass. (Other initializers invoke the `init` methods defined by the class.) Every public class should have one or more designated initializers. As examples of designated initializers there is `NSView`'s `initWithFrame:` and `NSResponder`'s `init` method. Where `init` methods are not meant to be overridden, as is the case with `NSString` and other abstract classes fronting class clusters, the subclass is expected to implement its own.

Designated initializers should be clearly identified because this information is important to those who want to subclass your class. A subclass can just override the designated initializer and all other initializers will work as designed.

When you implement a class of a framework, you often have to implement its archiving methods as well: `initWithCoder:` and `encodeWithCoder:`. Be careful not to do things in the initialization code path that doesn't happen when the object is unarchived. A good way to achieve this is to call a common routine from your designated initializers and `initWithCoder:` (which is a designated initializer itself) if your class implements archiving.

Error Detection During Initialization

A well-designed initialization method should complete the following steps to ensure the proper detection and propagation of errors:

1. Reassign self by invoking `super`'s designated initializer.

2. Check the returned value for `nil`, which indicates that some error occurred in the superclass initialization.
3. If an error occurs while initializing the current class, release the object and return `nil`.

Listing 1 (page 28) illustrates how you might do this.

Listing 1 Error detection during initialization

```
- (id)init {
    self = [super init];  // Call a designated initializer here.
    if (self != nil) {
        // Initialize object  ...
        if (someError) {
            [self release];
            self = nil;
        }
    }
    return self;
}
```

Versioning and Compatibility

When you add new classes or methods to your framework, it is not usually necessary to specify new version numbers for each new feature group. Developers typically perform (or should perform) Objective-C runtime checks such as `respondsToSelector:` to determine if a feature is available on a given system. These runtime tests are the preferred and most dynamic way to check for new features.

However, you can employ several techniques to make sure each new version of your framework are properly marked and made as compatible as possible with earlier versions.

Framework Version

When the presence of a new feature or bug fix isn't easily detectable with runtime tests, you should provide developers with some way to check for the change. One way to achieve this is to store the exact version number of the framework and make this number accessible to developers:

- Document the change (in a release note, for instance) under a version number.

- Set the current version number of your framework and provide some way to make it globally accessible. You might store the version number in your framework's information property list (`Info.plist`) and access it from there.

Keyed Archiving

If the objects of your framework need to be written to nib file, they must be able to archive themselves. You also need to archive any documents that use the archiving mechanisms to store document data.

You should consider the following issues about archiving:

- If a key is missing in an archive, asking for its value will return `nil`, `NULL`, `NO`, 0, or 0.0, depending on the type being asked for. Test for this return value to reduce the data that you write out. In addition, you can find out whether a key was written to the archive.
- Both the encode and decode methods can do things to ensure backwards compatibility. For instance, the encode method of a new version of a class might write new values using keys but can still write out older fields so that older versions of the class can still understand the object. In addition, decode methods might want to deal with missing values in some reasonable way to maintain some flexibility for future versions.
- A recommended naming convention for archive keys for framework classes is to begin with the prefix used for other API elements of the framework and then use the name of the instance variable. Just make sure that names cannot conflict with the names of any superclass or subclass.
- If you have a utility function that writes out a basic data type (in other words, a value that isn't an object), be sure to use a unique key. For example, if you have an "archiveRect" routine that archives a rectangle should take a key argument, and either use that; or, if it writes out multiple values (for instance, four floats), it should append its own unique bits to the provided key.
- Archiving bitfields as-is can be dangerous due to compiler and endianness dependencies. You should archive them only when, for performance reasons, a lot of bits need to be written out, many times. See "Bitfields" (page 31) for a suggestion.

Exceptions and Errors

Most Cocoa framework methods do not force developers to catch and handle exceptions. That is because exceptions are not raised as a normal part of execution, and are not typically used to communicate expected runtime or user errors. Examples of these errors include:

- File not found
- No such user
- Attempt to open a wrong type of document in an application

- Error in converting a string to a specified encoding

However, Cocoa does raise exceptions to indicate programming or logic errors such as the following:

- Array index out of bounds
- Attempt to mutate immutable objects
- Bad argument type

The expectation is that the developer will catch these kinds of errors during testing and address them before shipping the application; thus the application should not need to handle the exceptions at runtime. If an exception is raised and no part of the application catches it, the top-level default handler typically catches and reports the exception and execution then continues. Developers can choose to replace this default exception-catcher with one that gives more detail about what went wrong and offers the option to save data and quit the application.

Errors are another area where Cocoa frameworks differ from some other software libraries. Cocoa methods generally do not return error codes. In cases where there is one reasonable or likely reason for an error, the methods rely on a simple test of a boolean or object (`nil`/non-`nil`) returned value; the reasons for a `NO` or `nil` returned value are documented. You should not use error codes to indicate programming errors to be handled at runtime, but instead raise exceptions or in some cases simply log the error without raising an exception.

For instance, `NSDictionary`'s `objectForKey:` method either returns the found object or `nil` if it can't find the object. `NSArray`'s `objectAtIndex:` method can never return `nil` (except for the overriding general language convention that any message to `nil` results in a `nil` return), because an `NSArray` object cannot store `nil` values, and by definition any out-of-bounds access is a programming error that should result in an exception. Many `init` methods return `nil` when the object cannot be initialized with the parameters supplied.

In the small number of cases where a method has a valid need for multiple distinct error codes, it should specify them in a by-reference argument that returns either an error code, a localized error string, or some other information describing the error. For example, you might want to return the error as an `NSError` object; look at the `NSError.h` header file in Foundation for details. This argument might be in addition to a simpler `BOOL` or `nil` that is directly returned. The method should also observe the convention that all by-reference arguments are optional and thus allow the sender to pass `NULL` for the error-code argument if they do not wish to know about the error.

Framework Data

How you handle framework data has implications for performance, cross-platform compatibility, and other purposes. This section discusses techniques involving framework data.

Constant Data

For performance reasons, it is good to mark as constant as much framework data as possible because doing so reduces the size of the `__DATA` segment of the Mach-O binary. Global and static data that is not `const` ends up in the `__DATA` section of the `__DATA` segment. This kind of data takes up memory in every running instance of an application that uses the framework. Although an extra 500 bytes (for example) might not seem so bad, it might cause an increment in the number of pages required—an additional four kilobytes per application.

You should mark any data that is constant as `const`. If there are no `char *` pointers in the block, this will cause the data to land in the `__TEXT` segment (which makes it truly constant); otherwise it will stay in the `__DATA` segment but will not be written on (unless prebinding is not done or is violated by having to slide the binary at load time).

You should initialize static variables to ensure that they are merged into the `__data` section of the `__DATA` segment as opposed to the `__bss` section. If there is no obvious value to use for initialization, use 0, `NULL`, 0.0, or whatever is appropriate.

Bitfields

Using signed values for bitfields, especially one-bit bitfields, can result in undefined behavior if code assumes the value is a boolean. One-bit bitfields should always be unsigned. Because the only values that can be stored in such a bitfield are 0 and -1 (depending on the compiler implementation), comparing this bitfield to 1 is false. For example, if you come across something like this in your code:

```
BOOL isAttachment:1;
int startTracking:1;
```

You should change the type to `unsigned int`.

Another issue with bitfields is archiving. In general, you shouldn't write bitfields to disk or archives in the form they are in, as the format might be different when they are read again on another architecture, or on another compiler.

Memory Allocation

In framework code, the best course is to avoid allocating memory altogether, if you can help it. If you need a temporary buffer for some reason, it's usually better to use the stack than to allocate a buffer. However, stack is limited in size (usually 512 kilobytes altogether), so the decision to use the stack depends on the function and the size of the buffer you need. Typically if the buffer size is 1000 bytes (or `MAXPATHLEN`) or less, using the stack is acceptable.

One refinement is to start off using the stack, but switch to a `malloc`'ed buffer if the size requirements go beyond the stack buffer size. Listing 2 (page 32) presents a code snippet that does just that:

Listing 2 Allocation using both stack and malloc'ed buffer

```
#define STACKBUFSIZE (1000 / sizeof(YourElementType))
 YourElementType stackBuffer[STACKBUFSIZE];
 YourElementType *buf = stackBuffer;
 int capacity = STACKBUFSIZE;  // In terms of YourElementType
 int numElements = 0;  // In terms of YourElementType

while (1) {
    if (numElements > capacity) {  // Need more room
        int newCapacity = capacity * 2;  // Or whatever your growth algorithm is
        if (buf == stackBuffer) {  // Previously using stack; switch to allocated
 memory
            buf = malloc(newCapacity * sizeof(YourElementType));
            memmove(buf, stackBuffer, capacity * sizeof(YourElementType));
        } else {  // Was already using malloc; simply realloc
            buf = realloc(buf, newCapacity * sizeof(YourElementType));
        }
        capacity = newCapacity;
    }
    // ... use buf; increment numElements ...
  }
  // ...
  if (buf != stackBuffer) free(buf);
```

Object Comparison

You should be aware of an important difference between the generic object-comparison method `isEqual:` and the comparison methods that are associated with an object type, such as `isEqualToString:`. The `isEqual:` method allows you to pass arbitrary objects as arguments and returns `NO` if the objects aren't of the same class. Methods such as `isEqualToString:` and `isEqualToArray:` usually assume the argument is of the specified type (which is that of the receiver). They therefore do not perform type-checking and consequently they are faster but not as safe. For values retrieved from external sources, such as an application's information property list (`Info.plist`) or preferences, the use of `isEqual:` is preferred because it is safer; when the types are known, use `isEqualToString:` instead.

A further point about `isEqual:` is its connection to the `hash` method. One basic invariant for objects that are put in a hash-based Cocoa collection such as an `NSDictionary` or `NSSet` is that if `[A isEqual:B] == YES`, then `[A hash] == [B hash]`. So if you override `isEqual:` in your class, you should also override `hash` to preserve this invariant. By default `isEqual:` looks for pointer equality of each object's address, and `hash` returns a hash value based on each object's address, so this invariant holds.

Document Revision History

This table describes the changes to *Coding Guidelines for Cocoa*.

Date	Notes
2013-10-22	Updated information on +initialize.
2012-02-16	Noted that "_" is a suitable prefix for ivar names.
2010-05-05	Removed obsolete protocol information.
2006-04-04	Revised guidelines for instance variables and clarified implications of messages to nil. Changed title from "Coding Guidelines."
2005-07-07	Fixed bugs.
2004-07-23	Various bug fixes.
2003-04-28	First version of *Coding Guidelines*.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010