Cocoa向け コーディング ガイドライン

Developer

目次

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

リスト

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

Cocoa向け コーディング ガイドラインの概要

公開APIを使用したCocoaフレームワーク、プラグイン、またはその他の実行ファイルの開発には、アプリケーション開発で用いられるのとは異なるアプローチと規則が要求されます。プロダクトの主要なクライアントは開発者です。このため、開発者がプログラムインターフェイスで混乱しないようにする必要があります。API命名規約が役立つ場面です。このような命名規約により、インターフェイスの一貫性と明瞭性を維持しやすくなるためです。バージョン管理やバイナリ互換、エラー処理、メモリ管理など、フレームワークに固有の、あるいはフレームワークではさらに重要性を増すプログラミング手法もあります。ここでは、Cocoa命名規則とフレームワークに推奨されるプログラミング慣例に関する情報をとりあげます。

この書類の構成

本トピック内の記事は、大きく2種類に分けられます。最初に紹介するのは、プログラムインターフェイスの命名規約を扱ったグループで、ここに大半の記事が含まれます。Appleが同社独自のCocoaフレームワークの記述に使用する規約と同一のものです（一部マイナーな例外を含む）。命名規約に関する記事の一部を、以下に挙げます。

2番目のグループ（現在、権限レベル1）では、フレームワークのプログラミングの側面を扱っています。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

命名の基本事項

オブジェクト指向型ソフトウェアライブラリの設計で見過ごされることの多い側面に、クラス、メソッド、関数、定数、およびプログラムインターフェイスのその他の要素の命名があります。ここでは、Cocoaインターフェイスの大半の項目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

明瞭性

- 可能な限り明瞭かつ簡潔な名前を付けることが重要ですが、簡潔さを重視して明瞭さが損なわれないようにしてください。

コード	コメント
`insertObject: atIndex:`	良。
`insert:at:`	不明瞭。何が挿入されるのか？「at」は何を意味するのか？
`removeObjectAtIndex:`	良。
`removeObject:`	良。引数で引用されるオブジェクトが除かれているため。
`remove:`	不明瞭。何が除かれるか？

- 原則として、要素の名称は略さないでください。長い単語であってもスペルアウトしてください。

コード	コメント
`destinationSelection`	良。
`destSel`	不明瞭。
`setBackgroundColor:`	良。
`setBkgdColor:`	不明瞭。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

略語が広く認識されていると考えたようですが、認識されていない可能性があります。特に作成されたメソッド名または関数名を目にする開発者が、異なる文化および言語背景を持つ場合はその可能性があります。

- ただし、一部のわずかな略語は実際に共通に認識されており、長期的に使用されています。そのような略語は今後も使用できます。「使用できる略語と頭字語」（26 ページ）を参照してください。

- 複数の解釈が可能なメソッド名など、不明瞭なAPI名は付けないでください。

コード	コメント
`sendPort`	ポートを送信するのですか、それともポートを返すのですか？
`displayName`	ユーザインターフェイスで名前を表示するのですか、それともレシーバのタイトルを返すのですか？

一貫性

- Cocoaプログラムインターフェイスを通じて、一貫した名前を使用するようにしてください。一貫性が確実に把握できない場合は、現在のヘッダファイルまたは参照ドキュメントで、前例を参照してください。
- 一貫性は、クラスのメソッドでポリモーフィズムが活用される場合に特に重要になります。別のクラスで同じ振る舞いをするメソッドは、同じ名前を付ける必要があります。

コード	コメント
`- (NSInteger)tag`	`NSView`、`NSCell`、`NSControl`で定義されます。
`- (void)setStringValue:(NSString *)`	複数のCocoaクラスで定義されます。

「メソッドの引数」（17 ページ）も参照してください。

自己参照なし

- 名前は自己参照になりません。

コード	コメント
`NSString`	OK。
`NSStringObject`	自己参照。

- マスクの定数（したがってビット演算で結合が可能な定数）は、通知名の定数と同様にこのルールの例外です。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

コード	コメント
`NSUnderlineByWordMask`	OK。
`NSTableViewColumnDidMoveNotification`	OK。

プレフィックス

プレフィックスは、プログラムインターフェイスの名前の重要な部分です。ソフトウェアの機能領域を差別化します。通常、このようなソフトウェアは、フレームワークまたは（FoundationとApplication Kitの場合と同様に）密接な関連性を持つフレームワーク内にパッケージ化されています。プレフィックスは、サードパーティの開発者で定義されたシンボルとAppleで定義されたシンボル間（およびApple独自のフレームワークのシンボル同士）の衝突を防ぎます。

- プレフィックスは所定の形式が使用されます。2、3個の大文字から成り、アンダースコアや「サブプレフィックス」を含みません。次に、いくつかの例を示します。

プレフィックス	Cocoaフレームワーク
NS	Foundation
NS	Application Kit
AB	Address Book
IB	Interface Builder

- クラス、プロトコル、関数、定数、`typedef`構造体の名前を付ける場合、プレフィックスを使用します。メソッド名を付ける場合は、プレフィックスを*使用しない*でください。メソッドは、メソッドを定義するクラスで作成された名前空間に存在するためです。また、構造体のフィールドの命名にプレフィックスを使用しないでください。

表記規約

API要素の命名時には、いくつかの簡単な表記規約に従ってください。

- 複数の語句を含む名前の場合、名前の一部またはセパレータ（アンダースコア、ダッシュなど）としてピリオドを使用しないでください。各単語の先頭文字を大文字表記し、語句を続けて表記します（例: `runTheWordsTogether`）。これはキャメルケースと呼ばれます。ただし、以下の制限に注意してください。

- メソッド名については、先頭に小文字を使用し、組み込まれた単語の先頭文字を大文字で表記します。プレフィックスを使用しないでください。

```
fileExistsAtPath:isDirectory:
```

  このガイドラインの例外は、`TIFFRepresentation`などの広く知られている頭字語で始まるメソッド名です（`NSImage`）。

- 関数と定数の名前は、関連するクラスと同じプレフィックスを使用し、組み込まれた単語の先頭文字を大文字で表記します。

```
NSRunAlertPanel
NSCellDisabled
```

- メソッド名に、プライベートを意味するプレフィックスとしてアンダースコア文字を使用することは避けてください（*インスタンス変数*名に、プレフィックスとしてアンダースコア文字を使用することは可能です）。この命名規約は、Appleにより使用されるものとして予約されています。サードパーティにより使用された場合、名前空間の衝突が起こる可能性があります。既存のプライベートのメソッドを、独自のメソッドで無意識のうちに上書きしてしまい、多大な結果を引き起こしてしまうのです。プライベートAPIに関しての規約の指示については、「プライベートメソッド」（18 ページ）を参照してください。

クラス名とプロトコル名

クラスの名称には、クラス（またはクラスのオブジェクト）が何を表すか、あるいは何を実行するかを明確に示す名詞を含める必要があります。名前には適切なプレフィックスを含めます（「プレフィックス」（8 ページ）を参照）。Foundationとアプリケーションのフレームワークに多くの例が示されています。一例として`NSString`、`NSDate`、`NSScanner`、`NSApplication`、`UIApplication`、`NSButton`、`UIButton`などが挙げられます。

プロトコルは、動作のグループ化の方法に従って名前を付けます。

- ほとんどの*プロトコル*は、特定のクラスに関連付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。この種のプロトコルは、プロトコルがクラスと混乱されることのないように名前を付ける必要があります。共通の規則として、動名詞（「...ing」）形を使用します。

`NSLocking`	良。
`NSLock`	不良（クラスの名前と受け取られる）。

© 2003, 2012 Apple Inc. All Rights Reserved.

- 一部のプロトコルは、複数の無関係なメソッドをグループ化しています（少数から成るプロトコルのグループを個別に数多く作成しません）。このようなプロトコルは、プロトコルの主要な表現である特定のクラスに関連付けられる傾向にあります。このようなケースでは、規約によりプロトコルにクラスと同じ名前が割り当てられます。

  この種のプロトコルの例として、`NSObject`プロトコルが挙げられます。このプロトコルは、クラス階層内の位置に関して任意のオブジェクトに照会し、オブジェクトから特定のメソッドを呼び出し、参照カウントをインクリメントまたはデクリメントする場合に使用するメソッドをグループ化します。これらのメソッドの主要な表現は`NSObject`クラスから提供されるため、プロトコルはクラスにちなんで名付けられています。

ヘッダファイル

ヘッダファイル名の付け方が重要になるのは、使用する規約によりファイルの内容が示されるためです。

- **独立したクラスまたはプロトコルの宣言**。クラスまたはプロトコルがグループに所属しない場合、宣言されたクラスまたはプロトコルを名前に持つ独立したファイルに宣言を含めます。

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

- **関連するクラスとプロトコルの宣言**。関連する宣言のグループに対して（クラス、カテゴリ、プロトコル）、主要なクラス、カテゴリ、またはプロトコルの名前を持つファイルに宣言を含めます。

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

- **フレームヘッダファイルのインクルード**。各フレームワークに、フレームワークのすべてのパブリックヘッダファイルを含む、フレームワークにちなんだ名前の付いたヘッダファイルを作成する必要があります。

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

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

- **別のフレームワークのクラスへのAPIの追加**。別のフレームワークのクラスのカテゴリに含まれるフレームワークでメソッドを宣言する場合、元のクラスの名前に「Additions」を付加します。この例として、Application Kitの`NSBundleAdditions.h`ヘッダファイルが挙げられます。
- **関連する関数とデータ型**。関連する関数、定数、構造体、その他のデータ型をグループ化している場合、`NSGraphics.h`（Application Kit）など、適切な名前のヘッダファイルに関連する要素を含めます。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

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で最も共通の要素と考えられるため、その命名方法には特別に配慮する必要があります。ここでは、メソッドの命名に関する以下のような側面を説明します。

一般規則

メソッドの命名の際に留意すべき、いくつかの一般的なガイドラインを説明します。

- 名前は先頭に小文字を使用し、組み込まれた単語の先頭文字を大文字で表記します。プレフィックスを使用しないでください。See 「表記規約」 （8 ページ）を参照してください。

  このガイドラインには2つの例外があります。周知された頭字語（TIFFやPDFなど）を大文字で、メソッド名の先頭に使用できます。またプライベートメソッドのグループ化と識別にプレフィックスを使用できます（「プライベートメソッド」 （18 ページ）を参照してください）。

- オブジェクトで実行されるアクションを表すメソッドについては、動詞を名前の先頭に使用します。

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

  「do」または「does」は、意味が付加されることが稀な補助動詞であるため、名前に使用しないでください。また動詞の前に副詞と形容詞を使用しないでください。

- メソッドがレシーバの属性を返す場合、属性にちなんだ名前をメソッドに付けてください。1つまたは複数の値が間接的に返される場合を除き、「get」は使用する必要がありません。

`- (NSSize)cellSize;`	正。
`- (NSSize)calcCellSize;`	誤。
`- (NSSize)getCellSize;`	誤。

  「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使用してください。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

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

- 引数の前の単語を使って、引数を説明してください。

`- (id)viewWithTag:(NSInteger)aTag;`	正。
`- (id)taggedView:(int)aTag;`	誤。

- 継承されたメソッドよりも、よりぐらい的なメソッドを作成する場合、既存のメソッドの末尾に新しいキーワードを追加してください。

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

- レシーバの属性であるキーワードの接続に、「and」を使用しないでください。

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

「and」はこの例では問題がないように思われますが、多くのキーワードを使ってメソッドを作成しているため、問題を引き起こします。

- メソッドが2つの個別のアクションを説明している場合、「and」を使用して接続してください。

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

アクセサメソッド

アクセサメソッドは、オブジェクトのプロパティの値を設定し、返すメソッドです。プロパティの表現方式に応じて、推奨される一定の形式があります。

- プロパティが名詞として表現される場合、以下のような形式になります。

  `-（型）名詞；`

```
- (void)set名詞:(型)aNoun;
```

例を示します。

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

- プロパティが形容詞として表現される場合、以下のような形式になります。

```
- (BOOL)is形容詞;
- (void)set形容詞:(BOOL)flag;
```

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、以下のような形式になります。

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

例を示します。

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

動詞はシンプルな現在形で表します。

- 分詞を使用して動詞を形容詞化しないでください。

`- (void)setAcceptsGlyphInfo:(BOOL)flag;`	正。
`- (BOOL)acceptsGlyphInfo;`	正。
`- (void)setGlyphInfoAccepted:(BOOL)flag;`	誤。
`- (BOOL)glyphInfoAccepted;`	誤。

- 意味を明瞭化するために法動詞（「can」、「should」、「will」などが先行する動詞）を使用できますが、「do」または「does」を使用しないでください。

`- (void)setCanHide:(BOOL)flag;`	正。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

`- (BOOL)canHide;`	正。
`- (void)setShouldCloseDocument:(BOOL)flag;`	正。
`- (BOOL)shouldCloseDocument;`	正。
`- (void)setDoesAcceptGlyphInfo:(BOOL)flag;`	誤。
`- (BOOL)doesAcceptGlyphInfo;`	誤。

- 「get」は、オブジェクトと値を間接的に返すメソッドにのみ使用してください。この形は、複数の項目の戻り値が要求される場合にのみメソッドに使用できます。

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

上記のようなメソッドでは、発信者が1つまたは複数の戻り値に関心がないことを示すものとして、実装でこのようなin-outパラメータに対して`NULL`を受け付ける必要があります。

デリゲートメソッド

デリゲートメソッド（またはデリゲーションメソッド）は、特定のイベントが起こった場合に、オブジェクトがそのデリゲートで呼び出すメソッドです（デリゲートがデリゲートメソッドを実装している場合）。オブジェクトのデータソースで呼び出されるメソッドに同等に使用される、独自の形式をとります。

- 名前の先頭で、メッセージを送信しているオブジェクトのクラスを特定します。

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

クラス名はプレフィックスを省略し、先頭は小文字表記です。

- メソッドが送信者である1つの引数のみ使用する場合を除き、クラス名にコロンを添えます（引数はデリゲート側オブジェクトの参照）。

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

- この例外は、通知が投稿された結果呼び出されるメソッドです。この場合、唯一の引数となるのは通知オブジェクトです。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

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

- 何らかのアクションが起こる、または起こりそうであるとデリゲートに通知するために呼び出されるメソッドには、「did」または「will」を使用します。

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

- 別のオブジェクトの代理でデリゲートにアクションを依頼するために呼び出されるメソッドに「did」または「will」を使用できますが、「should」が優先されます。

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

コレクションメソッド

オブジェクトのコレクションを管理するオブジェクト（それぞれ、コレクションの要素と呼ばれる）については、規約により以下の形式のメソッドを使用します。

```
- (void)add要素:(elementType)anObj;

- (void)remove要素:(elementType)anObj;

- (NSArray *)要素;
```

例を示します。

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

このガイドラインの制約および改訂を以下で説明します。

- 実際にコレクションに順序がない場合、NSArrayオブジェクトではなくNSSetオブジェクトを返します。
- コレクションの特定の位置に要素を挿入することが重要であれば、上記以外に、または上記に代えて以下のようなメソッドを使用します。

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

コレクションメソッドの操作の際には、いくつかの実装の詳細に留意する必要があります。

- これらのメソッドは、通常は挿入されるオブジェクトの所有権を暗示するため、メソッドを追加または挿入するコード内ではメソッドを保持し、削除するコードではメソッドを解放する必要があります。
- 挿入されるオブジェクトがポインタで元のメインオブジェクトを示す必要がある場合、通常は、バックポインタを設定する`set...`メソッドでこの操作を行います。
  `insertLayoutManager:atIndex:`メソッドの場合、NSLayoutManagerクラスは以下のメソッドで同じ操作を実行します。

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

  通常は`setTextStorage:`を直接呼び出しませんが、このメソッドの上書きが必要になる場合があります。

コレクションメソッドに関する上記の規約については、NSWindowクラスの例もとりあげます。

```
- (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

メソッドの引数の名前に関して、いくつかの一般的なルールがあります。

- メソッドと同様に、引数も先頭は小文字です。後続の単語の先頭文字は大文字で表記します（例: `removeObject:(id)anObject`）。
- 名前に「pointer」または「ptr」を使用しないでください。引数の名前ではなく型を使って、ポインタかどうかを宣言してください。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

- 引数に1文字および2文字の名前を使うことは避けてください。
- 数文字のみ省略する略語の使用も避けてください。

従来から（Cocoaでは）、以下のキーワードと引数が同時に使用されています。

```
...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
```

プライベートメソッド

ほとんどの場合、プライベートメソッドの名前はパブリックメソッドの名前と同じルールに従うのが通例でした。ただし、共通の規約として、プライベートメソッドはプレフィックスを使用するため、パブリックメソッドとの区別は簡単です。このような規約が適用される場合でも、プライベートメソッドの名前により特定の種類の問題が生じる可能性があります。Cocoaフレームワークのクラスのサブクラスを設計する場合、プライベートメソッドが、同じ名前を持つプライベートフレームワークメソッドを意図せずに上書きする恐れがないかどうかを把握することができません。

Cocoaフレームワークのほとんどのプライベートメソッドは、名前のプレフィックスにアンダースコアを使って（例: `_fooData`）プライベートであることを示しています。この事実に基づき、2つの推奨事項に従ってください。

- 作成するプライベートメソッドのプレフィックスに、アンダースコアを使用しないでください。この命名規約は、Appleによって予約されています。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

- 大規模なCocoaフレームワークのクラス（`NSView`や`UIView`など）をサブクラス化し、プライベートメソッドの名前をスーパークラス内の名前と確実に区別する必要がある場合は、プライベートメソッドに固有のプレフィックスを追加します。このプレフィックスは、会社やプロジェクトなどをベースに、「XX_」を使用して可能な限り独自のものを使用します。プロジェクト名がByte Floggerであれば、プレフィックスは`BF_addObject:`などになります。

プライベート名にプレフィックスを付ける奨励は、メソッドがクラスの名前空間に存在するという前記の主張と矛盾するようですが、ここでの意図は異なります。スーパークラスのプライベートメソッドを意図せずに上書きするのを防ぐことが目的です。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

関数の命名

Objective-Cを使用すると、メソッド以外に関数を通じて動作を表現できます。基本のオブジェクトが常にシングルトンである場合、または独立した機能を持つサブシステムを記述する場合、クラスメソッドなどではなく、関数を使用します。

関数のいくつかの一般的な命名規則に従う必要があります。

- 関数名はメソッド名と同様に形成されますが、いくつかの例外があります。
  - クラスと定数に使用するのと同じプレフィックスを先頭に配置します。
  - プレフィックスの後の単語の先頭文字は大文字で表記します。
- ほとんどの関数名は、関数の処理を記述した動詞から開始します。

```
NSHighlightRect
NSDeallocateObject
```

プロパティを照会する関数には、さらにいくつかの命名規則が課せられます。

- 関数が最初の引数のプロパティを返す場合、動詞を省略します。

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

- 値が参照で返される場合、「Get」を使用します。

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

- 返される値がブール値である場合、関数は影響を受ける動詞で始まります。

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

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

プロパティとデータ型の命名

ここでは、プロパティ、インスタンス変数、定数、通知、および例外に関する命名規約について説明します。

プロパティ(Declared Property)とインスタンス変数

プロパティの宣言はそのアクセサメソッドも同時に定義する事になるので、プロパティの命名に関する規約は概ねアクセサメソッドの命名規約と同じです（「アクセサメソッド」（13 ページ）を参照）。プロパティが名詞または動詞として表現される場合、以下のような形式になります。

```
@property (…) 型nounOrVerb;
```

たとえば次のように宣言します。

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

プロパティの名前が形容詞として表現される場合、以下のように、プロパティ名から「is」プレフィックスが省略されますが、getアクセサの従来の名称は指定されています。

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

多くのケースで、プロパティを使用する場合、対応するインスタンス変数を合成しています。

インスタンス変数の名称が内部の属性を正確に表していることを確認してください。通常、インスタンス変数に直接アクセスすることはなく、アクセサメソッドを使用します（`init`メソッドと`dealloc`メソッドでは直接インスタンス変数にアクセスします）。これを効果的にシグナリングするために、以下のようにインスタンス変数名の接頭辞にアンダースコア（_）を使います。

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

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

プロパティを使用してインスタンス変数を合成する場合、`@synthesize`ステートメントでインスタンス変数名を指定します。

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

インスタンス変数をクラスに追加する際に、留意すべき事項がいくつかあります。

- パブリック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。

  開発者はオブジェクトデータの格納方法の詳細ではなく、オブジェクトのインターフェイスに配慮する必要があります。インスタンス変数の明示的な宣言を避けるためには、プロパティを使用し、対応するインスタンス変数を合成します。

- インスタンス変数を宣言する必要がある場合、`@private`または`@protected`で明示的に宣言します。

  クラスがサブクラス化され、このようなサブクラスがデータへの直接のアクセスを要求することが予想される場合、`@protected`ディレクティブを使用します。

- インスタンス変数がクラスのインスタンスのアクセス可能な属性になる場合、そのアクセサメソッドを必ず記述してください（可能であれば、プロパティを使用してください）。

定数

定数のルールは、定数がどのように作成されるかに応じて変わります。

列挙型定数

- 整数値をとる関連定数のグループには、列挙を使用します。
- 列挙型定数と列挙型定数がグループされるtypedefは、関数の命名規約に従います（「関数の命名」（20 ページ）を参照してください）。`NSMatrix.h`を使った例を以下に示します。

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

typedefタグ（上記の例では_NSMatrixMode）は不要であることに注意してください。

- ビットマスクなどには、以下のように名前のない列挙を作成できます。

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

};
```

constで作成された定数

- constを使用して、浮動小数点値の定数を作成します。定数が他の定数と関連性がない場合、constを使用して整数定数を作成できます。それ以外の場合は、列挙を使用します。
- const定数の形式の例示を以下の宣言で示します。

```
const float NSLightGray;
```

列挙された定数と同様に、命名規約は関数と同じです（「関数の命名」 （20 ページ）を参照してください）。

その他の種類の定数

- 原則として、#defineプリプロセッサコマンドを定数の作成に使用しないでください。整数定数の場合、列挙を使用し、浮動小数点定数には、上記で解説するようにconst修飾子を使用します。
- プリプロセッサがコードブロックを処理すべきか否かの判断に評価するシンボルには、大文字を使用します。例を示します。

```
#ifdef DEBUG
```

- コンパイラで定義されるマクロは、先頭と末尾にダブルアンダースコア文字を使用しています。例を示します。

```
__MACH__
```

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

- 通知名やディクショナリキーなどの目的に使用される文字列に、定数を定義します。文字列定数の使用により、正しい値の指定がコンパイラにより検証されていること（すなわちコンパイラがスペルチェックを実行すること）を確信できます。Cocoaフレームワークには、以下のような文字列定数の例が数多く使用されています。

```
APPKIT_EXTERN NSString *NSPrintCopies;
```

実際のNSString値は、実装ファイルで定数に割り当てられています。（`APPKIT_EXTERN`マクロは、Objective-C の`extern`に従って評価していることに注意。）

通知と例外

通知と例外の名前は、類似するルールに従います。ただし、いずれも独自の利用パターンが推奨されています。

通知

クラスにデリゲートが含まれる場合、その通知のほとんどは定義済みのデリゲートメソッドを通じてデリゲート側で受信される可能性があります。このような通知の名前には、対応するデリゲートメソッドを反映させなければなりません。たとえば、グローバル`NSApplication`オブジェクトのデリゲートは、アプリケーションが`NSApplicationDidBecomeActiveNotification`を投稿した場合は必ず`applicationDidBecomeActive:`メッセージを受信するように自動的に登録されています。

通知は、以下のような内容の名前を持つグローバル`NSString`オブジェクトで識別されます。

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

例を示します。

```
NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification
```

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

例外

例外（すなわちNSExceptionクラスと関連する関数で提供されるメカニズム）は、任意の目的で自由に使用できますが、Cocoaでは配列インデックスの境界越えなどのプログラミングエラーに例外を予約しています。Cocoaは通常の予測されるエラー状況の処理に例外を*使用しません*。このような場合、nil、NULL、NO、またはエラーコードなどの戻り値を使用します。詳細については、『*Error Handling Programming Guide*』を参照してください。

例外は、以下のような内容の名前を持つグローバルNSStringオブジェクトで識別されます。

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

名前の一意の部分は構成語句をグループ化し、各単語の先頭文字を大文字で表記します。次に、いくつかの例を示します。

```
NSColorListIOException
NSColorListNotEditableException
NSDraggingException
NSFontUnavailableException
NSIllegalSelectorException
```

使用できる略語と頭字語

一般に、プログラムインターフェイスを設計する場合、名称を省略できません（「一般原則」（6ページ）を参照）。ただし、以下の略語は広く定着している、あるいは過去に使用例があるため、そのまま使用することが許可されています。略語について、いくつかの注意すべき点を示します。

- 標準のCライブラリで長く使用されてきた形式と重複する略語（例:「`alloc`」や「`getc`」など）は、使用が許可されます。
- 引数名ではより自在に略語を使用できます（例:「`imageRep`」、「`col`」、（「カラム」の意味）、「`obj`」、「`otherWin`」など）。

略語	意味と説明
alloc	割り当て。
alt	代替。
app	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、「アプリケーション」はデリゲートメソッド、通知などではスペルアウトされます。
calc	計算。
dealloc	割り当て解除。
func	関数。
horiz	水平。
info	情報。
init	初期化（新しいオブジェクトを初期化するメソッドなど）。
int	整数（C `int`の文脈で。`NSInteger`値については、`integer`を使用）。
max	最大。
min	最小。
msg	メッセージ。
nib	インターフェイスビルダアーカイブ。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

略語	意味と説明
pboard	ペーストボード（ただし定数に限定）。
rect	矩形。
Rep	表示（`NSBitmapImageRep`などのクラス名で使用）。
temp	一時。
vert	垂直。

コンピュータ業界で共通に使用されている略語と頭字語を、その内容の語句の代わりに使用できます。広く認識されている一部の頭字語を示します。

ASCII

PDF

XML

HTML

URL

RTF

HTTP

TIFF

JPG

PNG

GIF

LZW

ROM

RGB

CMYK

MIDI

FTP

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

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者は、他の開発者以上にコードの記述方式に配慮することが要求されます。多くのクライアントアプリケーションはフレームワーク内にリンクが含まれています。このように広く外部に露出するために、フレームワーク内に何らかの不備が存在すると、システム全体の混乱を引き起こすことになります。以下の項目では、フレームワークの効率性と完全性を確実なものにするための、プログラミング手法について説明します。

> **注意** 下記の手法の一部は、フレームワーク以外にも適用されます。アプリケーション開発において、生産的に適用してください。

初期化

以下の提案と推奨事項は、フレームワークの初期化に関するものです。

クラスの初期化

`initialize` クラスメソッドは、同じクラスの他のメソッドが呼び出される前に、一部のコードを1回だけ、要求駆動的に実行する機会をもたらします。通常はクラスのバージョン番号の設定に使用されます（「バージョン管理と互換性」（30 ページ）を参照）。

クラスにより実装されていない場合でも、実行時に`initialize`を継承チェーンの各クラスに送信します。この後、クラスの`initialize`メソッドを複数回にわたり呼び出します（サブクラスにより実装されていない場合など）。通常は初期化コードの実行が必要になるのは1回限りです。初期化を確実に行う手段として、以下のようなチェックを行ってください。

```
if (self == [NSFoo class]) {
    // 初期化コード
}
```

`initialize`メソッドは明示的に呼び出すことはできません。初期化をトリガする必要が生じる場合、以下のような無害なメソッドを呼び出します。

```
[NSImage self];
```

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

指定イニシャライザ

指定イニシャライザとは、スーパークラスのinitを呼び出す、クラスのinitメソッドを指します。（他のイニシャライザは、クラスで定義されるinitメソッドを呼び出します。）すべてのパブリッククラスは、1つまたは複数の指定イニシャライザを使用します。指定イニシャライザの例として、NSViewのinitWithFrame:メソッドや、NSResponderのinitメソッドなどがあります。NSStringやクラスクラスタをフォント化するその他の抽象クラスと同様に、initメソッドの上書きが予定されていない場合、サブクラスは独自の実装が予定されます。

指定イニシャライザは、クラスのサブクラス化を必要とするユーザに重要な情報を含むため、明確に識別されなければなりません。サブクラスは指定イニシャライザを上書きするだけで、他のイニシャライザはすべて指定どおりに動作します。

特定のクラスのフレームワークを実装する場合、そのアーカイブメソッド、initWithCoder:とencodeWithCoder:も実装する場合が数多くあります。オブジェクトがアンアーカイブされているときには発生しない初期化コードパスで、操作を行わないように注意してください。このためには、指定イニシャライザの共有のルーチンと、クラスでアーカイブが実装されている場合は、（これ自体が指定イニシャライザである）initWithCoder:を呼び出す方法を用いるのが適切です。

初期化中のエラー検出

正しく設計された初期化メソッドを指定した場合、以下の手順が正常に終了し、エラーの正しい検出および伝達が行われます。

1. superの指定イニシャライザを呼び出して、自己再割り当てを実行します。
2. スーパークラス初期化で何らかのエラーが発生していることを示す、nilが戻り値に含まれていないか確認します。
3. 現在のクラスの初期化中にエラーが発生した場合、オブジェクトをリリースし、nilを返します。

リスト 1 （29 ページ）にこの方法を図示しています。

リスト 1　初期化中のエラー検出

```
- (id)init {
    self = [super init];  // Call a designated initializer here. (ここで指定イニシャライザを呼び出す)
    if (self != nil) {
        // オブジェクトを初期化
        if (someError) {
            [self release];
            self = nil;
```

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

```
        }
    }
    return self;
}
```

バージョン管理と互換性

作成中のフレームワークに新しいクラスまたはメソッドを追加する場合、通常は新しい機能グループごとに新しいバージョン番号を指定する必要はありません。通常、開発者は、`respondsToSelector:`などのObjective-C実行時チェックを実行し、所定のシステムで機能が利用できるかどうかを判断します。このような実行時テストは、新しい機能をチェックする推奨される最も動的な方法です。

ただし、フレームワークの新しいバージョンごとに正しくマーキングされ、前回のバージョンと互換性を持つかどうかを確認する場合、複数の方法を用いることができます。

フレームワークのバージョン

新しい機能またはバグ修正の存在が実行時テストで簡単に検出できない場合、変更を確認する何らかの手段を開発者に提供する必要があります。このために、フレームワークの正確なバージョン番号を保存し、この番号を開発者に公開する方法があります。

- 特定のバージョン番号で（リリースノートなどで）変更をドキュメント化します。
- 現在使用中のフレームワークのバージョン番号を設定し、グローバルなアクセスを可能にする何らかの手段を提供します。このバージョン番号は、フレームワークの情報のプロパティリスト（`Info.plist`）に保存し、ここからアクセスします。

キーアーカイブ

フレームワークのオブジェクトをnib fileに書き込む必要がある場合、自動的にアーカイブされなければなりません。またアーカイブメカニズムを使用してドキュメントデータを保存するすべてのドキュメントは、アーカイブする必要があります。

ここでアーカイブに関する以下の問題に注意してください。

- アーカイブでキーが紛失した場合、その値の照会で、照会された値のタイプに応じて`nil`、`NULL`、`NO`、0、または0.0が返されます。この戻り値をテストして、書き出すデータを減らします。また、キーがアーカイブに書き込まれているかどうかを確認することもできます。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

- エンコードとデコードはいずれの方法も、後方互換性を維持するための操作を実行します。たとえば、あるクラスの新しいバージョンのエンコード方式では、キーを使用して新しい値を記述しますが、古いバージョンのクラスがオブジェクトを理解できるように、古いフィールドを書き出す場合があります。また、デコード方式では欠落した値を何らかの合理的な方法で処理し、後のバージョンの互換性を維持する場合があります。
- フレームワークのクラスのアーカイブキーに推奨される命名規約では、フレームワークの他のAPI要素で使用される接頭辞を先頭に配置し、インスタンス変数の名称を続けます。この名前がスーパークラスまたはサブクラスの名称と重複しないか確認してください。
- ユーティリティ関数を使って基本データ型（すなわちオブジェクト以外の値）を書き出す場合、必ず一意のキーを使用してください。たとえば、「archiveRect」ルーチンが矩形をアーカイブする場合、キー引数をとるか、キー引数を使用します。あるいはルーチンが複数の値（浮動小数点値など）を書き出す場合、指定されたキーに一意のビットを付加する必要があります。
- ビットフィールドをそのままアーカイブする操作は、コンパイラとエンディアンの依存性により危険な場合があります。パフォーマンス上の理由により、ビットフィールドは、大量のビットを多数回にわたり書き出す必要のある場合にのみアーカイブしてください。詳細は「ビットフィールド」（33 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大半は、開発者に対して例外のキャッシュと処理を強制しません。これは、例外が通常の実行の一部として投げられるのではなく、通常は予測される実行時またはユーザエラーの伝達に使用されないためです。これらのエラーの例の一部を以下に示します。

- File not found（ファイルが見つかりません）
- No such user（ユーザが存在しません）
- Attempt to open a wrong type of document in an application（アプリケーションで間違ったタイプのドキュメントが開かれようとしています）
- Error in converting a string to a specified encoding（文字列の指定されたエンコーディングへの変換中にエラーが発生しました）

ただし、Cocoaでは以下のようなプログラミングエラーまたはロジックエラーを通知する場合に例外を投げます。

- 配列インデックスの境界越え
- 不変オブジェクトの変更の試み
- 不正な引数タイプ

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

開発者はテスト中のこのような種類のエラーを捕捉し、アプリケーションの発送前にエラーに対処することが予測されます。このため、アプリケーションでは実行時に例外を処理する必要性は生じません。例外が投げられ、アプリケーションのどのパートでも捕捉しない場合、通常は最上位のデフォルトハンドラが例外と実行を捕捉し、報告して、処理を続けます。開発者はこのようなデフォルトの例外キャッチャーを、問題の内容をより詳細に伝え、データを保存し、アプリケーションを中断するオプションを提供する機能に置き換えることができます。

エラー領域においても、Cocoaフレームワークは他のソフトウェアライブラリと差別化されます。Cocoaのメソッドは、一般にエラーコードを返しません。エラーに正当な理由あるいは予測可能な理由が存在する場合、ブール値またはオブジェクト（`nil`/non-`nil`）といった簡単な手段により戻り値をテストします。戻り値、`NO`または`nil`の理由はドキュメント化されます。実行時に処理されるプログラミングエラーの通知にエラーコードを使用できません。例外を投げたり、場合によっては例外を投げずにエラーを記録するだけの操作に留まります。

たとえば、`NSDictionary`の`objectForKey:`メソッドは検出されたオブジェクトを返すか、オブジェクトが検出されない場合は`nil`を返します。`NSArray`の`objectAtIndex:`メソッドは、`nil`を返すことができません（`nil`に対するメッセージはすべて`nil` が返されるという、最優先される一般言語規則を除く）。これは`NSArray`オブジェクトが`nil`値を格納することができず、定義上、境界越えアクセスはすべてプログラミングエラーになり、例外が発行されるという理由によるものです。多くの`init`メソッドは、指定されたパラメータでオブジェクトを初期化できない場合に`nil`を返します。

メソッドが複数の独自のエラーコードを正当に要求する、ごく少数のクラスにおいては、エラーコードかローカライズされたエラー文字列、あるいはエラーを記述するその他の情報を返す参照渡し引数でエラーコードを指定する必要があります。たとえば、エラーを`NSError`オブジェクトとして返すことが適切な場合があります。詳細については、Foundationの`NSError.h`ヘッダファイルを参照してください。この引数は、直接返されるシンブルな`BOOL`や`nil`と同様に使用できる場合があります。この方法は、すべての参照渡し引数をオプションとし、エラーの詳細を把握する必要がない場合に、送信者がエラーコード引数として`NULL`を渡す規則にも従っています。

フレームワークのデータ

フレームワークのデータの処理方法は、パフォーマンス、プラットフォーム間の互換、およびその他の目的に対して影響を持ちます。ここでは、フレームワークのデータに関連した手法について説明します。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

固定データ

パフォーマンス上の理由により、可能な限り多くのフレームデータを固定とマークするのが適切です。このような処理により、Mach-Oバイナリの__DATAセグメントのサイズを減らすことができます。const以外のグローバルな静的データは、__DATAセグメントの__DATAセクションで終了します。この種のデータは、フレームワークを使用するアプリケーションのインスタンスを実行するたびにメモリを消費します。500バイト（一例）の余剰消費は問題がないように思われますが、要求されるページ数の増加が生じる場合があります。アプリケーションあたり4キロバイトの余剰です。

固定したデータはすべてconstとマークする必要があります。ブロック内にchar *ポインタが使用されていない場合、データが__TEXTセグメントに入り（実際に固定データになる）、それ以外の場合は__DATAセグメントに留まりますが、（プリバインディングが実行されていない場合を除き、あるいはロード時のバイナリ渡しでプリバインディングに違反している場合を除き）データは書き込まれません。

静的変数を初期化し、__DATAの__bssセクションではなく、__dataセクションへのマージを確実に行います。初期化に使用できる値が見つからない場合は、0、NULL、0.0、または何らかの適切な値を使用します。

ビットフィールド

ビットフィールド、特に1ビットのビットフィールドに符号付きの値を使用した場合、コードが値をブール値と想定すれば、未定義の動作が起こることがあります。1ビットのビットフィールドは、必ず符号なしで使用してください。このようなビットフィールドに格納できる値は0と-1（コンパイラの実装に応じる）に限定されるため、このビットフィールドの1との比較はfalseです。たとえば、コード内で以下のような内容が見つかる場合:

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

タイプをunsigned intに変更する必要があります。

ビットフィールドには、アーカイブにも問題があります。一般に、ビットフィールドは元の形でディスクまたはアーカイブに書き込むことができません。別のアーキテクチャあるいは別のコンパイラで読み出す際に、形式が異なる場合があるためです。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

メモリ割り当て

フレームワークコードでは、避けられる場合は、メモリの同時割り当てを避けるのが賢明です。何らかの理由で一時バッファが必要になる場合、通常はバッファを割り当てるよりもスタックを使用する方が適切です。ただし、スタックは容量に制限があるため（通常は合計512キロバイト）、必要なバッファの機能と容量に基づいてスタックの使用を決定してください。通常は、バッファサイズが1000バイト（あるいはMAXPATHLEN）以下の場合、スタックの使用が推奨されます。

上記の改良型として、最初はスタックを使用し、サイズ要件がスタックのバッファ容量を超えるようになれば、mallocで割り当てたバッファに切り替えます。リスト 2 （34 ページ）では、この方法の解説を示しています。

リスト 2　スタックとmallocバッファを使用した割り当て

```
#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;
    }
    // ... バッファを使用、numElementsをインクリメント
  }
  // ...
  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isEqual:と、isEqualToString:などの、特定のオブジェクトタイプに関連付けられた比較メソッドの重要な違いを認識する必要があります。isEqual:メソッドの場合、引数として任意のオブジェクトを渡すことができ、オブジェクトが同じクラスでなければメソッドはNOを返します。isEqualToString:やisEqualToArray:などのメソッドは、通常は引数が指定されたタイプ（レシーバのタイプ）と想定します。したがって、このようなメソッドは型検査を実行せず、結果的に高速化しますが安全性は維持されません。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、外部ソースから取得された値の場合、安全上の理由によりisEqual:の使用が推奨されます。既知の型の場合はisEqualToString:で代用します。

isEqual:に関するもう一つの注意点は、hashメソッドとの関連です。NSDictionaryやNSSetなど、ハッシュベースのCocoaコレクションに配置されるオブジェクトには、[A isEqual:B] == YESの場合[A hash] == [B hash]という基本的な不変性があります。このため、現在のクラスでisEqual:を上書きする場合、hashも上書きし、この不変性を維持する必要があります。デフォルトで、isEqual:は各クラスのアドレスのポインタの等価性を調べ、hashは各オブジェクトのアドレスに基づいてハッシュ値を返すため、この不変性は維持されます。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

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
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年7月23日	いくつかのバグを修正しました。
2003年4月28日	『*コーディングのガイドライン*』の初版。

2012-02-16 | © 2003, 2012 Apple Inc. All Rights Reserved.

Apple Inc.
© 2003, 2012 Apple Inc.
All rights reserved.

本書の一部あるいは全部を Apple Inc. から書面による事前の許諾を得ることなく複写複製（コピー）することを禁じます。また、製品に付属のソフトウェアは同梱のソフトウェア使用許諾契約書に記載の条件のもとでお使いください。書類を個人で使用する場合に限り 1 台のコンピュータに保管すること、またその書類にアップルの著作権表示が含まれる限り、個人的な利用を目的に書類を複製することを認めます。

Apple ロゴは、米国その他の国で登録された Apple Inc. の商標です。

キーボードから入力可能な Apple ロゴについても、これを Apple Inc. からの書面による事前の許諾なしに商業的な目的で使用すると、連邦および州の商標法および不正競争防止法違反となる場合があります。

本書に記載されているテクノロジーに関しては、明示または黙示を問わず、使用を許諾しません。本書に記載されているテクノロジーに関するすべての知的財産権は、Apple Inc. が保有しています。 本書は、Apple ブランドのコンピュータ用のアプリケーション開発に使用を限定します。

本書には正確な情報を記載するように努めました。ただし、誤植や制作上の誤記がないことを保証するものではありません。

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

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/

Apple, the Apple logo, Cocoa, Mac, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

Apple Inc. は本書の内容を確認しておりますが、本書に関して、明示的であるか黙示的であるかを問わず、その品質、正確さ、市場性、または特定の目的に対する適合性に関して何らかの保証または表明を行うものではありません。その結果、本書は「現状有姿のまま」提供され、本書の品質または正確さに関連して発生するすべての損害は、購入者であるお客様が負うものとします。

いかなる場合も、Apple Inc. は、本書の内容に含まれる瑕疵または不正確さによって生じる直接的、間接的、特殊的、偶発的、または結果的損害に対する賠償請求には一切応じません。そのような損害の可能性があらかじめ指摘されている場合においても同様です。

上記の損害に対する保証および救済は、口頭や書面によるか、または明示的や黙示的であるかを問わず、唯一のものであり、その他一切の保証にかわるものです。 Apple Inc. の販売店、代理店、または従業員には、この保証に関する規定に何らかの変更、拡張、または追加を加える権限は与えられていません。

一部の国や地域では、黙示あるいは偶発的または結果的損害に対する賠償の免責または制限が認められていないため、上記の制限や免責がお客様に適用されない場合があります。 この保証はお客様に特定の法的権利を与え、地域によってはその他の権利がお客様に与えられる場合もあります。