Cocoa はやっぱり!
ジャガーの新機能の巻
2002.10.12 (4th Edition)

■ 今回のテーマ

今回のテーマは、**Mac OS X 10.2 (Jaguar) の新機能や変更点**についてです。全ての項目について説明はできませんが、これはと思ったものについてピックアップして解説を行っていきます。

・アニメ GIF 対応
・対応画像フォーマット
・ネットワークのプロキシー対応
・AppleScript を扱う
・プロセス情報
・円形のテキストフィールド
・スピニングスタイルのプログレスインジケータ
・テクスチャードウィンドウ
・ウィンドウの親子関係
・メニューバーを隠す
・24 ドットサイズのツールバーアイコン
・ドラッグ&ドロップ
・配列操作
・スレッドのプライオリティ
・バージョンチェック
・ファイル名とフォルダー名のローカライズ

◎ 推奨環境

この解説は、以下の環境を前提にしていますので、ご確認ください。

・Mac OS X 10.2
・Project Builder Version 2.0.1
・Interface Builder 2.3.1

◎ 更新履歴

・2002.09.07 ： 1st : 新規作成
・2002.09.09 ： 2nd : 配列の項を追加、ツールバーの詳細追加、AppleScript から値をもらうなど
・2002.09.14 ： 3rd : 図の追加、ProgressIndicator の詳細追加など
・2002.10.12 ： 4th : ドラッグ&ドロップに HFS Promises を追加

■ アニメ GIF 対応 : NSBitmapImageRep

NSBitmapImageRep は、アニメ GIF を扱えるようになりました。「 フレーム数／現在のフレーム番号、現在のフレームの長さ 」をプロパティとして扱えます。ただし、変更できるのは「 現在のフレーム番号 」だけです。まずは、フレーム数の取得を見てみましょう。以下のようになります。

フレーム数を取得

```
NSBitmapImageRep *bRep = 読み込み処理;
int iNumFrame = [ [ bRep valueForProperty : NSImageFrameCount ] intValue ];
```

NSBitmapImageRep のプロパティを取得するには「 **valueForProperty :** 」メソッドを使います。パラメータにはプロパティ名を文字列で指定しますが、フレーム数に対応するプロパティ名は「 **NSImageFrameCount** 」になります。値は NSNumber のインスタンスで取得しますので、NSNumber の intValue メソッドで int に変換します。

プロパティ名は以下の 3 つになります。フレーム数とカレントフレーム番号は整数値。カレントフレームの長さは秒の単位で浮動小数値になります。

アニメGIF関連のプロパティ名

```
NSImageFrameCount           : 全フレーム数
NSImageCurrentFrame         : カレントのフレーム番号 ( 0 ～ 全フレーム数-1 )
NSImageCurrentFrameDuration : カレントのフレームの長さ ( 秒 : float )
```

カレントフレーム番号だけが変更可能で、変更には「 **setProperty : withValue :** 」メソッドを使います。**カレントフレーム番号は、0 から始まります**ので、以下のように 2 をセットすると 3 枚目のフレームがカレントになります。カレントフレームを変えた後に再描画を行うことでアニメーションとして表示されることになります。

```
[ bRep setProperty : NSImageCurrentFrame
         withValue : [ NSNumber numberWithInt : 2 ] ]; // 3枚目のフレーム
```

NSBitmapImageRep : プロパティを取得

書式

```
- (id) valueForProperty : (NSString *) property
```

入力

property : 取得したいプロパティの名前

出力

返り値 : 指定したプロパティのインスタンス

NSBitmapImageRep : プロパティを変更

書式

```
- (void) setProperty : (NSString *) property
            withValue : (id        ) value
```

入力

```
property : 変更したいプロパティの名前
id       : 変更したいプロパティの値
```

では、NSImageView を拡張してアニメーションを表示するビューを作ってみましょう。startAnimation : と stopAnimation : の 2 つのメソッドを追加します。クラス名は AnimImageView とします。2 つのメソッドはアクションとして定義します。

AnimImageView.h

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

@interface AnimImageView : NSImageView {
    NSTimer *animTimer; // アニメーション用のタイマー
}
 - (IBAction) startAnimation : (id) sender; // アニメーションを開始
 - (IBAction) stopAnimation  : (id) sender; // アニメーションを停止
@end
```

AnimImageView.m

```
#import "AnimImageView.h"

@implementation AnimImageView

///// アニメーションを1フレーム進める /////

- (void) stepAnimation : (NSTimer *) aTimer {

  // セットされているNSImageからNSImageRepを取り出す
  NSImageRep *rep = [ [ self image ] bestRepresentationForDevice : nil ];

  // セットされているNSImageRepがNSBitmapImageRepかどうかを判定する
  if ( [ rep isKindOfClass : [ NSBitmapImageRep class ] ] ) {

    NSBitmapImageRep *bRep = (NSBitmapImageRep *) rep;

    // 各種プロパティを取得
    int iNum = [ [ bRep valueForProperty : NSImageFrameCount   ] intValue ];
    int iCur = [ [ bRep valueForProperty : NSImageCurrentFrame ] intValue ];
    float fDur;
```

AnimImageView.m : 続き

```
    if ( aTimer != nil ) { // すでに動いている場合はフレームを進める
      if ( iNum <= iCur + 1 ) iCur = 0; // 最後のフレームなら先頭に戻す
      else                    iCur++;   // 1フレーム進める
      [ bRep setProperty : NSImageCurrentFrame
               withValue : [ NSNumber numberWithInt : iCur ] ];
    }

    fDur = [ [ bRep valueForProperty : NSImageCurrentFrameDuration ]
                floatValue ]; // 現在のフレームの長さを取得

    animTimer = [ NSTimer scheduledTimerWithTimeInterval : fDuration
                                          target : self
                                        selector : @selector( stepAnimation : )
                                        userInfo : nil
                                         repeats : NO ]; // タイマーを起動

    [ self setNeedsDisplay : YES ]; // 再描画

  }
}

///// アニメーションスタート /////

- (IBAction) startAnimation : (id) sender {
   [ self stepAnimation : nil ]; // 1フレーム進める
}

///// アニメーションストップ /////

- (IBAction) stopAnimation : (id) sender {
    [ animTimer invalidate ]; // タイマーをとめる
    animTimer = nil;
}
```

startAnimation : メソッドはアクションとして定義していますので、ボタンなどからこの AnimImageView に接続して使うことができます。このメソッドは、stepAnimation : というアニメを 1 フレーム進めるためのメソッドを呼ぶだけです。

stepAnimation : メソッドは、「 フレームを 1 つ進める ／ 新しいフレームの長さを取得する ／ 新しいフレームの長さ分の時間が経過したら再度 stepAnimation : メソッドが呼ばれるようにタイマーを仕掛ける 」という処理を行います。

startAnimation : メソッドから stepAnimation : メソッドを呼んでいるところではパラメータが nil になっていますが、これは、NSTimer によって起動されたかどうかを見分けるためのものです。NSTimer に

よってあるメソッドを起動した場合は、そのタイマーのインスタンスがパラメータとして渡されるので nil 以外の値になります。なぜ見分ける必要があるかですが、アニメーションをスタートさせたときの処理は、フレームを進めずに、現在のフレームを長さ分表示する必要があるからです。

さて、stepAnimation : メソッドを詳しく見ていきましょう。パラメータは、NSTimer から起動させるために NSTimer を 1 つ受けられるようにしています。

```
- (void) stepAnimation : (NSTimer *) aTimer {

  // セットされているNSImageからNSImageRepを取り出す
  NSImageRep *rep = [ [ self image ] bestRepresentationForDevice : nil ];

  // セットされているNSImageRepがNSBitmapImageRepかどうかを判定する
  if ( [ rep isKindOfClass : [ NSBitmapImageRep class ] ] ) {
```

まず最初に行うのは、ビューにセットされている画像が NSBitmapImageRep かどうかを判定することです。例えば、PDF がセットされているような場合ですと、先ほどのプロパティを取得するようなメソッドが存在しません。GIF を読み込んだ場合は、画像 (NSImage) に NSBitmapImageRep が格納されますので、それを確認します。

image メソッドで自分にセットされている画像を取得して、さらに bestRepresentationForDevice : で NSImageRep を取り出します。「 **isKindOfClass :** 」を使うことで**インスタンスのクラスを比較**できます。クラスの比較というのはあまりなじみがないかもしれませんが、NSObject には「 **class** 」メソッドというクラス自身が取り出せるクラスメソッドがありますので、これと組み合わせることで比較ができます。

```
    NSBitmapImageRep *bRep = (NSBitmapImageRep *) rep;

    // 各種プロパティを取得
    int iNum = [ [ bRep valueForProperty : NSImageFrameCount   ] intValue ];
    int iCur = [ [ bRep valueForProperty : NSImageCurrentFrame ] intValue ];
    float fDur;
```

ここでは、NSBitmapImageRep だと分かったので、NSImageRep をキャストして NSBitmapImageRep に代入しています。通常、スーパークラスからサブクラスへのキャストはご法度ですが、クラスを調べているのでこのケースでは問題ありません。その後、プロパティを取得しています。

```
    if ( aTimer != nil ) { // すでに動いている場合はフレームを進める
      if ( iNum <= iCur + 1 ) iCur = 0; // 最後のフレームなら先頭に戻す
      else                    iCur++;   // 1フレーム進める
      [ bRep setProperty : NSImageCurrentFrame
               withValue : [ NSNumber numberWithInt : iCur ] ];
    }
```

そして、フレームを 1 つ進めています。先ほど説明したように、startAnimation : から起動された場合はフレームは進めません。

```
    fDur = [ [ bRep valueForProperty : NSImageCurrentFrameDuration ]
                floatValue ]; // 現在のフレームの長さを取得

    animTimer = [ NSTimer scheduledTimerWithTimeInterval : fDuration
                                          target : self
                                        selector : @selector( stepAnimation : )
                                        userInfo : nil
                                         repeats : NO ]; // タイマーを起動

    [ self setNeedsDisplay : YES ]; // 再描画

  }
}
```

そして、現在のフレームの長さを取得して、その時間が経過したら働くタイマーを生成しておきます。起動するメソッドは、stepAnimation : 、つまり自分自身です。そして、画面を更新するために再描画を要求して終わりです。

さて、これで Web ブラウザのようにアニメ GIF が表示できるように思われるかもしれませんが、実はそこまではうまくいきません。各フレームの画像を取り出して表示できるようにはなるのですが、アニメ GIF によって画面に表示されるイメージは、透過色を含む場合などの場合に過去の複数のフレームと重ね合わせたものになることがあります。

現在のフレームだけで表示するか、過去のフレームと重ねるかというのは、GIF のデータの中に埋まっていますが、その情報を NSBitmapImageRep のメソッドで取り出すことはできませんので、あくまでも各フレームを独立して表示するしかできません。OS にバンドルされているプレビューアプリケーションでアニメ GIF を開くと同様の表示がなされますので、現状のフレームワークではここまでしかできないと思われます。

GIF ファイルは NSMovie にムービーとして読み込むことができますので、その方がよいかもしれません。

■ 対応画像フォーマット：NSImage, NSImageRep, NSBitmapImageRep

扱える画像のフォーマットが増えたようです。ただし、検証は行っていません。

・1, 2, 4bit パレット TIFF (Read)。
・16bit/sample TIFF (Read / Write)。
・CMYK JPEG (Read / Write)。
・グレイスケール JPEG (Read / Write)。

なお、リリースノートでは触れられていませんが、QuickTime 6.0 がインストールされていれば、**JPEG 2000 を読み込むことは可能**ですし、 `[ NSImage imageFileTypes ]` の中にも `'JP2'` が含まれています。

ただし、Editable にしても、`NSImageView` に対してのドラッグ&ドロップで JPEG 2000 を読み込ませることはできませんでしたし、JPEG 2000 への書き出す手段も提供されていないようです。当面は、QuickTime.framework を直接呼び出すしかないのでしょう。

■ 画像のインクリメンタルロード

今までは、画像を扱うクラスでは画像データを扱うには完全に読み込んだ状態にしておく必要がありました。例えば、Web ブラウザのように途中までしか読み込んでいない画像を表示するということはできなかったわけですが、JPEG 限定で可能になりました。GIF と PNG については将来サポートされるようです。

■ ネットワークのプロキシー対応：NSURLHandle

やっとという感じですが、NSURLHandle はシステム環境設定でプロキシーを設定している場合に、**プロキシー経由で通信を行う**ようになりました。今まではプロキシーに未対応だったために NSURLHandle の記事をあえて書いていませんでした。ですので、Mac OS X 10.2 の新機能ではありませんが、ここで少し解説をしておきます。

NSURL は、いわば URL の文字列そのものを扱うクラスです。ダウンロードなどのネットワーク処理機能は持っていません。ファイルパスを扱うには NSString を使いますが、それと同じ思っていただいてもよいでしょう。ダウンロードなどのネットワーク処理を行うのは **NSURLHandle** です。ファイル操作を行うのに NSFileHandle を使うのと似ています。

指定の URL からコンテンツをダウンロードするのは簡単です。NSURL を URL の文字列から「 **URLWithString :** 」メソッドで生成し、NSData の「 **dataWithContentsOfURL :** 」メソッドで一気にダウンロードすることができます。以下のサンプルは、画像をダウンロードして NSImageView にセットして画面に表示するまでの処理です。

フォアグランドで画像をダウンロード

```
- (void) loadForeground : (NSString *) asUrl {

    NSURL   *url = [ NSURL URLWithString : asUrl ];        // NSURL作成
    NSData  *dat = [ NSData dataWithContentsOfURL : url ]; // NSDataでダウンロード
    NSImage *img = [ [ [ NSImage alloc ]
                         initWithData : dat ]
                         autorelease ];                    // 画像生成
    [ viewImage setImage : img ]; // NSImageViewにセット

}
```

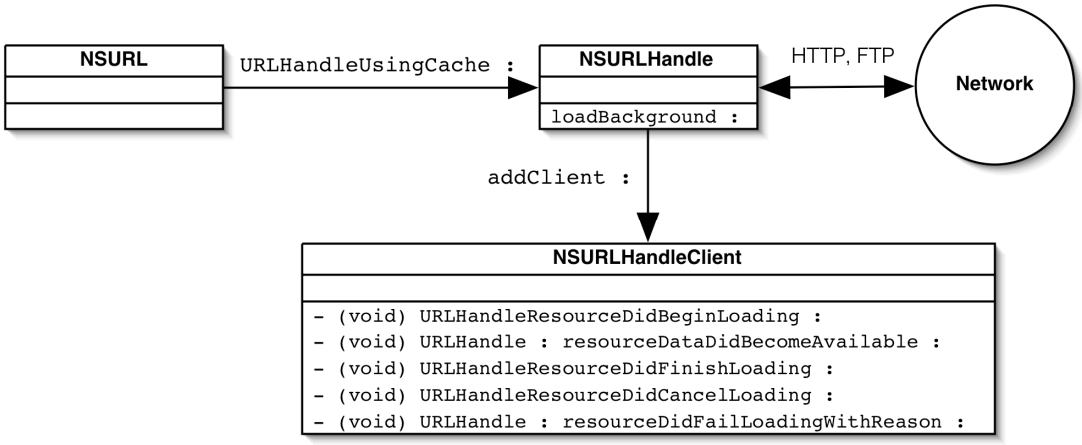
このコードですと、ダウンロードが完了するまで制御が戻ってきませんので、フレームワーク側にバックグランドでダウンロードをさせておいて、ダウンロードが終わったら知らせてもらうようにしましょう。

バックグランドで画像をダウンロード

```
- (void) loadBackground : (NSString *) asUrl {

    NSURL        *url;  // URL
    NSURLHandle  *urlh; // NSURLを使ったネットワーク操作

    url  = [ NSURL URLWithString : asUrl ];
    urlh = [ url   URLHandleUsingCache : NO ]; // NSURLからハンドラを取得
    [ urlh addClient : self ];                 // 通知先を指定
    [ urlh loadInBackground ];                 // ダウンロード開始

}
```

NSURL には「 **URLHandleUsingCache :** 」メソッドがあり、NSURL から NSURLHandle を取得できます。名前から分かるようにキャッシュ機構も持っています。「 **addClient :** 」メソッドでは、ダウンロード完了などの通知先（ **クライアント**と呼びます ）を指定できます。ダウンロードが完了したり、エラーになった場合などには、このクライアントの特定のメソッドが呼ばれます。そのメソッドは、**NSURLHandleClient** にプロトコルとして定義されていますので、クライアントに実装しておく必要があります。メソッドは、以下の 5 つがあります。

```
@protocol NSURLHandleClient

  // ダウンロード開始
  - (void) URLHandleResourceDidBeginLoading : (NSURLHandle *) sender;

  // データ取得
  - (void) URLHandle : (NSURLHandle *) sender
            resourceDataDidBecomeAvailable : (NSData *) newBytes;

  // ダウンロード完了
  - (void) URLHandleResourceDidFinishLoading : (NSURLHandle *)sender;

  // ダウンロードキャンセル
  - (void) URLHandleResourceDidCancelLoading : (NSURLHandle *) sender;

  // ダウンロード失敗
  - (void) URLHandle : (NSURLHandle *) sender
            resourceDidFailLoadingWithReason:(NSString *)reason;

@end
```



ダウンロードが開始されたら「 **URLHandleResourceDidBeginLoading :** 」が呼ばれます。そしてデータを受信する度に「 **URLHandle : resourceDataDidBecomeAvailable :** 」が繰り返し呼ばれます。そして、ダウンロードが完了したら「 **URLHandleResourceDidFinishLoading :** 」が呼ばれます。

一般のソフトウェアではダウンロードをユーザがキャンセルできるようにボタン等を用意します。それがクリ

ックされた場合、NSURLHandle インスタンスの「 **cancelLoadInBackground** 」メソッドを呼ぶとダウンロードはキャンセルされます。さらに、クライアントの「 **URLHandleResourceDidCancelLoading :** 」が呼ばれます。エラーが発生した場合は「 **URLHandle : resourceDidFailLoadingWithReason :** 」が呼ばれます。

バックグランドダウンロード時のクライアント

```
///// ダウンロード開始 /////
- (void) URLHandleResourceDidBeginLoading : (NSURLHandle *) sender {
    NSLog( @"URLHandleResourceDidBeginLoading" );
}

///// データ受信 /////
- (void) URLHandle : (NSURLHandle *) sender
          resourceDataDidBecomeAvailable : (NSData *) newBytes {
    NSLog( @"resourceDataDidBecomeAvailable %d", [ newBytes length ] );
}

///// ダウンロード完了 /////
- (void) URLHandleResourceDidFinishLoading : (NSURLHandle *) sender {

    NSData      *dat;
    NSImage     *img;

    // 画像生成
    dat = [ sender resourceData ];  // データ取得
    img = [ [ [ NSImage alloc ] initWithData : dat ] autorelease ];
    [ viewImage setImage : img ];

    [ sender removeClient : self ]; // クライアント登録抹消

    NSLog( @"URLHandleResourceDidFinishLoading" );
}

///// ダウンロードキャンセル /////
- (void) URLHandleResourceDidCancelLoading : (NSURLHandle *) sender {
    [ sender removeClient : self ]; // クライアント登録抹消
    NSLog( @"URLHandleResourceDidCancelLoading" );
}

///// ダウンロードエラー /////
- (void) URLHandle : (NSURLHandle *) sender
          resourceDidFailLoadingWithReason : (NSString *) reason {
    [ sender removeClient : self ]; // クライアント登録抹消
    NSLog( @"resourceDidFailLoadingWithReason : %@", reason );
}
```

■ FTP と IPv6 のサポート

NSURLHandle は HTTP と HTTPS に加えて FTP もサポートするようになりました。また IPv6 も利用できます。NSURLHandle のプロパティにも FTP 用のものがいくつか追加されています。

```
NSFTPPropertyUserLoginKey;      // ログインID : デフォルト値 "anonymous"
NSFTPPropertyUserPasswordKey;   // パスワード : デフォルト値 "NSURLHandle@apple.com"
NSFTPPropertyActiveTransferModeKey; // アクティブモードかどうか : デフォルト値 NO
NSFTPPropertyFileOffsetKey;     // デフォルト値 0
```

■ AppleScript を扱う : NSAppleScript

■ 文字列でスクリプトを与える

Foundation の中に **NSAppleScript** というクラスが新たに追加されました。名前のとおり **AppleScript を扱うためのクラス**です。まずは、単純なサンプルを見ていただきましょう。「 open location "http://www.appl.com/" 」という AppleScript を実行するサンプルです。このスクリプトは「 デフォルトブラウザで "http://www.apple.com/" を表示する 」というものです。通常ならば、Internet Explorer が起動して Apple のサイトが表示されます。

文字列で与えたAppleScriptを実行する

```
- (IBAction) RunAppleScript : (id) sender {

    NSDictionary  *asErrDic = nil; // エラー情報

    NSAppleScript *as = [ [ NSAppleScript alloc ]
        initWithSource : @"open location ¥"http://www.apple.com/¥"" ]; // 初期化

    [ as executeAndReturnError : &asErrDic ]; // 実行

}
```

まず、NSAppleScript のインスタンスを「 **alloc** 」メソッドで生成して「 **initWithSource :** 」メソッドで初期化しています。initWithSource : では、AppleScript のソースを文字列として与えることができます。初期化ができたら「 **executeAndReturnError :** 」メソッドでスクリプトを実行します。スクリプトのコンパイルも自動的に行われます。エラー情報を辞書形式で受け取るために、パラメータには NSDictionary のポインタのアドレスを指定します。

NSAppleScript : 文字列で与えられたスクリプトで初期化する

書式
```
- (id) initWithSource : (NSString *) source
```
入力
source : 実行したいスクリプトの文字列
出力
返り値 : 初期化されたインスタンス

■ ファイルでスクリプトを与える

AppleScript を文字列ではなく、別途用意したスクリプトのファイルで与える方法もあります。「 **initWithContentsOfURL : error :** 」メソッドを使います。これは、URL 指定ですので、ファイルの場所を NSURL で渡します。

ファイルで与えたAppleScriptを実行する

```
- (IBAction) RunAppleScript : (id) sender {

    NSDictionary  *asErrDic = nil; // エラー情報

    // Resourcesの中のFileのパスをURLにする
    NSString *asPath = [ [ [ NSBundle mainBundle ] resourcePath ]
                        stringByAppendingPathComponent : @"TEST.applescript" ];
    NSURL    *url    = [ NSURL fileURLWithPath : asPath ];

    NSAppleScript *as = [ [ NSAppleScript alloc ]
                              initWithContentsOfURL : url
                                              error : &asErrDic ]; // 初期化

    [ as executeAndReturnError : &asErrDic ]; // 実行
}
```

このサンプルでは、アプリケーションの中の Resources フォルダーの中に「 TEST.appleScript 」というファイルを入れておいてそれを読み込んでいます。こうしておけば、アプリケーションの中にスクリプトのファイルが隠されることになりますので、アプリケーションを利用しているユーザーには、普通のアプリケーションにしか見えません。

なお、スクリプトファイルは、コンパイルしていないものならば Project Builder 上で直接編集が可能です。

NSAppleScript : ファイルで与えられたスクリプトで初期化する

書式

```
- (id) initWithContentsOfURL : (NSURL         *) url
                      error : (NSDictionary **) err
```

入力

```
url    : 実行したいスクリプトファイルの場所
err    : エラー情報を受け取る辞書のポインタのアドレス
```

出力

```
err    : エラー情報
返り値 : 初期化されたインスタンス
```

NSAppleScript : ファイルで与えられたスクリプトで初期化する

書式

```
- (NSAppleEventDescriptor *) executeAndReturnError : (NSDictionary **) err
```

入力

```
err    : エラー情報を受け取る辞書のポインタのアドレス
```

出力

```
err    : エラー情報
返り値 : AppleScriptから戻された情報
```

■ AppleScript から結果を受け取る

AppleScript から実行結果を受け取るには、`executeAndReturnError :` メソッドの返り値を使います。返り値の **`NSAppleEventDescriptor`** というのは、AppleEvent を使ってアプリケーション間などでやりとりする情報が入っているものです。

例えば、iTunes で現在再生中の曲名を問い合わせる AppleScript で書くと以下のようになります。

iTunesで現在再生中の曲を取得

```
tell application "iTunes"
  return ( name of current track as text )
end tell
```

`return` 文で文字列を返していますが、この文字列を `NSAppleEventDescriptor` から取り出すには、**`stringValue`** メソッドを呼ぶだけです。AppleScript から返ってくる情報は文字列以外にも様々な形式のものがありますので、その形式に応じて取り出し方は変わります。

AppleScriptの実行結果から文字列を取り出す

```
NSAppleEventDescriptor *aed;
  : 省略
aed = [ as executeAndReturnError : &asDic ];      // 実行
NSLog( @"Song Name = %@", [ aed stringValue ] );  // 文字列取り出し
```

■ プロセス情報 : NSWorkspace

■ 起動しているアプリケーションのリスト

今までは、起動しているアプリケーションの情報を得るためには Carbon API を使う必要がありましたが、NSWorkspace で取得できるようになりました。

NSWorkspace の「 **launchedApplications** 」メソッドで**起動しているアプリケーションの情報が配列で取得**できます。配列の中身は辞書 (NSDictionary) になっているので、objectForKey : メソッドを使って必要な情報を取り出します。情報を取り出すためのキーには以下のものがあります。

アプリケーションの情報を取り出すキー

```
NSApplicationPath                    : アプリケーションのパス ( NSString )
NSApplicationName                    : アプリケーションの名前 ( NSString )
NSApplicationProcessSerialNumberHigh : プロセスシリアル番号の上位 ( NSNumber long )
NSApplicationProcessSerialNumberLow  : プロセスシリアル番号の下位 ( NSNumber long )
ProcessIdentifier                    : プロセスID ( NSNumber )
```

起動しているアプリケーションの名前とパスを表示するサンプルを作ると以下のようになります。

起動している全アプリケーションを表示する

```
NSEnumerator  *apps = [ [ [ NSWorkspace sharedWorkspace ]
                              launchedApplications ] objectEnumerator ];
NSDictionary *dicApp;

while ( dicApp = [ apps nextObject ] ) {
  NSLog( @"Name = %@", [ dicApp objectForKey : @"NSApplicationName" ] );
  NSLog( @"Path = %@", [ dicApp objectForKey : @"NSApplicationPath" ] );
}
```

Finder ならば、名前は「 Finder 」、パスは「 /System/Library/CoreServices/Finder.app 」という値が返ってきます。

NSWorkspace : 起動している全アプリケーションの情報取得

書式

```
- (NSArray *) launchedApplications
```

出力

返り値 : 起動している全てのアプリケーションの情報 (NSDictionary) の配列

■ アクティブなアプリケーション

また、アクティブなアプリケーションについての情報も得ることできます。「 **activeApplication** 」メソッドでアクティブアプリケーションに関する情報の辞書を取得して、objectForKey : で同様に詳細を取り

出します。

NSWorkspace ： アクティブなアプリケーションの情報取得

書式

```
- (NSDictionary *) activeApplication
```

出力

返り値 ： アクティブなアプリケーションの情報 (NSDictionary)

アクティブなアプリケーションの情報を表示する

```
NSDictionary *ap = [ [ NSWorkspace sharedWorkspace ] activeApplication ];

NSLog( @"Path = %@", [ ap objectForKey : @"NSApplicationPath" ] );
NSLog( @"Name = %@", [ ap objectForKey : @"NSApplicationName" ] );
```

一応実験した範囲ですが、起動直後に呼ばれる applicationDidFinishLaunching ： メソッドの中でこのコードを実行すると自分のパスを得ることができました。これが保証されるかどうかは分かりません。自分のパスは NSBundle の mainBundle で調べるのが確実でしょう。

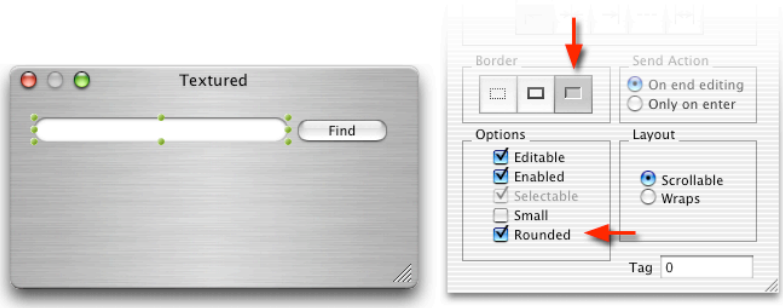
■ NSWorkspace の通知

NSWorkspace には、他のアプリケーションの起動／終了などの通知を受ける仕組みがありますが、その通知の中でもアプリケーションの情報がとれるようになりました。以下のコードで起動されたアプリのパスを表示できます。

```
// 起動時に通知先を設定
- (void) applicationDidFinishLaunching : (NSNotification *) aNote {

  NSNotificationCenter *nc = [ [ NSWorkspace sharedWorkspace ]
                                 notificationCenter ];
  [ nc addObserver : self
          selector : @selector( noteLaunch : )
              name : NSWorkspaceDidLaunchApplicationNotification
            object : nil ];
}

// 通知を受けるメソッド
- (void) noteLaunch : (NSNotification *) aNote {
  NSString *sPath = [ [ aNote userInfo ] objectForKey : @"NSApplicationPath" ];
  NSLog( @"Path = %@", sPath );
}
```

■ 円形のテキストフィールド：NSTextField, NSTextFieldCell

NSTextField と NSTextFieldCell のベゼル形状に円形のものが追加されました。Finder や Mail のツールバーで使われている検索用のフィールドは角が丸くなっていますが、あのようなものが作れます。



Interface Builder 上で Border をベゼル（一番右）にして、Options の中の **Rounded** にチェックをつけると丸くなるのが分かるでしょう。ポイントは**ベゼルが選択されているときにのみ有効**というところです。プログラムからコントロールする場合も同じで、ボーダーをベゼルにしてベゼルスタイルを指定します。

NSTextFieldを円形に変更

```
[ textFld setBezeled     : YES                        ]; // ボーダーをベゼルに変更
[ textFld setBezelStyle : NSTextFieldRoundedBezel ]; // ベゼルを円形に変更
```

ベゼルの形状には、以下の 2 つがあります。

ベゼルスタイル

```
typedef enum {
    NSTextFieldSquareBezel  = 0, // 四角
    NSTextFieldRoundedBezel = 1  // 円形
} NSTextFieldBezelStyle;
```

NSTextField ： ベゼルスタイルを変更

書式

```
- (void) setBezelStyle : (NSTextFieldBezelStyle) style
```

入力

style ： 変更したいベゼルスタイルの種類

NSTextField ： ベゼルスタイルを取得

書式

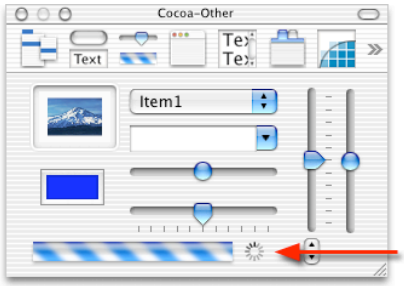
```
- (NSTextFieldBezelStyle) bezelStyle
```

出力

返り値 ： ベゼルスタイルの種類

■ スピニングスタイルのプログレスインジケータ : NSProgressIndicator

経過表示を行うためのプログレスインジケータ（ NSProgressIndicator ）に**スピニングスタイルが追加**されました。その場でクルクル回るだけなので、処理の経過状態というよりも「 現在処理を行っています 」ということを表示するためのものです。Interface Builder の Cocoa-Other に追加されていますので、そのままウィンドウにドロップして試すことができます。



スタイルを変更するのは、「 **setStyle :** 」メソッド、スタイルを取得するには「 **style** 」メソッドを使います。

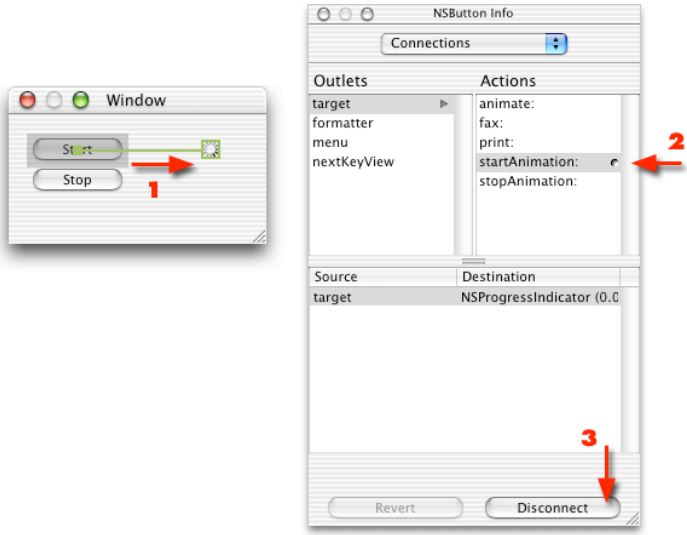
NSProgressIndicator : スタイルを変更
書式 `- (void) setStyle : (NSProgressIndicatorStyle) style` **入力** `style` : 変更したいスタイルの種類

NSProgressIndicator : スタイルを取得
書式 `- (NSProgressIndicatorStyle) style` **出力** 返り値 : スタイルの種類

スタイルには従来のバー状のものと、スピニングの 2 種類があります。これは、Interface Builder 上でも選択できます。

NSProgressIndicator : スタイルの種類
`NSProgressIndicatorBarStyle` : バー状のもの `NSProgressIndicatorSpinningStyle` : スピニングスタイル

アクションに startAnimation : と stopAnimation : がありますので、そこにボタンなどを繋げば、コーディングをしなくとも動きを試してみることができます。



startAnimation ： で動きを試す

プログレスバーは、経過表示を行っていない場合（ 処理が行われていない場合 ）には画面に表示されていない方がベターだということで、そのためのメソッドの追加も行われました。デフォルト値は、スタイルによって違います。「 バーときは表示する 」「 スピンの場合は表示しない 」になっています。

NSProgressIndicator ： 止まっているときの表示属性の変更

書式

```
- (void) setDisplayedWhenStopped : (BOOL) flag
```

入力

flag: 止まっているときも表示する - YES、止まっているときは隠す - NO

NSProgressIndicator ： 止まっているときの表示属性の取得

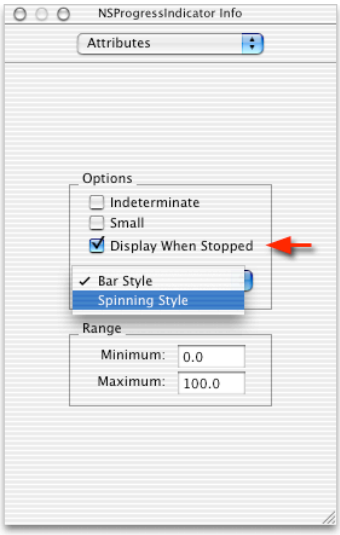
書式

```
- (BOOL) isDisplayedWhenStopped
```

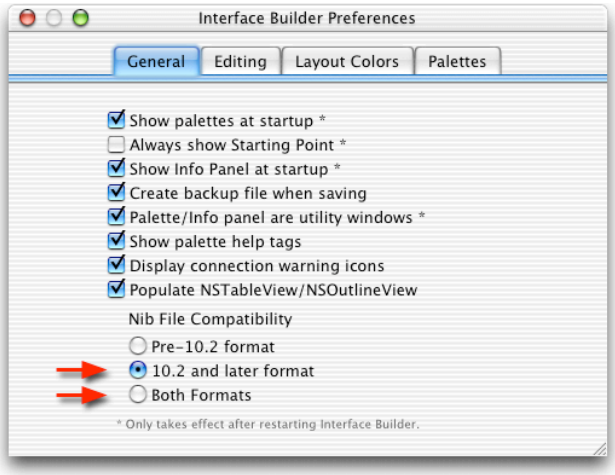
出力

返り値 ： 止まっているときも表示する - YES、止まっているときは隠す - NO

この属性も Interface Builder 上で設定できます。



なお、このスピニングスタイルのインジケーターを使用する場合は、Interface Builder の出力する nib ファイルのフォーマットを新しいものにしておく必要があります。環境設定で「 10.2 and later format 」か「 Both Formats 」を選択します。



Interface Builder の環境設定画面

■ テクスチャードウィンドウ (Textured Window) : NSWindow, NSView

ウィンドウは iTunes のようなメタリックな外観を持つものを作れるようになりました。Interface Builder 上で「 Textured Window 」のチェックボックスをクリックするだけです。また、ウィンドウにドローワーをつけた場合は、ドローワーの外観は親のウィンドウと連動して変化します。

このウィンドウは、タイトルバー以外のウィンドウの背景部分ならば、どこをつかんでもドラッグできますが、一般のウィンドウもそのようにすることができます。

NSWindow ： ウィンドウの背景でのドラッグ可能属性を変更

書式
```
- (void) setMovableByWindowBackground : (BOOL) flag
```
入力
flag ： 背景をドラッグ可能にする - YES、背景でのドラッグを不能に - NO

NSWindow ： ウィンドウの背景でのドラッグ可能属性を取得

書式
```
- (BOOL) isMovableByWindowBackground
```
出力
返り値 ： 背景がドラッグ可能 - YES、背景がドラッグ不能 - NO

この変更に伴って、NSView にもメソッドが追加されました。「 **ビューの中で発生したマウスダウンを処理しないので、ウィンドウを動かすのに使ってください** 」と宣言できるようになりました。必要に応じて mouseDownCanMoveWindow メソッドをオーバーライドしてください。

NSView ： マウスダウンをウィンドウドラッグに使用するかどうかの宣言

書式
```
- (BOOL) mouseDownCanMoveWindow
```
出力
返り値 ： ウィンドウドラッグにする - YES、しない - NO

■ ウィンドウの親子関係 : NSWindow

ウィンドウに親子関係を持たせることが可能になりました。親のウィンドウの位置を変更した時に、子供のウィンドウも同時に同じ距離だけ動きます。親のウィンドウをドラッグすると子供も同時に動きますが、子供を動かしたときは、その子供だけが動きます。また、子供のウィンドウは複数個持つことができます。

子供を追加するには、「 **addChildWindow : ordered :** 」メソッドを使います。2 つ目のパラメータは、ウィンドウの画面上での前後関係で、親のウィンドウに対して子供を手前に表示するか後ろに表示するかを指定できます。子供でなくすには「 **removeChildWindow :** 」メソッドを使います。

NSWindow : 子供ウィンドウの追加

書式

```
- (void) addChildWindow : (NSWindow           *) childWin
                 ordered : (NSWindowOrderingMode) place
```

入力

```
childWin : 子供にしたいウィンドウ
place    : 位置関係 ( NSWindowAbove = 親より手前, NSWindowBelow = 親より後ろ )
```

NSWindow : 子供ウィンドウを解除

書式

```
- (void) removeChildWindow : (NSWindow *) childWin
```

入力

```
childWin : 解除したいウィンドウ
```

現在の子供のウィンドウのリストは「 **childWindows** 」メソッドで取得できますし、子供のウィンドウ側から親を知るための「 **parentWindow** 」メソッドもあります。

NSWindow : 子供ウィンドウのリスト取得

書式

```
- (NSArray *) childWindows
```

出力

```
返り値 : 子供ウィンドウ ( NSWindow ) の配列
```

NSWindow : 親ウィンドウの取得

書式

```
- (NSWindow *) parentWindow
```

出力

```
返り値 : 親ウィンドウ
```

■ メニューバーを隠す : NSMenu

動画再生やスライドショー／ゲームなどでメニューを消したい場合がありますが、これが簡単にできるようになりました。NSMenu の「 **setMenuBarVisible :** 」メソッドを呼ぶだけです。

メニューバーを消す

```
[ NSMenu setMenuBarVisible : NO ];
```

NSMenu : メニューを隠したり表示したりする

書式

```
+ (void) setMenuBarVisible : (BOOL) visible
```

入力

visible : YES - 表示する、NO - 隠す

NSMenu : メニュー表示状態を取得

書式

```
+ (BOOL) menuBarVisible
```

入力

返り値 : YES - 表示されている、NO - 隠されている

■ 24 ドットサイズのツールバーアイコン : NSToolbar, NSToolbarItem

ツールバーのアイコンは、今までの 32 ドットに加えて **24 ドットサイズ**が利用可能になりました。特に何もしなくても自動的に使えるようになります。カスタマイズパレットには 24 ドットサイズを利用するためのチェックボックスが追加されたり、コンテキストメニューからも変更が可能です。

1 つのファイルに複数の解像度の画像を格納できる画像ファイル形式を使って、24 と 32 ドットの両方のアイコンを入れておくと奇麗に表示されます。

ツールバーのアイコンサイズ定義

```
typedef enum {
  NSToolbarSizeModeDefault,  // デフォルトサイズ
  NSToolbarSizeModeRegular,  // 32ドットサイズ
  NSToolbarSizeModeSmall     // 24ドットサイズ
} NSToolbarSizeMode;
```

NSToolbar ： ツールバーのアイコンサイズを変更

書式
```
- (void) setSizeMode : (NSToolbarSizeMode) sizeMode;
```
入力
sizeMode ： ツールバーのアイコンサイズ

NSToolbar ： ツールバーのアイコンサイズを取得

書式
```
- (NSToolbarSizeMode) sizeMode;
```
出力
返り値 ： ツールバーのアイコンサイズ

■ ドラッグ&ドロップ

ドラッグ&ドロップでは、**HFS Promises** のサポートが追加されました。Internet Explorer で画像を Finder へドロップしたら画像ファイルが作られます。また、Fetch で FTP 上のファイルリストから Finder にドロップすることでドロップ先のフォルダーにダウンロードが出来ます。こういったことをやるには、この HFS Promises を使う必要があります（逆に言うと今までは Cocoa アプリからはできなかったということです）。

ドラッグするデータの種類には、テキストや画像などがありましたが、そこに「**NSFilesPromisePboardType**」という種類が追加されました。それをドラッグするデータに格納しておくことになるのですが、専用のメソッドがあるので通常はドラッグを開始する側のビューには以下のようなコードを書いておけばよいです。

```
- (void) mouseDragged : (NSEvent *) theEvent {

  [ self dragPromisedFilesOfTypes : [ NSArray arrayWithObject : @"txt" ]
                         fromRect : [ self bounds ]
                           source : self
                        slideBack : YES
                            event : theEvent ]; // HFS Promisesをドラッグ開始

}
```

最初のパラメータが、ドラッグするファイルの種類の配列です。拡張子もしくはファイルタイプを文字列として与えます。ここでは 1 つにしていますが、もちろん複数指定が出来ます。2 番目（fromRect）は、ドラッグするファイルがビューの中のどこにあるかの矩形情報です。ここでは、ビュー全体にしています。

3 番目（source）は、このドラッグ元の処理を担当するインスタンスを渡します。NSDraggingSource プロトコルを実装しているインスタンスになりますが、このコードでは、NSView を継承したビューの中に NSDraggingSource プロトコルを実装している場合の書き方を想定していまして、self を渡しています。

このインスタンスには、ドラッグ中やドロップ時に色んなメソッドが呼ばれます。

4 番目（ slideBack ）は、ドロップ先にドロップを拒否されたときのアニメーションを表示するかどうかのフラグ。5 番目（ event ）は、イベント情報ですので、受け取ったイベント情報をそのまま横流しにしています。

NSDraggingSource ： ドロップが受け入れられた時に呼ばれるメソッド

書式

```
- (BOOL) dragPromisedFilesOfTypes : (NSArray *) typeArray
                         fromRect : (NSRect   ) aRect
                           source : (id       ) sourceObj
                        slideBack : (BOOL     ) slideBack
                            event : (NSEvent *) theEvent
```

入力

typeArray ： ドロップするファイルの種類。文字列の配列で指定。拡張子かファイルタイプ。
aRect ： ビューの中のファイルの場所の指定。
sourceObj ： ドラッグ元の処理を行うインスタンス。NSDraggingSourceを実装していること。
slideBack ： ドロップを拒否された場合にアニメーションするかしないか。
event ： マウスダウン時のイベント情報。

出力

返り値 ： 成功したら - YES、失敗したら - NO。

これで、ドラッグは開始できました。次は、ドロップが行われた時に行う処理です。

```
- (NSArray *) namesOfPromisedFilesDroppedAtDestination : (NSURL *) dropDestination {

  NSString *sFile  = @"PROMISE.txt";  // 書き出すファイル名
  NSString *sText  = @"HELLO";        // ファイルの中身

  // 書き出し先のURLを作成
  NSURL    *urlDst = [ NSURL URLWithString : sFile
                             relativeToURL : dropDestination ];

  [ sText writeToURL : urlDst atomically : NO ]; // ファイル書き出し

  return( [ NSArray arrayWithObject : sFile ] ); // 書き出したファイルリストを返す
}
```

先程、self で渡したインスタンスにこのメソッドが必要になります。ドロップ先にドロップが受け入れられたら、このメソッドが呼ばれます。ドロップ先のフォルダーのパスが URL として渡されますので、その URL のパスにファイル名をくっつけて（ URLWithString ： relativeToURL ：)、そこにファイルを書き出します。完了しましたら、書き出したファイルのリストを配列で返します。

NSDraggingSource : ドロップが受け入れられた時に呼ばれるメソッド

書式
```
- (NSArray *) namesOfPromisedFilesDroppedAtDestination :
                (NSURL *) dropDst
```
入力
dropDst : ドロップ先のフォルダーのURL
出力
返り値 : 作成したファイル名の配列 (フルパスではない)。

■ 配列操作 : NSArray

NSArray にも地味ながらも変更点があります。「 **initWithArray : copyItems :** 」メソッドは、既存の配列を使って新たな配列を初期化するもので、「 initWithArray : 」の拡張版です。もとになる配列の中身を複製して要素に加えるか、複製せずに加えるかが選べます。実は、10.0 の頃から存在はしていたようです。

NSArray : 既存の配列を使って配列を初期化

書式
```
- (id) initWithArray : (NSArray *) array
            copyItems : (BOOL) flag
```
入力
array : もとになる配列
copyItems : 配列の要素を複製してから加える - YES、複製せずに加える - NO
出力
返り値 : 初期化された配列のインスタンス
備考
flag = YESのときは、元の配列の要素に対して copyWithZone : を呼んで複製を作る。
flag = NOのときは、retain を呼んで保持する。

NSMutableArray には、要素の入れ替えを行うメソッドが追加されました。

NSMutableArray : 2つの要素を入れ替える

書式
```
- (void) exchangeObjectAtIndex : (unsigned) index1
              withObjectAtIndex : (unsigned) index2
```
入力
index1 : 入れ替えたい要素のインデックス
index2 : index1と入れ替えたい要素のインデックス

■ スレッドのプライオリティ : NSThread

カレントのスレッドのプライオリティを変更できるようになりました。ゆっくり処理してもいいもの、大急ぎ

で処理したいものなどに応じて使い分けられます。

NSThread ： カレントのスレッドのプライオリティを変更

書式
```
+ (BOOL) setThreadPriority : (double) priority
```
入力
priority ： カレントスレッドのプライオリティ (低 ： 0.0 〜 1.0 ： 高)

NSThread ： カレントのスレッドのプライオリティを取得

書式
```
+ (double) threadPriority
```
出力
返り値 ： カレントスレッドのプライオリティ (低 ： 0.0 〜 1.0 ： 高)

■ バージョンチェック

様々な拡張や変更がなされていますので、OS のバージョンをチェックが必要になることがあります。現在の Application Kit のバージョンは「 **NSAppKitVersionNumber** 」で確認できます。

バージョン情報関連定義

```
APPKIT_EXTERN double NSAppKitVersionNumber;

#define NSAppKitVersionNumber10_0 577  // 10.0でのAppKitのバージョン
#define NSAppKitVersionNumber10_1 620  // 10.1でのAppKitのバージョン
```

これらを利用して、バージョンごとに処理を分ける際には以下のように書きます。

```
 if ( floor( NSAppKitVersionNumber ) <= NSAppKitVersionNumber10_0 ) {
    // 10.0.x
} else if ( floor( NSAppKitVersionNumber ) <= NSAppKitVersionNumber10_1 ) {
    // 10.1.x
} else {
    // 10.2.0以降
}
```

■ ファイル名とフォルダー名のローカライズ

Cocoa というよりも Mac OS X 自身の話になりますが、重要な事項ですので、「 Inside Mac OS X : System Overview 」に記述されている「 **Localizing File System Names** 」にも触れておきます。この話題については、というドキュメントに解説が行われています。

■ 概要

Mac OS 9 までは、フォルダーやアプリケーションなどのファイルには、利用している言語に応じた名前がついていました (例. " 書類 "、" 計算機 ") 。それが、Mac OS X になって全てが英語表記になってしまいました。これは、OS がマルチユーザでマルチリンガル対応になったことが主な原因です。ある日本人のユーザーがログインしている場合はフォルダー名を日本語で表示し、別のアメリカ人がログインした場合には英語で表示する必要があるわけですが、そういった仕掛けが用意されていなかったのです。

そのため、Mac OS X 10.2 では、**フォルダーやファイルの名前をログインしているユーザーの言語情報をもとに切り替える**といった仕組みを導入しました。フォルダーとファイルの名前と書いていますが、ファイルについては**パッケージ形式になっているものに限られます**。パッケージというのは、見た目はファイルですが実態はフォルダーですので、ファイルシステムレベルで見るとフォルダー名をローカライズする仕掛けということになります。

基本的な仕組みですが、フォルダーやファイルに対して**「 ローカライズしている 」という情報と「 英語ならこの名前を使う 」とか「 日本語ならこの名前を使う 」といった情報を与えておく**というものです。ファイルシステムレベルの名前とは別に**「 表示用の名前 」**というものが定義できるのです。

■ アプリケーション名のローカライズ

アプリケーションの場合は、各言語用のリソースが収められているフォルダ「 `xxx.lproj` 」の中に「 **`InfoPlist.strings`** 」というファイルを置いておき、その中に

```
"CFBundleDisplayName" = "その言語用の名前";
```

と書いておきます。これで、その言語用の名前が定義されます。また、アプリケーションの直下の「 **`Info.plist`** 」の中にも同様の定義ができますが、これはデフォルトの名前の定義になります。

`Info.plist` に定義するのは、Project Builder を使います。

アプリケーションの構造

```
🗀 APPLICATION.app
    🗀 Contents
      🗎 Info.plist 「 "CFBundleDisplayName" = "デフォルトの名前" 」
      🗀 Resources
        🗀 English.lproj
          🗎 InfoPlist.strings 「 "CFBundleDisplayName" = "英語用の名前"; 」
        🗀 Japanese.lproj
          🗎 InfoPlist.strings 「 "CFBundleDisplayName" = "日本語用の名前"; 」
```

■ フォルダー名 (一般) のローカライズ

フォルダー名を言語別に用意するためには、まず、**フォルダー名の後ろに「 .localized 」という拡張子をつけます**。これで、フォルダー名からローカライズされているかどうかを判断できます。この拡張子は隠されるためユーザーの目に触れることは基本的にありません。

そして、そのフォルダーの中に「 **.localized** 」というフォルダーを作り、その中に各言語用の定義ファイルをおきます。定義ファイルの名前は、英語用が「 **en.strings** 」、日本語用が「 **ja.strings** 」のようになっています。やや階層が深くなっていますが、定義ファイルは他言語対応するとファイル数が多くなるため、ローカライズしたいフォルダーの直下に置くのはやや問題があるからでしょう。また、「 **.localized** 」というフォルダーは、名前の最初がピリオドですので非表示になります。これによって、全ての定義ファイルが隠されることになります。

各定義ファイルの中身ですが、ja.strings ならば以下のようになります。

```
"日本語用の名前" = "Localized name"
```

フォルダーの構造

```
🗀 FOLDER.localized
    🗀 .localized
      🗎 en.strings 「 "英語用の名前" = "Localized name" 」
      🗎 ja.strings 「 "日本語用の名前" = "Localized name" 」
```

■ フォルダー名 (システム用) のローカライズ

「 ユーザ 」や「 書類 」といったフォルダーのようにシステムのインストールによって作られるものは別の仕掛けが用意されています。以下のフォルダーの中に表示用の名前が定義されています。

```
/System/Library/CoreServices/SystemFolderLocalizations/
```

日本語用の定義は、さらにこのフォルダーの中の

```
ja.lproj/SystemFolderLocalization.strings
```

にあります。中身は以下のようになっています。

```
/* Top-level folders */

"System" = "システム";
"Applications" = "アプリケーション";
"Library" = "ライブラリ";
"Users" = "ユーザ";
"Shared" = "共有";
"Utilities" = "ユーティリティ";

/* Folders in user homes */

"Desktop" = "デスクトップ";
"Documents" = "書類";
"Movies" = "ムービー";
"Music" = "ミュージック";
"Pictures" = "ピクチャ";
"Public" = "パブリック";
"Sites" = "サイト";
"Drop Box" = "ドロップボックス";
```

「 フォント 」や「 サウンド 」や「 スクリーンセーバー 」などがないことも分かります。左側にあるのが置き換え対象となるフォルダーの名前で右側が表示用の名前です。フォルダーの中に「 **.localized** 」という名前のファイルがあるときにこの置き換え機構が働きます。

フォルダーの構造 （ システム ）

```
🗀 FOLDER
    🗎 .localized
```

なぜ、フォルダーに関して 2 つの方式を導入したかを推測してみます。フォルダー名からローカライズされているかどうかを直接判定できる方が速度面で有利なので「 FOLDER.localized 」という形を採用した。けれども、「 /Users 」フォルダーなどの名前を「 /Users.localized 」のように変えるのは、既存のソフトで動かなくなるものがありそうだということで、これに関しては特殊ルールを作った…ということでしょうか。

■ Cocoa での表示用文字列の取得

Cocoa で表示用の文字列を取得する方法ですが、NSFileManager の displayNameAtPath : メソッドを使います。

```
[ [ NSFileManager defaultManager ] displayNameAtPath : @" ファイルパス " ]
```

これは、Mac OS X 10.1 で拡張子を隠すかどうかの設定がファイル毎に出来るようになったために、既に追加されていたメソッドです。このメソッドの挙動がこのローカライズの仕掛けを反映するようになったために、displayNameAtPath : を使っていれば特別な対応は必要ありません。

それから、ちょっと便利なメソッドが追加されました。「 **componentsToDisplayForPath :** 」というメソッドです。これは、**表示用のファイルパス**を得るために使います。displayNameAtPath : は、ファイルパスの最後の部分 (コンポーネント) について表示用の名前を返してくるので、ファイルパスを得ようとするとファイルパスをコンポーネントに分解して...という処理が必要でした。

これからは、componentsToDisplayForPath : でコンポーネントの配列を得ることができます。

```
[ [ NSFileManager defaultManager ]
        componentsToDisplayForPath : NSHomeDirectory() ]
```

上のコードを実行すると、ホームディレクトリの表示用パスの配列が得られます。以下のようになります。

```
0 : < 起動ボリューム名 >
1 : ユーザ
2 : < ユーザ名 >
```

配列をパスの文字列にするには、NSArray の「 **componentsJoinedByString :** 」メソッドを使います。componentsJoinedByString : というのは「 配列に入っている文字列を順に繋ぎあわせる 」というメソッドです (厳密には、配列に入っているインスタンスの description メソッドの返り値を繋ぎあわせるので、文字列以外のインスタンスでも使用できます)。繋ぐ時に間に挟む文字列を指定できます。以下のようになります。

```
[ [ [ NSFileManager defaultManager ]
        componentsToDisplayForPath : NSHomeDirectory() ]
          componentsJoinedByString : @":" ]
```

■ その他

・**Inkwell 対応**（NSResponder）: Inkwell に対応するための拡張が行われています。
・**アクセシビリティ API**（NSCell）: 画面上の文字の読み上げなどの機能に対応しました。独自の Cell を実装している場合は、NSAccessibility プロトコルを実装します。
・**ムービー編集のアンドゥ**（NSMovieView）: NSMovieView が NSUndoManager に対応してムービー編集の多段階アンドゥが可能になりました。
・**タブの位置**（NSTabView）: タブビューのタブの位置は上下左右から選べるようになりました。
・**サウンド再生が QuickTime に対応**（NSSound）: サウンド再生のために QuickTime を利用するようになったため、対応する音声フォーマットが増え、ストリーミング再生も可能になりました。
・**ランデブー関連** ：**NSNetService** や **NSNetServiceBrowser** というランデブー関連のクラスが追加されました。

・**フォントパネルにプレビュー機能**（NSFontPanel）: フォントパネルには、フォントのプレビュー機能がつきました。API としての変化はないようです。
・**カラーパネル**（NSColorPanel）: カラーパネルにカラーピッカーのような**クレヨンモード**が追加されました。また、カラーパネルから「 Apply 」ボタンが取り除かれ、changeColor : メソッドは常に連続的に呼ばれるようになりました。

・**Carbon からの Cocoa の利用**：今まで Cocoa から Carbon は呼べましたが、逆は出来ませんでした。Jaguar では、Carbon から Cocoa が呼べるようになるため、フォントパネルやカラーパネルも Carbon アプリから利用出来ます。
・**AddressBook.framekwork** : システムワイドデータベースになったアドレス帳へアクセスするためのフレームワークが追加されました。
・**DiscRecording APIs** : CD-R や DVD-R への書き込みを行うための API を Cocoa から利用できるようになりました。

・**ドックメニューの改善** ：ドックのメニューからアクションが呼ばれた時にメソッドのパラメータの sender には NSApp の値がセットされていましたが、選ばれたメニューの NSMenuItem が渡るようになりました。

以上