Cocoa向け
コーディング
ガイドライン

Developer

目次

リスト

コード作成のガイドライン（Cocoa）：はじめに

Cocoaフレームワークやプラグインその他、公開APIが付属するプログラム製品を開発する際には、一般のアプリケーションの場合とは異なる方針や規約にもとづいて進めなければなりません。こういった製品を主に利用するのはアプリケーション開発者なので、開発者が混乱しないようにプログラムインターフェイスを設計することが重要です。このような状況では、APIの命名規約が有用です。首尾一貫した明瞭なインターフェイスを提供できるからです。版管理、バイナリ互換性、エラー処理、メモリ管理など、フレームワークならではの（あるいはこの場合に特に重要な）プログラミング技法もあります。Cocoaの命名規約や、フレームワークに適用するべきプログラミング上の慣習も、これに関連する話題です。

この書類の構成

この資料では、大きく分けて2つの話題を扱います。ひとつはプログラムインターフェイスの命名規約に関する話題で、資料の大半を占めています。AppleがCocoaフレームワークに適用している規約と（若干の例外を除いて）同じものです。この話題について、次の各章に分けて解説します。

- “コード命名の基本事項”（6 ページ）
- “メソッドの命名”（12 ページ）
- “関数の命名”（20 ページ）
- “プロパティやデータ型の命名”（21 ページ）
- “使って構わない略語や頭字語”（26 ページ）

もうひとつはフレームワークのプログラミングに関する話題です（現時点では1つの章しかありません）。

- “フレームワーク開発者のためのヒントと技法”（28 ページ）

コード命名の基本事項

オブジェクト指向ソフトウェアライブラリの設計において、クラス、メソッド、関数、定数など、プログラムインターフェイスを構成する要素の名前づけは、問題があっても見落とされがちです。ここでは、Cocoaインターフェイスを構成する多くの要素に共通の命名規約について、いくつかの点を解説します。

一般原則

明確性

- できる限り明確かつ簡潔であることが大切ですが、簡潔すぎて明確性が損なわれないようにしてください。

コード例	注記
`insertObject: atIndex:`	適切。
`insert:at:`	不明確。何を挿入（insert）するのか？「at」は何を表すのか？
`removeObjectAtIndex:`	適切。
`removeObject:`	良好。引数で示されたオブジェクトを削除する旨が明確。
`remove:`	不明確。何を削除するのか？

- 一般に、「もの」の名前を省略せず、長い単語であってもそのまま綴ってください。

コード例	注記
`destinationSelection`	適切。
`destSel`	不明確。
`setBackgroundColor:`	適切。
`setBkgdColor:`	不明確。

広く知られている略語と思っても、実際にはそうでない可能性があります。特に、異なる文化圏に属し、母語が異なる開発者であれば、メソッド名や関数名が何を意味するか、すぐにはわからないかも知れません。

- しかしながら、古くから広く認識されている略語も多少はあります。“使って構わない略語や頭字語”（26 ページ）に挙げる略語は使っても構いません。

- 何通りにも解釈できるメソッド名など、曖昧なAPI名は避けてください。

コード例	注記
`sendPort`	ポートを送信するのか、送信に用いるポート番号を返すのか？
`displayName`	名前を表示するのか、画面に表示する名前（見出し）を返すのか？

一貫性

- Cocoaのプログラムインターフェイス全体で、一貫した名前を使ってください。自信がなければ、ヘッダファイルや参照資料で、どのような名前が使われているか確認するとよいでしょう。
- メソッドのポリモーフィズム（多態）を活用しているクラスの場合、一貫性が特に重要です。クラスが違っても、同じことをするメソッドは同じ名前にしなければなりません。

コード例	注記
`- (NSInteger)tag`	`NSView`、`NSCell`、`NSControl`に定義がある。
`- (void)setStringValue:(NSString *)`	Cocoaのさまざまなクラスに定義がある。

“メソッドの引数”（17 ページ）も参照してください。

自己言及の回避

- 自分自身が「何であるか」を表す名前は避けてください。

コード例	注記
`NSString`	適切。
`NSStringObject`	自己言及。

- ただし、マスク定数（ビット演算で組み合わせることができる定数）や通知名を表す定数は例外です。

コード例	注記
`NSUnderlineByWordMask`	適切。
`NSTableViewColumnDidMoveNotification`	適切。

プレフィックス

プログラムインターフェイスにおいて、プレフィックスは名前を構成する重要な要素です。ソフトウェアのどの機能領域に属する名前か、識別しやすくする働きがあります。ソフトウェアは通常、単一のフレームワークに、あるいは密接に関連するいくつかのフレームワーク（たとえばFoundationおよびApplication Kit）にパッケージ化されています。プレフィックスを使えば、サードパーティ開発者が定義したシンボルとAppleが定義したシンボル（あるいはAppleが提供する複数のフレームワークで定義されたシンボルどうし）が衝突することを回避できます。

- プレフィックスは形式が決まっています。2〜3字の英大文字から成り、アンダースコアや「サブプレフィックス」を含みません。以下に例をいくつか示します。

プレフィックス	これを用いるCocoaのフレームワーク
NS	Foundation
NS	Application Kit
AB	Address Book
IB	Interface Builder

- プレフィックスは、クラス、プロトコル、関数、定数、（`typedef`で定義する）構造体の名前に使います。メソッド名には*使わないでください*。メソッドは、これを定義するクラスによって区切られた名前空間に属します。また、構造体の要素名にも使わないでください。

用字に関する規約

API要素の名前づけには、用字に関していくつか簡単な規約があります。

- 複数の語から成る名前の場合、その一部として、あるいは区切り文字として、句読点類（アンダースコア、ダッシュなどを含む）を使わないでください。代わりに、各語の先頭を大文字にし、（`runTheWordsTogether`のように）続けて表記します。このような表記法をキャメルケースと言います。ただし次の点に注意してください。

- メソッド名については、先頭を小文字とし、2つめ以降の語の先頭を大文字にします。プレフィックスはつけません。

```
fileExistsAtPath:isDirectory:
```

  例外として、広く知られている頭字語で始まるメソッド名は、`TIFFRepresentation`（`NSImage`）のように先頭が大文字になります。

- 関数や定数の名前は、関連するクラスと同じプレフィックスをつけ、各語の先頭を大文字にします。

```
NSRunAlertPanel
NSCellDisabled
```

- メソッド名の先頭にアンダースコアをつけて非公開（private）である旨を表すことは避けてください（ただし*インスタンス変数*名の先頭をアンダースコアにすることは可）。この規約にもとづくメソッド名はAppleが予約しています。サードパーティがこのようなメソッド名を使うと、名前空間の衝突が起こる恐れがあります。既存の非公開メソッドを誤って独自メソッドでオーバーライドすると、予期しない結果が生じる可能性があるのです。非公開APIに関する規約については、“非公開メソッド”（18 ページ）を参照してください。

クラス名とプロトコル名

クラス名には、当該クラス（またはそのインスタンス）が何を表すか、何をおこなうか、を明確に表す名詞を用いなければなりません。さらに、適切なプレフィックスを使います（“プレフィックス”（8 ページ）を参照）。Foundationその他のアプリケーションフレームワークに属するクラスは、いずれもこの規約に従っています。ごく一部の例を挙げると、`NSString`、`NSDate`、`NSScanner`、`NSApplication`、`UIApplication`、`NSButton`、`UIButton`などがあります。

プロトコルには、動作をどのようにグループ化するか、にもとづいて名前をつけます。

- ほとんどの*プロトコル*は、特定のクラスに関連づけられていない、関連するメソッドをグループ化しています。このようなプロトコルは、クラスと混同されないような名前にしてください。一般に、動名詞（「...ing」）形の名前にするとよいでしょう。

`NSLocking`	適切。
`NSLock`	不適切（クラスの名前と混同しやすい）。

- 互いに関連しない多数のメソッドを、（いくつかの小さなプロトコルに分けず）まとめてグループ化するプロトコルもあります。このようなプロトコルは、その主たる表現であるクラスに関連づけられる傾向にあります。この場合、当該クラスと同じ名前をプロトコルにもつけてください。

  具体例として`NSObject`プロトコルがあります。任意のオブジェクトに対して、クラス階層のどの位置に当たるか問い合わせる、指定したメソッドを起動する、参照カウントを増減する、などといった一連のメソッドをグループ化するプロトコルです。こういったメソッドの主たる表現が`NSObject`クラスにあるので、プロトコル名もNSObjectとなっています。

ヘッダファイル

ヘッダファイルの名前が重要なのは、その内容を表すからです。

- **独立したクラスやプロトコルの宣言**。クラスやプロトコルがグループに属しない場合、その宣言を独立したファイルに記述し、当該クラス/プロトコル名と同じファイル名にします。

ヘッダファイル	宣言
`NSLocale.h`	`NSLocale`クラス。

- **関連するクラスやプロトコルの宣言**。関連するクラス、カテゴリ、プロトコルの宣言をグループ化する場合、主たるクラス、カテゴリ、プロトコルの名前と同じファイル名にします。

ヘッダファイル	宣言
`NSString.h`	`NSString`クラスおよび`NSMutableString`クラス。
`NSLock.h`	`NSLocking`プロトコルと、`NSLock`、`NSConditionLock`、`NSRecursiveLock`の各クラス。

- **フレームワークに属するヘッダファイルをすべてインクルードする旨の記述**。各フレームワークに対して、それと同じ名前のヘッダファイルを用意しなければなりません。ここには、当該フレームワークに属する公開ヘッダファイルを、すべてインクルードする旨記述します。

ヘッダファイル	フレームワーク
`Foundation.h`	`Foundation.framework`。

- **別のフレームワークに属するクラスにAPIを追加する旨の記述**。あるフレームワークに属するクラスと同じカテゴリの、別のフレームワークにメソッドを宣言する場合、元のクラス名に「Additions」を付加します。たとえばApplication Kitには`NSBundleAdditions.h`というヘッダファイルがあります。

- **関連する関数やデータ型**。関連する関数や定数、構造体その他のデータ型をグループ化する場合、`NSGraphics.h`（Application Kit）のように、適切な名前のヘッダファイルに記述します。

メソッドの命名

メソッドはプログラミングインターフェイスとして最もよく使われる要素なので、その名前づけには特に配慮する必要があります。以下、メソッドの名前づけについて、いくつかの観点から説明します。

一般規則

メソッドの命名に当たって留意するべき、一般的なガイドラインをいくつか示します。

- 名前の先頭は小文字とし、構成する各単語（2つめ以降）の先頭を大文字にします。プレフィックスはつけません。“用字に関する規約”（8 ページ）を参照してください。

  これには2つ例外があります。広く知られている頭字語（TIFF、PDFなど）は、先頭の単語であっても大文字で表します。また、非公開メソッドをグループ化し、非公開である旨を識別しやすいよう、プレフィックスをつけても構いません（“非公開メソッド”（18 ページ）を参照）。

- オブジェクトが実行するアクションを表すメソッドは、先頭を動詞とします。

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

  ただし「do」や「does」は、特段の意味を付加しない助動詞なので、メソッド名には使わないでください。また、動詞の前に副詞や形容詞を置かないでください。

- レシーバーの属性を返すメソッドは、その属性と同じ名前とします。出力引数を介して間接的に値を返す場合を除き、「get」をつける必要はありません。

`- (NSSize)cellSize;`	適切。
`- (NSSize)calcCellSize;`	不適切。
`- (NSSize)getCellSize;`	不適切。

  “アクセサメソッド”（13 ページ）も参照してください。

- 引数の前には必ずキーワードを置きます。

`- (void)sendAction:(SEL)aSelector to:(id)anObject forAllCells:(BOOL)flag;`	適切。
`- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;`	不適切。

- 引数の前に置く語（キーワード）は、当該引数の意味を説明するものとしてください。

`- (id)viewWithTag:(NSInteger)aTag;`	適切。
`- (id)taggedView:(int)aTag;`	不適切。

- あるメソッドを継承した上で、さらに特殊化したメソッドを定義する場合は、既存のメソッドの末尾にキーワードを追加した名前とします。

`- (id)initWithFrame:(CGRect)frameRect;`	`NSView`、`UIView`。
`- (id)initWithFrame:(NSRect)frameRect mode:(int)aMode cellClass:(Class)factoryId numberOfRows:(int)rowsHigh numberOfColumns:(int)colsWide;`	NSMatrix（NSViewのサブクラス）

- レシーバーの属性名と一致するキーワードを接続するために「and」をつける必要はありません。

`- (int)runModalForDirectory:(NSString *)path file:(NSString *)name types:(NSArray *)fileTypes;`	適切。
`- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(NSArray *)fileTypes;`	不適切。

この例では「and」があると（英語として）自然に読み下せるように見えますが、キーワードの数が多くなると問題が生じます。

- 独立した2つのアクションを実行するメソッドであれば、各アクションを表す語を「and」で接続した名前にしてください。

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

アクセサメソッド

アクセサメソッドとは、オブジェクトのプロパティ値を設定し、あるいは返すメソッドのことです。その名前は、プロパティの表現方法によって、推奨される形式が異なります。

- プロパティが名詞（noun）として表現される場合、次のような形式になります。

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

以下に例を示します。

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

- プロパティが形容詞（adjective）として表現される場合、次のような形式になります。

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

以下に例を示します。

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

- プロパティが動詞として表現される場合、次のような形式になります。

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

以下に例を示します。

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

動詞は単に現在形で表します。

- 分詞形にして形容詞的性質を持たせることは避けてください。

`– (void)setAcceptsGlyphInfo:(BOOL)flag;`	適切。
`– (BOOL)acceptsGlyphInfo;`	適切。
`– (void)setGlyphInfoAccepted:(BOOL)flag;`	不適切。
`– (BOOL)glyphInfoAccepted;`	不適切。

- 法助動詞（「can」、「should」、「will」など）を動詞の前に置いて意味を明確化しても構いませんが、「do」や「does」は使わないでください。

`- (void)setCanHide:(BOOL)flag;`	適切。
`- (BOOL)canHide;`	適切。
`- (void)setShouldCloseDocument:(BOOL)flag;`	適切。
`- (BOOL)shouldCloseDocument;`	適切。
`- (void)setDoesAcceptGlyphInfo:(BOOL)flag;`	不適切。
`- (BOOL)doesAcceptGlyphInfo;`	不適切。

- 「get」は、オブジェクトや値を、出力引数を介して間接的に返すメソッドにのみ使ってください。この形の名前は、複数の値を返すメソッドにのみ用います。

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

上記のようなメソッドでは、呼び出し元が入出力引数として`NULL`を渡すことにより、該当する値が必要ない旨を伝えることができるようにしてください。

デリゲートメソッド

デリゲートメソッド（あるいは委譲メソッド）とは、所定のイベントに応じて、オブジェクトがそのデリゲートで起動するメソッドのことです（デリゲートに実装されている場合）。その名前は特別な形式であり、これはあるオブジェクトのデータ源で起動されるあらゆるメソッドに等しく適用されます。

- メソッド名の先頭で、メッセージの送信元オブジェクトのクラスを表します。

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

クラス名からプレフィックスを除去し、先頭を小文字にしてください。

- クラス名にコロンを後置します（引数は委譲元オブジェクト）。ただしメソッドの引数が送信元だけである場合を除きます。

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

- ただし、通知に応じて起動されるメソッドは例外です。この場合、通知オブジェクトだけが引数となります。

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

- 何らかの事象が発生した、あるいは発生しようとしている旨をデリゲートに通知するためのメソッドは、名前に「did」や「will」をつけてください。

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

- 他のオブジェクトに代わって何らかのアクションを実行するよう、デリゲートに指示するためのメソッドは、名前に「did」や「will」をつけても構いませんが、「should」の方が望ましいでしょう。

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

コレクションメソッド

オブジェクト（要素）のコレクションを管理するオブジェクトについては、次のような形式のメソッド名を使う規約になっています。

```
- (void)add Element :( elementType ) anObj ;

- (void)remove Element :( elementType ) anObj ;

- (NSArray *) elements ;
```

以下に例を示します。

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

次のような状況では、上記の規約に対して修正や改善を施します。

- （規約の第3項）コレクションの要素間に、真に順序がない場合、NSArrayオブジェクトではなくNSSetオブジェクトを返します。
- （規約の第1項、第2項）コレクションにおける要素の「挿入位置」が重要であれば、上記のメソッドの代わりに、あるいはこれに加えて、次のようなメソッドを実装してください。

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

コレクションメソッドに関しては、いくつか実装詳細にも留意する必要があります。

- 上記のメソッドは通常、挿入したオブジェクトの所有権が呼び出し元にあると想定します。したがって、オブジェクト（要素）を追加/挿入する側で、当該オブジェクトを保持しなければなりません。同様に、オブジェクト（要素）をコレクションから削除した場合、当該オブジェクトを解放するのは呼び出し元の責任です。
- 挿入したオブジェクトが、主たるオブジェクトを指す逆ポインタを持つ必要がある場合、（通常は）この逆ポインタの設定のみおこなう（保持はしない）`set...`メソッドを使います。たとえば`insertLayoutManager:atIndex:`メソッドの場合、NSLayoutManagerクラスが以下のメソッドでこれをおこないます。

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

  通常、`setTextStorage:`を直接呼び出すことはありませんが、このメソッドをオーバーライドする必要があるかも知れません。

コレクションメソッドの規約に関する例としてもうひとつ、NSWindowクラスも示します。

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

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

メソッドの引数

メソッドの引数名についても、いくつか一般的な規約があります。

- メソッド名と同様に、引数名も先頭は小文字、2つめ以降の単語の先頭は大文字で表します（例：`removeObject:(id)anObject`）。

- 名前に「pointer」や「ptr」を使わないでください。ポインタか否かの区別は、引数名ではなく型でおこないます。
- 1〜2文字の短い引数名は避けてください。
- 数文字短くなる程度の略語も避けてください。

（Cocoaでは）従来から、次のようなキーワードと引数を組にして使うのが通例です。

```
...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
```

非公開メソッド

多くの場合、非公開メソッドの名前も、公開メソッドの場合と同じ規則に従います。もっとも、非公開メソッドにはプレフィックスをつけて、公開メソッドと識別しやすいようにするのが通常の規約です。その場合でも、非公開メソッドの名前が原因で、ある種の問題が生じる可能性があります。Cocoaフレームワークに属するクラスのサブクラスを設計する際、サブクラス側に定義した非公開メソッドが、フレームワーク側の同名の非公開メソッドを誤ってオーバーライドしていても、気がつかない恐れがあります。

Cocoaフレームワークの非公開メソッドの多くは、アンダースコアを前置する（例：`_fooData`）ことにより、非公開であることを示しています。したがって、次の2つの推奨事項に従ってください。

- 非公開メソッドを定義する際、プレフィックスとしてアンダースコアを使わないでください。このような名前はAppleが予約しています。
- Cocoaフレームワークの大規模なクラス（`NSView`や`UIView`など）から派生したサブクラスで、スーパークラス側の非公開メソッドと名前が衝突するのを確実に回避したければ、独自に定めたプレフィックスをつけてください。会社名やプロジェクト名にもとづき、「XX_」という形式で、できるだけ一意性を確保できるプレフィックスを選ぶとよいでしょう。たとえばプロジェクト名が「Byte Flogger」であれば、「`BF_addObject:`」などという名前になります。

この推奨事項は、メソッドがこれを定義するクラスによって区切られた名前空間に属する、とした先の主張と矛盾するように見えますが、ここでの意図は異なります。スーパークラスの非公開メソッドを誤ってオーバーライドするのを防ぐのが目的です。

関数の命名

Objective-Cでは、メソッドだけでなく関数でも動作を表現できます。クラスメソッドなどよりも関数が適している状況としては、基盤となるオブジェクトが常にシングルトンである場合や、明らかに機能本位のサブシステムを扱う場合が考えられます。

関数名は一般的な命名規則に従ってつけてください。

- 関数名はメソッド名と同様に考えてつけますが、例外が2つあります。
  - クラスや定数に用いるのと同じプレフィックスをつけてください。
  - プレフィックスに続く語の先頭は大文字で表します。
- 関数名の多くは、先頭にその関数の効能を表す動詞を置きます。

```
NSHighlightRect
NSDeallocateObject
```

プロパティを問い合わせる関数には、さらに次の規則も適用します。

- 第1引数で指定するオブジェクトのプロパティを返す関数の場合、動詞を省略します。

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

- 参照引数を介して値を返す関数には「Get」を使います。

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

- 戻り値がブール型であれば、関数名の先頭は語尾変化した動詞とします。

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

プロパティやデータ型の命名

ここでは、宣言して使うプロパティ、インスタンス変数、定数、通知、例外の命名規約について説明します。

宣言して用いるプロパティやインスタンス変数

宣言して用いるプロパティは、実質的にそのアクセサメソッドも宣言しているので、命名規約はアクセサメソッドの場合と基本的に同じです（“アクセサメソッド” （13 ページ）を参照）。プロパティ名が名詞や動詞として表現される場合、次のような形式になります。

```
@property (…) typenounOrVerb;
```

以下に例を示します。

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

一方、プロパティ名が形容詞として表現される場合、次のように「is」はつけませんが、対応するgetアクセサには、明示的に「is」で始まる慣例の名前を指定します。

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

多くの場合、プロパティを宣言すれば、対応するインスタンス変数も自動的に割り当てることになります。

インスタンス変数の名称は、内部的に格納する属性を的確に表現するものでなければなりません。通常、インスタンス変数は直接操作せず、アクセサメソッドを介しておこないます（ただし`init`メソッドと`dealloc`メソッドは、インスタンス変数を直接操作します）。このことが名前から窺えるよう、次のように、アンダースコア（_）で始まるインスタンス変数名にするとよいでしょう。

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

プロパティを宣言することによりインスタンス変数も割り当てる場合は、その名前を@synthesize文で明示的に指定します。

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

インスタンス変数をクラスに追加する際には、次の点に留意してください。

- 公開（public）のインスタンス変数を明示的に宣言することは避けてください。

  オブジェクトが保持するデータの具体的な格納方法ではなく、外部に対するインターフェイスに主眼を置いて実装することが大切です。インスタンス変数を明示的に宣言する代わりに、プロパティを宣言し、対応するインスタンス変数を割り当てる、という方法があります。
- インスタンス変数の宣言がどうしても必要であれば、@private（非公開）または@protected（限定公開）を明示的に指定します。

  サブクラスを定義し、その実装コード内で直接アクセスすることになる変数には、@protectedを指定してください。
- インスタンス変数を外部からアクセス可能な属性として用いる場合は、アクセサメソッドを実装してください（できればプロパティを宣言して自動合成）。

定数

定数の命名規約は、その定義方法によって異なります。

列挙型定数

- 関連する一連の定数に整数値を割り当てる場合、列挙型を利用して宣言します。
- 列挙型定数名と、その列挙型にtypedefで与える名前は、関数の命名規約に従います（"関数の命名" （20 ページ）を参照）。NSMatrix.hに記述されている例を以下に示します。

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

なお、typedefタグ（先の例の_NSMatrixMode）はなくても構いません。

- 名前なしの列挙を利用して、次のように、ビットマスクなどを定義できます。

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

};
```

constつきで宣言する定数

- 浮動小数点数の定数は、constつきで宣言することにより定義します。整数であっても、他の定数と関連しない独立した定数であれば、constつきで宣言する方法でもよいでしょう。そうでなければ列挙を利用してください。
- const定数の宣言は次のようになります。

```
const float NSLightGray;
```

列挙型定数と同様、関数の命名規約に従います（“関数の命名” （20 ページ）を参照）。

その他の種類の定数

- 一般に、定数の宣言に#defineプリプロセッサコマンドは使わないでください。上記のように、整数であれば列挙型、浮動小数点数であればconst修飾子を使います。
- あるひとかたまりのコードを活かすかどうか、プリプロセッサで判断するためのシンボルには、大文字のみから成る語を使ってください。以下に例を示します。

```
#ifdef DEBUG
```

- コンパイラが定義するマクロは、先頭と末尾にアンダースコアが2つついた名前です。以下に例を示します。

```
__MACH__
```

- 通知の名前や辞書のキーに用いる文字列を、定数として定義することもできます。こうしておけば、綴りの誤りがないかどうか、コンパイラが検査できます。Cocoaフレームワークには次のような文字列定数が多数あります。

```
APPKIT_EXTERN NSString *NSPrintCopies;
```

このNSString型変数には、実装ファイルで定数が代入されています（APPKIT_EXTERNマクロはObjective-Cのextern というキーワードに展開されます）。

通知と例外

通知と例外の命名規約はよく似ています。ただし、それぞれ独自の規約もあります。

通知

クラスにデリゲートがある場合、通知の多くは、定義済みのデリゲートメソッドを介して、デリゲート側に届くことになります。このような通知の名前は、対応するデリゲートメソッドを反映したものでなければなりません。たとえばNSApplicationというグローバルオブジェクトのデリゲートは、アプリケーションがNSApplicationDidBecomeActiveNotification通知を送出したとき applicationDidBecomeActive:メッセージが届くよう、あらかじめ登録済みになっています。

通知の識別には、次のような形式の文字列（NSStringのグローバルオブジェクト）を使います。

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

以下に例を示します。

```
NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification
```

例外

例外（`NSException`クラスおよび関連する関数が提供する機構）はさまざまな目的で利用できますが、Cocoaでは、配列の範囲を超えるインデックスなど、プログラミング上のエラーを検出するために例外を予約しています。通常でも起こりうるようなエラー状況を処理する目的では*使いません*。このような場合、`nil`、`NULL`、`NO`、エラーコードなどの戻り値を使うことになっています。詳しくは『*Error Handling Programming Guide*』を参照してください。

例外の識別には、次のような形式の文字列（`NSString`のグローバルオブジェクト）を使います。

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

「UniquePartOfName」の部分は、構成する語をそのまま連結し、各単語の先頭を大文字で表します。以下に例をいくつか示します。

```
NSColorListIOException
NSColorListNotEditableException
NSDraggingException
NSFontUnavailableException
NSIllegalSelectorException
```

使って構わない略語や頭字語

プログラムインターフェイスの設計において、通常、略語は使いません（“一般原則” （6 ページ）を参照）。ただし以下に示す略語は、広く定着している、あるいは先例があるため、使ってもよいことになっています。略語について、いくつか注意するべき事項を示します。

- 標準Cライブラリで古くから使われてきた略語（「`alloc`」、「`getc`」など）は、使っても構いません。
- 引数名にはより積極的に略語を使って構いません（「`imageRep`」、「`col`」（「column」の略）、「`obj`」、「`otherWin`」など）。

略語	意味と説明
alloc	allocate（割り当て）。
alt	alternate（代替）。
app	application（アプリケーション）。NSApp（グローバルアプリケーションオブジェクト）など。ただし、デリゲートメソッドや通知では、「application」とそのまま綴ります。
calc	calculate（計算）。
dealloc	deallocate（割り当て解除）。
func	function（関数）。
horiz	horizontal（水平）。
info	information（情報）。
init	initialize（初期化）。新規オブジェクトを初期化するメソッドなど。
int	integer（整数）。Cの`int`型を表す文脈で使用。`NSInteger`値については`integer`と表記。
max	maximum（最大）。
min	minimum（最小）。
msg	message（メッセージ）。

略語	意味と説明
nib	Interface Builderのアーカイブ（「NeXT Interface Builder」に由来）。
pboard	pasteboard（ペーストボード）。ただし定数でのみ。
rect	rectangle（矩形）。
Rep	representation（表現）。`NSBitmapImageRep`などのクラス名で使用。
temp	temporary（一時）。
vert	vertical（垂直）。

コンピュータ業界で広く使われている略語や頭字語は使っても構いません。このような頭字語のいくつかを以下に示します。

ASCII

PDF

XML

HTML

URL

RTF

HTTP

TIFF

JPG

PNG

GIF

LZW

ROM

RGB

CMYK

MIDI

FTP

フレームワーク開発者のためのヒントと技法

フレームワークの開発に当たっては、通常以上にコードの書き方に配慮しなければなりません。フレームワークは多くのクライアントアプリケーションに組み込まれることになるので、何らかの不備があるとシステム全体に影響が及びかねないからです。以下、フレームワークの効率性や整合性を確保するための、プログラミング技法について解説します。

> **注意:** フレームワークでなくても適用可能な技法がいくつかあるので、アプリケーション開発の際、積極的に活用してください。

初期化

フレームワークの初期化に関する提言や推奨事項を以下に示します。

クラスの初期化

`initialize`というクラスメソッドには、当該クラスの他のメソッドを実行する前に、1回だけ、必要が生じた時点で遅延実行するコードを記述します。典型的な用途として、クラスの版番号を設定する処理が考えられます（“版管理と互換性” （30 ページ）を参照）。

実行時ルーチンは継承チェーンをたどって、各クラスに`initialize`メッセージを送信します（該当するメソッドが実装されていなくても同様）。したがって、あるクラスの`initialize`メソッドが、複数回呼び出されることもありえます（サブクラス側に実装されていない場合など）。通常、初期化コードを実行するのは1回だけです。確実に1度だけ実行するための手段として、`dispatch_once()`があります。

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

注意: 実行時ルーチンはあらゆるクラスにinitializeメッセージを送信するので、サブクラスのコンテキストでinitializeが呼び出される可能性があります。サブクラスにinitializeが実装されていなければ、スーパークラスに向かい、継承チェーンをたどって順に起動を試みるからです。該当するクラスのコンテキスト内で初期化をおこなう特段の必要があれば、dispatch_once()を使うのではなく、次のように条件分岐を記述すればよいでしょう。

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

initializeメソッドを明示的に呼び出すことはできません。明示的な初期化が必要であれば、次のように無害な（特に何もしない）メソッドを呼び出してください。

```
[NSImage self];
```

指定イニシャライザ

指定イニシャライザとは、クラスのinitメソッドのうち、その中でスーパークラスのinitメソッドを起動するもののことです（当該クラスに定義されたinitメソッドを起動する、他のイニシャライザと区別します）。公開クラスには指定イニシャライザが少なくとも1つ必要です。例として、NSViewのinitWithFrame:メソッド、NSResponderのinitメソッドなどがあります。initメソッドがオーバーライドされることを想定していない場合（たとえば、クラスクラスタの外部向けインターフェイスとして使われる、NSStringその他の抽象クラス）、サブクラス側にそれぞれ初期化メソッドを実装することになるでしょう。

指定イニシャライザは、開発者が当該クラスのサブクラスを定義する際、他のイニシャライザと明確に区別できるようにしなければなりません。サブクラス側では、指定イニシャライザのみオーバーライドすれば、他のイニシャライザはすべて想定どおりに動作するはずです。

フレームワークに属するクラスには、アーカイブ処理メソッド（initWithCoder:、encodeWithCoder:など）も実装することが少なくありません。アーカイブからオブジェクトを復元する際には実行するべきでない処理を、初期化コードに記述しないよう注意してください。そのためには、アーカイブ処理を実装する場合、指定イニシャライザとinitWithCoder:（これ自身も指定イニシャライザ）の両方から共通のルーチンを呼び出す形で記述する、という方法がよいでしょう。

初期化中のエラー検出

初期化メソッドは、エラーが生じたとき適切に検出し、伝達できるよう、次の手順に従って処理を進めなければなりません。

1. superの指定イニシャライザを呼び出して、自分自身（self）を割り当て直します。
2. その戻り値がnil（スーパークラスの初期化処理でエラーが生じたことを表す）でないか検査します。
3. その後、サブクラス側の初期化処理を進めますが、エラーが発生した場合はオブジェクトを解放し、nilを返します。

リスト 1 （30 ページ）に、この手順にもとづくコードの記述を示します。

リスト 1　初期化中のエラー検出

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

版管理と互換性

フレームワークに新しいクラスやメソッドを追加する場合、一般に、新しい機能グループごとに新しい版番号を与える必要はありません。アプリケーション開発者は通常、respondsToSelector:などObjective-Cの実行時ルーチンに組み込まれた検査ルーチンを使って、ある機能が利用できるかどうか確認します。新しい機能が当該システムで利用できるかどうか調べるためには、このような実行時検査ルーチンで動的に調べる方法が望ましいでしょう。

一方、フレームワークの新しい版ごとにその旨の印をつけ、旧版との互換性をできるだけ維持するための技法がいくつかあります。

フレームワークの版

新機能やバグ修正の有無が実行時テストで簡単に調べられない場合、アプリケーション開発者向けに、何らかの代替手段を用意しなければなりません。そのひとつに、フレームワークの正しい版番号を保持し、開発者向けに公開する、という方法があります。

- 版番号ごとに、変更点を文書（リリースノートなど）にまとめてください。
- フレームワークの版番号を記述しておき、どこからでも実行できるアクセス手段を提供します。版番号を記述する場所としては、フレームワークの情報プロパティリスト（`Info.plist`）がよいでしょう。アプリケーションからはこれを介してアクセスできます。

キーつきのアーカイブ保存

フレームワークのオブジェクトをnibファイルに記述する必要がある場合、このオブジェクト自身をアーカイブ保存できるようにしなければなりません。さらに、アーカイブ機構を用いて中身のデータを保存する文書も、アーカイブ保存する必要があります。

アーカイブ保存については、次の事項を検討してください。

- アーカイブにキーが欠落している場合、値を問い合わせると、戻り値はその型に応じて、`nil`、`NULL`、`NO`、0、0.0のいずれかになります。戻り値がこういった値でないか検査することにより、記述するデータ量を削減できます。また、キーがアーカイブに書き込まれているかどうか確認することも可能です。
- エンコード/デコードをおこなうメソッドには、後方互換性を維持するための処理も実装できます。たとえば、あるクラスの新しいエンコード処理メソッドは、キーと値を組にして書き出す方式になっているとしましょう。この場合、旧版のクラスで使われていた要素も書き出すようにすれば、旧クラスが組み込まれたアプリケーションもなお、このアーカイブから情報を取得できます。さらに、デコード処理メソッドで、欠落している値を何らかの手段で埋め合わせるようにすれば、将来の版でも柔軟に対処できるかも知れません。
- フレームワークに属するクラスのアーカイブキーに関する命名規約では、他のAPI要素にも使われているプレフィックスと、インスタンス変数の名前を連結することになっています。これがスーパークラスやサブクラスの名前と衝突しないことを確認してください。
- 基本データ型（オブジェクト以外の型）の値を書き出すユーティリティ関数を使う場合、キーが一意的であるようにしてください。たとえば、矩形データをアーカイブ保存する「archiveRect」ルーチンで、引数でキーを指定するようになっている場合、この引数を使います。あるいは、複数の値（たとえば4つの浮動小数点数）を保存する場合、指定されたキーに短い字句を付加することにより、それぞれの値に対する一意的なキーを生成するという方法もあります。

- ビットフィールドをそのままアーカイブ保存する操作は、コンパイラの実装やエンディアンの違いによって、結果が異なる可能性があります。したがって、大量のビットを何度も書き出すなど、性能上の問題のためどうしても必要な場合に限定するべきでしょう。詳しくは“ビットフィールド”（34 ページ）を参照してください。

例外とエラー

Cocoaフレームワークのメソッドの多くは、例外を捕捉して処理するよう、開発者に強制しないようになっています。これは、通常の処理の流れで例外が発生することはないからです。想定内の実行時エラーや、ユーザの操作誤りを、例外機構を介して伝達することはありません。具体的には次のような状況です。

- ファイルが見つからない
- 該当するユーザが登録されていない
- 開こうとしている文書の種類が適切でない
- 文字列のエンコーディング変換時にエラーが発生した

一方、次のようなプログラミングエラーや論理エラーがあると例外が発生します。

- 配列の範囲外を指すインデックスが指定された
- 状態不変であるオブジェクトの状態を変更しようとした
- 不適切な型の引数が渡された

開発者はテスト期間中にこの種のエラーを捕捉し、すべて対処した上で出荷することになっています。したがって、（正式な）アプリケーションが、実行時に例外を処理する必要はないはずです。発生した例外を、アプリケーションのどの部分も捕捉しなかった場合、通常は最上位のデフォルトハンドラが例外を捕捉し、その旨を報告した上で処理を続行します。デフォルトの例外ハンドラを置き換えることにより、問題をより詳しく報告する、データを保存する機会を提供した上でアプリケーションを終了する、などといった動作に変更することも可能です。

Cocoaフレームワークはエラーの取り扱いに関しても、他のソフトウェアライブラリと異なる点があります。Cocoaのメソッドは一般に、エラーコードを返しません。エラーとして扱う合理的な理由があれば、戻り値として`nil`（ブール値）や`nil`オブジェクトを返すことによりその旨を通知します。`NO`や`nil`を返す場合の具体的な理由は、当該メソッドの仕様として文書に記載しておきます。プログラミングエラーを表すエラーコードを返し、実行時に対処させる、という使い方はしないでください。この場合、例外を投げるか、あるいは単にログとして記録する（例外は投げない）だけにとどめます。

たとえばNSDictionaryのobjectForKey:メソッドは、キーに対応するオブジェクトを返しますが、見つからなかった場合はnilを返します。NSArrayのobjectAtIndex:メソッドがnilを返すことはありません（ただし当該配列オブジェクトがnilであった場合を除く）。というのも、nil オブジェクトにnil値としてNSArrayを格納することはできない上に、配列の範囲外を指すインデックスが指定された場合、プログラミングエラーとして例外が発生するからです。また、initメソッドの多くは、指定された引数でオブジェクトを初期化できない場合、nilを返すようになっています。

状況に応じて何通りかのエラーコードを返すべき、正当な理由があるメソッドも少数ながら存在します。この場合、参照渡しの出力引数を使って、エラーコード、あるいはその内容を表す（ローカライズした）文字列など、必要な情報を返すようにしてください。たとえばエラーに関する情報を、NSErrorオブジェクトの形で返すとよい場合もあります。詳しくはFoundationのNSError.hヘッダファイルを参照してください。この出力引数を、戻り値のBOOL値やnilと併用してもよいでしょう。この方法は、呼び出し元がエラーの詳細を把握しなくても構わない場合、出力引数としてNULLを渡す、という規約にも適っています。

フレームワークのデータ

フレームワークのデータの扱い方によって、処理性能やプラットフォーム間の互換性その他に違いが生じます。以下、このようなデータを取り扱う技法について説明します。

定数データ

性能上の理由で、できるだけ多くのデータを定数として扱うのが得策です。これにより、Mach-Oバイナリの__DATAセグメントの容量を減らすことができるからです。const以外の大域データや静的データは、__DATAセグメントの__DATAセクションに収容されます。こういったデータは、当該フレームワークを組み込んだアプリケーションを起動する都度、一定量のメモリを消費します。たとえば500バイト増える程度ならば問題ないと思うかも知れません。しかしその結果、必要なページ数が増えれば、アプリケーションごとに4キロバイトも増加することになります。

定数データはすべてconstをつけて宣言してください。ブロック内にchar *ポインタがなければ、データは__TEXTセグメントに収容されます（したがって真に定数）。それ以外の場合はそのまま__DATAセグメントに収容されますが、やはり書き出しはできません（プリバインディング（システムフレームワークの関連づけ）が済んでいない、あるいはロード時にバイナリを移動しなければならないため、この関連づけが崩れている場合を除く）。

静的変数も確実に初期化することにより、__DATAセグメントの__bssセクションではなく、__dataセクションにマージされるようにしてください。初期値として自明な値がなければ、0、NULL、0.0など適当な値にします。

ビットフィールド

ビットフィールド（特に1ビットのみのビットフィールド）として符号つきの値を用いると、これをブール値として扱うコードがあれば、未定義の動作が起こるかも知れません。したがって、1ビットのビットフィールドは符号なしにしてください。このようなビットフィールドに格納できる値は、（コンパイラの実装にもよりますが）0と-1だけであり、これと1を比較した結果はfalseになります。たとえば次のようなコードがあったとしましょう。

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

この場合、`unsigned int`型に変更しなければなりません。

アーカイブ保存に関しても、ビットフィールドには問題があります。一般に、ビットフィールドをそのままの形でディスクやアーカイブに保存してはなりません。異なるアーキテクチャのデバイス上で、あるいは別のコンパイラで構築したアプリケーションで読み込もうとすると、形式が異なるため、正しく解釈されない恐れがあるからです。

メモリ割り当て

フレームワークのコードでは、できるだけメモリの割り当てを避けてください。何らかの理由で一時的にデータを保存する領域が必要になっても、ヒープから確保したバッファではなく、スタックを使う方がよいでしょう。ただしスタックには容量制限があるので（通常は合計512キロバイト）、機能や必要な容量を考慮して判断してください。一般に、必要な容量が1000バイト（あるいは`MAXPATHLEN`）以下であれば、スタックを推奨します。

より洗練された方法として、最初はスタックを用い、容量が増えてきたら`malloc`で割り当てたバッファに切り替える、というやり方があります。リスト 2 （34 ページ）に、この方針に基づくコード例を示します。

リスト 2　スタックとヒープを使い分けるコード例

```
#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);
```

オブジェクトの比較

汎用のオブジェクト比較メソッドである`isEqual:`と、特定の型のオブジェクト比較に用いる`isEqualToString:`などには、重要な違いがあることを認識しなければなりません。`isEqual:`メソッドには引数として任意のオブジェクトを渡すことができます。属するクラスが異なる場合、戻り値は`NO`になります。一方、`isEqualToString:`や`isEqualToArray:`などのメソッドは通常、引数が所定の型（レシーバーの型と同じ）であると想定して比較します。したがって型検査をおこなわないので、高速に処理できる代わり安全性は損なわれます。アプリケーションの情報プロパティリスト（`Info.plist`）や環境設定など、外部情報源から取得した値の場合、安全性を考慮して`isEqual:`を使うよう推奨します。型がわかっていれば`isEqualToString:`などで比較して構いません。

`isEqual:`についてはもうひとつ、`hash`メソッドとの関連にも注意が必要です。`NSDictionary`や`NSSet`など、ハッシュを基盤とするCocoaコレクションに収容されるオブジェクトには、`[A isEqual:B] == YES`ならば`[A hash] == [B hash]`である、という基本的な不変表明が成り立ちます。したがって、あるクラスで`isEqual:`をオーバーライドする場合、`hash`もオーバーライドして、この不変表明が崩れないようにしなければなりません。`isEqual:`のデフォルトの実装では、各オブジェクトのアドレス（ポインタ）を比較し、一方`hash`も、各オブジェクトのアドレスからハッシュ値を求めて返すようになっているので、この不変表明が成り立ちます。

書類の改訂履歴

この表は「*Cocoa向け コーディングガイドライン*」の改訂履歴です。

日付	メモ
2013-10-22	+initializeに関する説明を改訂しました。
2012-02-16	ivar名に適した接頭辞は「_」であるという注釈を付記しました。
2010-05-05	古いプロトコル情報を削除しました。
2006-04-04	インスタンス変数のガイドラインを改訂し、nilへのメッセージの意味を明確にしました。『コーディングのガイドライン』からドキュメント名を変更しました。
2005-07-07	バグを修正しました。
2004-07-23	いくつかのバグを修正しました。
2003-04-28	『*コーディングのガイドライン*』の初版。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

Apple Japan
〒106-6140 東京都港区六本木 6
丁目10番1号 六本木ヒルズ
http://www.apple.com/jp

Offline copy. Trademarks go here.