
Low-Level File Management Programming Topics

Data Management: File Management



2009-07-31



Apple Inc.
© 1997, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Carbon, Cocoa, Mac, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries.

Finder and iPhone are trademarks of Apple Inc.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY,

MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction to Low-Level File Management Programming Topics 5

Organization of This Document 5
See Also 5

File Management Classes 7

NSFileManager 7
NSWorkspace 7
NSURL 8
NSString 8

Creating Paths and Locating Directories 9

Creating Paths 9
 URL-Based Path Utilities 9
 String-Based Path Utilities 10
Standard System Directories 10
 Locating Directories as URLs 11
 Locating Directories as Paths 11
Dealing with Broken Links 12

File Management 13

Moving, Copying, and Deleting Files 13
Finder-Like Operations 14
 Moving Files to the Trash 14
 Duplicating Files 14

Information about Files and Volumes 15

Getting and Setting File Attributes 15
Getting Information about Volumes 16

Using URLs 17

Creating File Reference URLs 17
Working with Bookmarks and Aliases 17

Working with the Contents of a Directory 19

Listing the Contents of a Directory 19

Contents as URLs 19
Contents as Path Strings 20
Using a Directory Enumerator 20

Resolving Aliases 23

File Handle 25

Overview 25
Creating a File Handle Object 25
Background Inter-Process Communication Using Sockets 26

HFS File Types 29

Document Revision History 31

Introduction to Low-Level File Management Programming Topics

This document describes methods and functions for manipulating files and directories (folders).

You should read this document to learn how to:

- Represent file paths in Cocoa
- Perform basic operations with files and directories
- Find standard directories on the system

Organization of This Document

This document contains the following articles:

- [“File Management Classes”](#) (page 7) describes the main classes you use for file-system operations.
- [“Creating Paths and Locating Directories”](#) (page 9) explains how to compose paths used in file-system operations.
- [“File Management”](#) (page 13) describes how you can perform many generic file-system operations.
- [“Information about Files and Volumes”](#) (page 15) describes how you can get and set information about files and volumes.
- [“Using URLs”](#) (page 17) describes how to use `NSURL` for various file-related operations.
- [“Working with the Contents of a Directory”](#) (page 19) describes how to retrieve the contents of a directory using an `NSDirectoryEnumerator` object.
- [“Resolving Aliases”](#) (page 23) describes how to resolve aliases in a file path.
- [“File Handle”](#) (page 25) describes the class that provides a wrapper for accessing open files or communications channels.
- [“HFS File Types”](#) (page 29) describes the APIs that allow a Cocoa application to retrieve and create HFS type and creator codes.

See Also

To learn how to use file wrappers (and the `NSFileWrapper` class), see *Application File Management*.

File Management Classes

This article describes the main classes you use to interact with files, volumes, and the Finder.

NSFileManager

You use an `NSFileManager` object to perform many generic file-system operations—for example you can:

- Create directories and files.
- Extract the contents of files (as `NSData` objects).
- Change your current working location in the file system.
- Copy, move, and link files and directories.
- Remove files, links, and directories.
- Get and (where appropriate) set the attributes of a file, a directory, or the file system.
- Determine the contents of directories.
- Compare files and directories for equality.
- Make and evaluate symbolic links.

Besides offering a useful range of generic functionality, the `NSFileManager` class insulates an application from the underlying file system. An important part of this insulation is the encoding of filenames (in, for example, Unicode, ISO Latin1, and ASCII). There is a default `NSFileManager` object for the file system; this object responds to all messages that request a operation on the associated file system.

The pathnames specified as arguments to `NSFileManager` methods can be absolute or relative to the current directory (which you can determine with `currentDirectoryPath` and set with `changeCurrentDirectoryPath:`). However, pathnames cannot include wildcard characters.

You can set a delegate for a file manager; the delegate allows you to modify the behavior of the manager after various events have occurred (for example `fileManager:shouldProceedAfterError:removingItemAtPath:` or—in Mac OS X v10.6 and later—`fileManager:shouldProceedAfterError:removingItemAtURL:`).

NSWorkspace

You use `NSWorkspace` to:

- Open, manipulate, and obtain information about files and devices.
- Track changes to the file system, devices, and the user database.

- Get and set Finder information for files.
- Launch applications.

NSURL

The `NSURL` class provides a way to manipulate URLs *and the resources they reference*. You can use `NSURL` objects to refer to files, and are the preferred way to do so. Particularly in Mac OS X v10.6 and later, objects that can read data from or write data to a file generally have methods that accept an `NSURL` object instead of a pathname as the file reference. URL-based operations are typically much more efficient than path-based equivalents.

NSString

You can use `NSString` objects to represent paths. `NSString` provides a number of utility methods to allow you to manipulate paths, for example to find the file name or path extension. In Mac OS X v10.6 and later, you should use `NSURL` objects rather than strings to represent paths—URLs are typically much more efficient.

Creating Paths and Locating Directories

This article describes how to create URLs and strings that represent paths, and how to locate standard directories in the filesystem.

Creating Paths

`NSString` provides a number of path-utility methods that you can use to, for example, extract the components of a path (the directory, filename, and extension), create paths from those components, “translate” path separators, clean up paths containing symbolic links and redundant slashes, and perform similar tasks.

In Mac OS X v10.6 and later, `NSURL` provides a number of path-utility methods analogous to those provided by `NSString`. Where possible, you should typically use URL-based method rather than string-based methods, since the URL-based operations are usually much more efficient than the string-based equivalents. Rather than manipulating paths as strings, therefore, where possible you should use `NSURL` objects directly.

Whether you use `NSString` or `NSURL`, whenever you need to perform any operation on a path, you should use these methods rather than any other approach.

Absolute and Relative Paths: An absolute pathname starts with the root directory of the file system, represented by a slash (/), and ends with the file or directory that the pathname identifies. A relative pathname is relative to the current directory, the directory in which you are working and in which saved files are currently stored (if no pathname is specified). Relative pathnames start with a subdirectory of the current directory—without an initial slash—and end with the name of the file or directory the pathname identifies.

URL-Based Path Utilities

In Mac OS X v10.6 and later, you should typically use `NSURL`-based APIs to perform file-related operations. The following code fragment shows how, given a URL that contains a file path, you can determine the filename and path extension, remove the filename to determine the directory that contains the file, and create a new file URL in which the original file name is prepended by “Copy of”. (For example, if the original path were `/Users/me/MyFile.txt`, the new path would be `/Users/me/Copy of MyFile.txt`.)

```
NSURL *url = <#URL containing a file path#>;

NSString *extension = [url pathExtension];
NSString *fileName = [[url lastPathComponent] stringByDeletingPathExtension];
NSString *copyFileName = [@"Copy of " stringByAppendingString:fileName];

NSURL *copyURL = [url URLByDeletingLastPathComponent];
copyURL = [copyURL URLByAppendingPathComponent:copyFileName];
copyURL = [copyURL URLByAppendingPathExtension:extension];
```

(To create a copy of a file in the same way that Finder does, see [“Moving Files to the Trash”](#) (page 14).)

String-Based Path Utilities

The following code fragment shows how, given a string that contains a file path, you can determine the filename and path extension, remove the filename to determine the directory that contains the file, and create a new path in which the original file name is prepended by “Copy of”. (For example, if the original path were `/Users/me/MyFile.txt`, the new path would be `/Users/me/Copy of MyFile.txt`.)

```
NSString *path = <#String containing a file path#>;

NSString *extension = [path pathExtension];
NSString *fileName = [[path lastPathComponent] stringByDeletingPathExtension];
NSString *copyFileName = [@"Copy of " stringByAppendingString:fileName];

NSString *copyPath = [path stringByDeletingLastPathComponent];
copyPath = [copyPath stringByAppendingString:copyFileName];
copyPath = [copyPath stringByAppendingPathExtension:extension];
```

(To create a copy of a file in the same way that Finder does, see [“Duplicating Files”](#) (page 14).)

Standard System Directories

Mac OS X and iPhone OS define a number of standard directories—for example directories to contain a user’s documents or system frameworks. Obviously the location of some of these directories—such as the user’s documents directory—is not fixed (it depends on the current user), however, in general you should not rely on other directories, such as the system frameworks directory, remaining in the same location across different versions of the operating system. Rather than hard-coding directory paths, therefore you should use one of the functions or methods that help you to locate standard directories in the file system.

Cocoa provides the following functions that return path strings for a few standard directories directly:

<code>NSHomeDirectory</code>	Returns the path to the current user’s home directory.
<code>NSHomeDirectoryForUser</code>	Returns the path to a given user’s home directory.
<code>NSTemporaryDirectory</code>	Returns the path of the temporary directory for the current user.

You can use the returned values in conjunction with `NSString` path utility methods to create other paths, for example:

```
NSString *path = [NSHomeDirectory() stringByAppendingPathComponent:@"MyFile.txt"];
```

For other standard directories, you use the `NSSearchPathForDirectoriesInDomains` function (which returns paths as `NSString` objects) or—in Mac OS X v10.6 and later—the `URLsForDirectory:inDomains:andURLForDirectory:inDomain:appropriateForURL:create:error:` methods of `NSFileManager` (which return URLs). The function and the methods use constants to identify the directory—or directories—you’re interested in. There are two types of constant:

- Constants in the `NSSearchPathDirectory` enumeration that identify the name or type of directory (for example, `Library`, `Documents`, or `Applications`); such as, `NSDocumentDirectory`, `NSDesktopDirectory`, and `NSLibraryDirectory`.

- Bitmask constants in the `NSSearchPathDomainMask` enumeration that identify the file-system domain (User, System, Local, Network) or all domains; such as, `NSUserDomainMask` and `NSLocalDomainMask`.

In general, `NSSearchPathForDirectoriesInDomains` and `URLsForDirectory:inDomains:` may return multiple values if the parameters imply multiple locations. However, you should not make any assumptions as to the number of paths returned. Some domains or locations might be made obsolete over time, or other new domains or locations might be added (while preserving the older ones); in either case, the number of paths in the returned array might increase or decrease. Simply look at all of the returned values if you want to enumerate all of the files, or, if you just want to copy or move a file to a location and multiple paths are returned, use the first one in the array.

Locating Directories as URLs

In Mac OS X v10.6 and later, you can use an `NSFileManager` object to locate standard system directories. Its methods return `NSURL` objects rather than string-based paths.

`URLsForDirectory:inDomains:` is analogous to `NSSearchPathForDirectoriesInDomains`. You specify the type of directory and the domain you want to find and the method returns an array of URLs. The following example illustrates how you can use the method to get a URL for the current user's Documents directory:

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSArray *urls = [fileManager URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask];
if ([paths count] > 0) {
    NSURL *userDocumentsURL = [urls objectAtIndex:0];
    // Implementation continues...
```

`URLsForDirectory:inDomains:` simply returns a URL for the appropriate directory; it does not guarantee that the exists exists. If you want to perform operations such as writing to the directory, you may have to create it first.

`URLForDirectory:inDomain:appropriateForURL:create:error:` is a URL-based replacement for `FSFindFolder`. You can specify and optionally create a directory for a particular purpose (for example, the replacement of a particular item on disk, or a particular Library directory). You may pass only one of the values from the `NSSearchPathDomainMask` enumeration, and you may not pass `NSAllDomainsMask`.

Locating Directories as Paths

The following example illustrates how you can use the `NSSearchPathForDirectoriesInDomains` function to find the current user's Documents directory:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
if ([paths count] > 0) {
    NSString *userDocumentsPath = [paths objectAtIndex:0];
    // Implementation continues...
```

`NSSearchPathForDirectoriesInDomains` simply returns the path of the appropriate directory, it does not guarantee that the directory exists. If you want to perform operations such as writing to the directory, you may have to create it first.

Dealing with Broken Links

Constructing a pathname to a file does not guarantee that the file exists at that path. Specifying a path results in one of the following possibilities:

- A file exists at that path.
- A link to a file exists at that path.
- A broken link exists at that path.
- No file exists at that path.

If the pathname specifies a valid file or link, you can obtain information about the file using the `NSFileManager` method `attributesOfItemAtPath:error:` (see [“Getting and Setting File Attributes”](#) (page 15)). This method does not traverse a link however. You can first call `destinationOfSymbolicLinkAtPath:error:` and then `attributesOfItemAtPath:error:.` In Mac OS X v10.6 and later, you can use NSURL methods `URLByResolvingSymlinksInPath` and `resourceValuesForKeys:error:` (see [“Getting and Setting File Attributes”](#) (page 15)).

File Management

This article describes how you can perform a number of file and file-related operations.

Moving, Copying, and Deleting Files

The following examples show file manipulation using the URL-based methods of `NSFileManager` that are available in Mac OS X v10.6 and later.

To move or rename a file or directory, use `moveItemAtURL:toURL:error:`, as illustrated by this code fragment:

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *srcURL = <#Get the source URL#>;
NSURL *destinationURL = <#Create the destination URL#>;

NSError *error = nil;
if (![fileManager moveItemAtURL:srcURL toURL:destinationURL error:&error]) {
    // Handle the error.
}
```

To copy a file or directory, use `copyItemAtURL:toURL:error:`, as illustrated by this code fragment:

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *srcURL = <#Get the source URL#>;
NSURL *destinationURL = <#Create the destination URL#>;

NSError *error = nil;
if (![fileManager copyItemAtURL:srcURL toURL:destinationURL error:&error]) {
    // Handle the error.
}
```

To delete a file or directory, use `removeItemAtURL:error:`, as illustrated by this code fragment:

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *url = <#Get the URL of the item to delete#>;

NSError *error = nil;
if (![fileManager removeItemAtURL error:&error]) {
    // Handle the error.
}
```

You can also replace a file or directory with another file or directory, renaming the original item with a name to signify that it is a backup using `replaceItemAtURL:withItemAtURL:backupItemName:options:resultingItemURL:error:` (for Carbon developers, this is a replacement for the `FSExchangeObjects` function). You typically need this functionality when saving documents; since `NSDocument` performs this operation itself when saving a document, there may generally be little reason for you to use it yourself.

Finder-Like Operations

In Mac OS X v10.6 and later, `NSWorkspace` provides two methods you can use to duplicate files or move them to the trash in the same way that Finder does.

Mac OS X v10.5 and earlier: To move items to the trash in Mac OS X v10.5 and earlier, you can use the `NSWorkspace` method `performFileOperation:source:destination:files:tag: method specifying NSWorkspaceRecycleOperation as the operation.`

Moving Files to the Trash

You use `recycleURLs:completionHandler:` to move files at the specified URLs to the trash in the same manner as the Finder. The completion handler argument is a block object that is invoked when the operation has finished.

```
NSWorkspace *workspace = [NSWorkspace sharedWorkspace];
NSArray *URLs = <#An array of file URLs#>;
[workspace recycleURLs:URLs completionHandler:^(NSDictionary *newURLs, NSError
*error) {
    if (error != nil) {
        // Deal with any errors here.
    }
}];
```

Duplicating Files

You use `duplicateURLs:completionHandler:` to make copies of files at specified URLs in the same manner as the Finder. The completion handler argument is a block object that is invoked when the operation has finished.

```
NSWorkspace *workspace = [NSWorkspace sharedWorkspace];
NSArray *URLs = <#An array of file URLs#>;
[workspace duplicateURLs:URLs completionHandler:^(NSDictionary *newURLs, NSError
*error) {
    if (error != nil) {
        // Deal with any errors here.
    }
}];
```

Information about Files and Volumes

This article describes how you can get and set information about files and volumes.

Getting and Setting File Attributes

In Mac OS X v10.6 and later, you can use `NSURL` to get and set attributes of the item the URL points to. There are two methods to get attribute values and two methods to set attribute values, to get or set a single value, or to get or set multiple values simultaneously:

<code>getResourceValue:forKey:error:</code>	Returns the resource value for the property identified by a given key.
<code>setResourceValue:forKey:error:</code>	Sets the resource property of the URL specified by a given key to a given value.
<code>resourceValuesForKeys:error:</code>	Returns the resource values for the properties identified by specified array of keys.
<code>setResourceValues:error:</code>	Sets resource properties of the URL specified by a given set of keys to a given set of values.

`NSURL` also defines constants for the keys to specify the attributes, including for example `NSURLNameKey`, `NSURLLocalizedNameKey`, `NSURLIsPackageKey`, `NSURLCreationDateKey`, `NSURLFileSizeKey`, and `NSURLCustomIconKey`.

Note: The values associated with some attributes (for example, `NSURLLabelColorKey` and `NSURLCustomIconKey`) are colors or images, represented by instances of `NSColor` or `NSImage`. These are not available to programs that are not linked against AppKit.

The following examples show how you can get and set single and multiple attributes:

```
NSURL *url = <#Get a URL#>;
NSError *error = nil;
BOOL ok;

// Get the item's label color
NSColor *labelColor;
ok = [url getResourceValue:&labelColor forKey:NSURLLabelColorKey error:&error];
if (!ok) {
    // Handle the error.
}

// Make the item hidden
ok = [url setResourceValue:[NSNumber numberWithBool:YES] forKey:NSURLIsHiddenKey
error:&error];
```

```

if (!ok) {
    // Handle the error.
}

// Get numerous properties
NSArray *keys = [NSArray arrayWithObjects:NSURLNameKey, NSURLLocalizedNameKey,
                                                NSURLIsRegularFileKey,
                                                NSURLIsDirectoryKey,
                                                NSURLIsSymbolicLinkKey,
                                                NSURLIsPackageKey, nil];
NSDictionary *properties = [url resourceValuesForKeys:keys error:&error];
if (properties == nil) {
    // Handle the error.
}

```

Getting Information about Volumes

In Mac OS X v10.6 and later, you can find out about the volumes mounted on the system using `mountedVolumeURLsIncludingResourceValuesForKeys:options:`. The method returns an array of `NSURL` objects containing the resource values corresponding to the keys you specify (if you simply want to find out what volumes are present, you can pass `nil` for the keys argument). The options argument allows you to specify whether you want to omit hidden volumes, and whether the method should return file reference URLs (see [“Creating File Reference URLs”](#) (page 17)).

The following examples shows how to get, for all the available volumes, URLs including the localized name and the default icon:

```

NSArray *resourceKeys = [NSArray arrayWithObjects:
                        NSURLLocalizedNameKey, NSURLEffectiveIconKey, nil];
NSArray *volumeURLs = [[NSFileManager defaultManager]

mountedVolumeURLsIncludingResourceValuesForKeys:resourceKeys
                    options:0];

```

Using URLs

This article describes various operations you can perform using `NSURL` with Mac OS X v10.6 and later.

Creating File Reference URLs

Sometimes you need a reference to a file that the user might move while your application is running. In this situation, a path-based URL is not useful since the path will change when the user moves the file but the URL will continue to point to the original location. File reference URLs provide a way to track a file by its ID. This means that the reference is valid even if the file's name or location in the file system changes.

You create a file reference URL from an existing URL using the method `fileReferenceURL`:

```
NSURL *existingURL = <#A URL for an existing file or directory#>;
NSURL *fileReferenceURL = [existingURL fileReferenceURL];
```

The representations of the two URLs in the example are different:

```
existingURL = file://localhost/Users/me/MyFile.txt
fileReferenceURL = file:///file/id=6238375.726492
```

Both the standard and file reference URLs are, however, valid URLs.

Two caveats apply to file reference URLs:

- `NSDocument` already uses file reference URLs internally, so there should typically be no need to manage them yourself in a document-based application.
- You should not store or archive file reference URLs. A file's ID may be different for different boots of the operating system. If you need to store a URL, see [“Working with Bookmarks and Aliases”](#) (page 17).

Working with Bookmarks and Aliases

You can create a persistent representation of a URL using `bookmarkDataWithOptions:includingResourceValuesForKeys:relativeToURL:error:`. The `options` argument allows you to specify a number of aspects of the bookmarked URL, including file alias information. (This allows you to read a bookmark as an alias on versions of Mac OS X prior to 10.6.)

```
NSURL *url = <#Create a URL#>;
NSError *error = nil;
NSData *bookmarkData = [url
    bookmarkDataWithOptions:NSURLBookmarkCreationSuitableForBookmarkFile
                        includingResourceValuesForKeys:nil
                        relativeToURL:nil];
```

```
                                error:&error];  
if (bookmarkData == nil) {  
    // Handle the error...  
}
```

If you create a bookmark using the `NSURLBookmarkCreationSuitableForBookmarkFile` option, you can subsequently create an alias file from the bookmark using the class method `writeBookmarkData:toURL:options:error:.`

```
NSURL *bookmarkURL = <#Create a URL for the bookmark#>;  
BOOL ok = [NSURL writeBookmarkData:bookmarkData toURL:bookmarkURL options:0  
error:&error];  
if (!ok) {  
    // Handle the error...  
}
```

If the bookmark URL points to a directory, an alias file is created in that directory with its name derived from the information in the bookmark data. If the URL points to a file, the alias file is created with the location and name specified by the bookmark URL, and its extension changed to `.alias` if it is not already.

You can recreate a URL from bookmark data using `URLByResolvingBookmarkData:options:relativeToURL:bookmarkDataIsStale:error:.`

Working with the Contents of a Directory

This article describes two approaches to working with the contents of a directory. `NSFileManager` provides several methods that return arrays of paths or URLs for items in a directory. To iterate over the contents of a directory, you can use an `NSDirectoryEnumerator` object.

Listing the Contents of a Directory

`NSFileManager` provides several methods that return the contents of a directory as an array of paths or (in Mac OS X v10.6 and later) URLs.

Contents as URLs

In Mac OS X v10.6 and later, you can obtain an array of `NSURL` objects for each of the top-level items in a directory using `contentsOfDirectoryAtURL:includingPropertiesForKeys:options:error:`. (This is the URL-based equivalent of `contentsOfDirectoryAtPath:error:`. If you need to recurse into subdirectories, use `enumeratorAtURL:includingPropertiesForKeys:options:errorHandler:` as shown in [“Using a Directory Enumerator”](#) (page 20)). If you’re only interested in the URLs and no other attributes, then pass an empty array for the keys and 0 for the options, as shown in this example:

```
NSURL *url = <#A URL for a directory#>;
NSError *error = nil;
NSArray *array = [[NSFileManager defaultManager]
                 contentsOfDirectoryAtURL:url
                 includingPropertiesForKeys:[NSArray array]
                 options:0
                 error:&error];
if (array == nil) {
    // Handle the error
}
```

One of the benefits of using URLs, however, is that you can also efficiently retrieve additional information about each item. If you want to have the property caches of the returned URLs pre-populated with a default set of attributes, then pass `nil` for the keys and 0 for the options. You can also specify which attributes to retrieve and options to omit subdirectories, the contents of file packages, and hidden files, as illustrated in the following example:

```
NSURL *url = <#A URL for a directory#>;
NSError *error = nil;
NSArray *properties = [NSArray arrayWithObjects: NSURLLocalizedNameKey,
                      NSURLCreationDateKey, NSURLLocalizedTypeDescriptionKey,
                      nil];

NSArray *array = [[NSFileManager defaultManager]
                 contentsOfDirectoryAtURL:url
```

```

        includingPropertiesForKeys:properties
        options:(NSDirectoryEnumerationSkipsPackageDescendants |
                 NSDirectoryEnumerationSkipsHiddenFiles)
        error:&error];
if (array == nil) {
    // Handle the error
}

```

Contents as Path Strings

If you simply want a list of the contents of a directory, excluding any subdirectories (and without traversing symbolic links), you use `contentsOfDirectoryAtPath:error:`, as shown in this example:

```

NSString *path = <#A path to a directory#>;
NSError *error = nil;
NSArray *array = [[NSFileManager defaultManager]
                  contentsOfDirectoryAtPath:path error:&error];
if (array == nil) {
    // Handle the error
}

```

If you need a recursive listing—one that contains the filenames of the items in a given directory and all its subdirectories—you can use `subpathsOfDirectoryAtPath:error:`, as shown in this example:

```

NSString *path = <#A path to a directory#>;
NSError *error = nil;
NSArray *array = [[NSFileManager defaultManager]
                  subpathsOfDirectoryAtPath:path error:&error];
if (array == nil) {
    // Handle the error
}

```

Using a Directory Enumerator

You use an `NSDirectoryEnumerator` object to enumerate the contents of a directory and any subdirectories that it contains. `NSDirectoryEnumerator` is an abstract class, a cover for a private concrete subclass tailored to the file system's directory structure. You cannot create an `NSDirectoryEnumerator` object directly—you use `NSFileManager` to retrieve a suitable instance.

You use the `NSFileManager` method `enumeratorAtPath:` to retrieve a directory enumerator that returns items as file path strings. In Mac OS X v10.6 and later, you can use a directory enumerator that returns items as URLs. (If you're going to perform any operations with the returned items, it's typically more efficient to use URLs.) You retrieve such an instance from a file manager object using `enumeratorAtURL:includingPropertiesForKeys:options:errorHandler:`.

Both enumerators return the paths of all files and directories contained within that directory. The paths are relative to the directory. The enumeration is recursive, including the files of all subdirectories, and crosses device boundaries. It does not resolve symbolic links or attempt to traverse symbolic links that point to directories. Typically you simply enumerate the items in the `NSDirectoryEnumerator` object. You can also, though, use the `skipDescendents` method to avoid listing the contents of any directories you're not interested in.

The following URL-based example (for Mac OS X v10.6 and later) illustrates how you can use `enumeratorAtURL:includingPropertiesForKeys:options:errorHandler:` to list all the user-visible subdirectories of a given directory, noting whether they are directories or file packages. The keys array argument specifies that for each URL produced by this enumeration the specified property values are pre-fetched and cached—this makes subsequent access more efficient. The options argument specifies that enumeration should not list the contents of file packages and hidden files. The error handler is a block object that returns a Boolean value; if it returns YES, the enumeration continues after the error; if it returns NO, the enumeration stops.

```

NSURL *directoryURL = <#An NSURL object that contains a reference to a
directory#>;

NSArray *keys = [NSArray arrayWithObjects:
    NSURLIsDirectoryKey, NSURLIsPackageKey, NSURLLocalizedNameKey, nil];

NSDirectoryEnumerator *enumerator = [[NSFileManager defaultManager]
    enumeratorAtURL:directoryURL
    includingPropertiesForKeys:keys

options:(NSDirectoryEnumerationSkipsPackageDescendants |
NSDirectoryEnumerationSkipsHiddenFiles)
    errorHandler:^(NSURL *url, NSError *error)
{
    // Handle the error.
    // Return YES if the enumeration should
    continue after the error.
    return <#YES or NO#>;
}];

for (NSURL *url in enumerator) {

    // Error-checking is omitted for clarity.

    NSNumber *isDirectory = nil;
    [url getResourceValue:&isDirectory forKey:NSURLIsDirectoryKey error:NULL];

    if ([isDirectory boolValue]) {

        NSString *localizedName = nil;
        [url getResourceValue:&localizedName forKey:NSURLLocalizedNameKey
error:NULL];

        NSNumber *isPackage = nil;
        [url getResourceValue:&isPackage forKey:NSURLIsPackageKey error:NULL];

        if ([isPackage boolValue]) {
            NSLog(@"Package at %@", localizedName);
        }
        else {
            NSLog(@"Directory at %@", localizedName);
        }
    }
}

```

You can use other methods declared by `NSDirectoryEnumerator` to determine attributes of files during the enumeration—both of the parent directory and the current file or directory—and to control recursion into subdirectories. The following string-based example enumerates the contents of a directory and lists files that have been modified within the last 24 hours; if, however, it comes across RTFD file packages, it skips recursion into them:

```
NSString *directoryPath = <#Get a path to a directory#>;
NSDirectoryEnumerator *directoryEnumerator = [[NSFileManager defaultManager]
enumeratorAtPath:directoryPath];

NSDate *yesterday = [NSDate dateWithTimeIntervalSinceNow:(-60*60*24)];

for (NSString *path in directoryEnumerator) {

    if ([[path pathExtension] isEqualToString:@"rtfd"]) {
        // Don't enumerate this directory.
        [directoryEnumerator skipDescendents];
    }
    else {

        NSDictionary *attributes = [directoryEnumerator fileAttributes];
        NSDate *lastModificationDate = [attributes
objectForKey:NSFileModificationDate];

        if ([yesterday earlierDate:lastModificationDate] == yesterday) {
            NSLog(@"%@ was modified within the last 24 hours", path);
        }
    }
}
```

Resolving Aliases

Unlike symbolic links, Mac OS X aliases are not handled automatically by Cocoa. This article explains how to resolve aliases in a path.

Paths returned by Cocoa classes such as `NSOpenPanel` may contain alias references that you must resolve before using. To resolve aliases in a path contained in an `NSString`, you need to first convert the string to a URL, convert the URL to an `FSRef`, resolve the alias, and reverse the series of conversions to yield another `NSString`. To perform the necessary conversions, you need to use the URL and alias services provided in `<CoreServices/CoreServices.h>`. The following code fragment uses `FSResolveAliasFile` to resolve any aliases in path and stores the resolved path in `resolvedPath`:

```
NSString *path = <#Get a suitable path#>;
NSString *resolvedPath = nil;

CFURLRef url = CFURLCreateWithFileSystemPath
    (kCFAllocatorDefault, (CFStringRef)path, kCFURLPOSIXPathStyle,
    NO);
if (url != NULL)
{
    FSRef fsRef;
    if (CFURLGetFSRef(url, &fsRef))
    {
        Boolean targetIsFolder, wasAliased;
        OSErr err = FSResolveAliasFile (&fsRef, true, &targetIsFolder,
        &wasAliased);
        if ((err == noErr) && wasAliased)
        {
            CFURLRef resolvedUrl = CFURLCreateFromFSRef(kCFAllocatorDefault,
            &fsRef);
            if (resolvedUrl != NULL)
            {
                resolvedPath = (NSString*)
                    CFURLCopyFileSystemPath(resolvedUrl,
            kCFURLPOSIXPathStyle);
                CFRelease(resolvedUrl);
            }
        }
        CFRelease(url);
    }

    if (resolvedPath == nil)
    {
        resolvedPath = [[NSString alloc] initWithString:path];
    }
}
```

The second argument to `FSResolveAliasFile` specifies whether you want the function to resolve all aliases in a chain (for example, an alias file that refers to an alias file and so on), stopping only when it reaches the target file.

File Handle

`NSFileHandle` objects provide an object-oriented wrapper for accessing open files or communications channels.

Overview

An `NSFileHandle` is an object that represents an open file or communications channel. It enables programs to read data from or write data to the represented file or channel. You can use other Cocoa methods for reading from and writing to files—`NSFileManager`'s `contentsAtPath:` and `NSData`'s `writeToFile:options:error:` are but a couple of examples. Why would you use `NSFileHandle` then? What are its advantages?

- `NSFileHandle` gives you greater control over input/output operations on files. It allows more manipulative operations on and within open files, such as seeking, truncating, and reading and writing at an exact position within a file (the file pointer). Other methods read or write a file in its entirety; with `NSFileHandle`, you can range over an open file and insert, extract, and delete data.
- The scope of `NSFileHandle` is not limited to files. It provides the only Foundation object that can read and write to communications channels such as those implemented by sockets, pipes, and devices.
- `NSFileHandle` makes possible asynchronous background communication. With it a program can connect to, and read from, a socket in a separate thread. (See [“Background Inter-Process Communication Using Sockets”](#) (page 26) below for details on how this is done.)

Instances of `NSPipe`, a class closely related to `NSFileHandle`, represent pipes: unidirectional interprocess communication channels. See the `NSPipe` reference for details.

Creating a File Handle Object

`NSFileHandle` is a class clusters, file handle objects are not actual instances of the `NSFileHandle` class but of one of its private subclasses. Although a file handle object's class is private, its interface is public, as declared by the abstract superclass `NSFileHandle`.

Generally, you instantiate a file handle object by sending one of the `fileHandle...` messages to the `NSFileHandle` class object. These methods return a file handle object pointing to the appropriate file or communications channel. As a convenience, `NSFileHandle` provides class methods that create objects representing files and devices in the file system and that return objects representing the standard input, standard output, and standard error devices. You can also create file handle objects from file descriptors (such as found on BSD systems) using the `initWithFileDescriptor:` and `initWithFileDescriptor:closeOnDealloc:` methods. If you create file handle objects with these methods, you “own” the represented descriptor and are responsible for removing it from system tables, usually by sending the file handle object a `closeFile` message.

Background Inter-Process Communication Using Sockets

Sockets are full-duplex communication channels between processes either local to the same host machine or where one process is on a remote host. Unlike pipes, in which data goes in one direction only, sockets allow processes both to send and receive data. `NSFileHandle` facilitates communication over stream-type sockets by providing mechanisms run in background threads that accept socket connections and read from sockets.

`NSFileHandle` currently handles only communication through stream-type sockets. If you want to use datagrams or other types of sockets, you must create and manage the connection using native system routines.

The process on one end of the communication channel (the server) starts by creating and preparing a socket using system routines. These routines vary slightly between BSD and non-BSD systems, but consist of the same sequence of steps:

1. Create a stream-type socket of a certain protocol.
2. Bind a name to the socket.
3. Adding itself as an observer of `NSFileHandleConnectionAcceptedNotification`.
4. Sending `acceptConnectionInBackgroundAndNotify` to this file handle object. This method accepts the connection in the background, creates a new `NSFileHandle` object from the new socket descriptor, and posts an `NSFileHandleConnectionAcceptedNotification`.

In a method implemented to respond to this notification, the server extracts the `NSFileHandle` object representing the “near” socket of the connection from the notification’s `userInfo` dictionary; it uses the `NSFileHandleNotificationFileHandleItem` key to do this.

Typically the other process (the client) then locates the named socket created by the first process. Instead of accepting a connection to the socket by calling the appropriate system routine, the client creates an `NSFileHandle` object using the socket identifier as argument to `initWithFileDescriptor:`.

The client can now send data to the other process over the communications channel by sending `writeData:` to the `NSFileHandle` instance. (Note that `writeData:` can block.) The client can also read data directly from the `NSFileHandle`, but this would cause the process to block until the socket connection was closed, so it is usually better to read in the background. To do this, the process must:

1. Add itself as an observer of `NSFileHandleReadCompletionNotification` or `NSFileHandleReadToEndOfFileCompletionNotification`.
2. Send `readInBackgroundAndNotify` or `readToEndOfFileInBackgroundAndNotify` to this `NSFileHandle` object. The former method sends a notification after each transmission of data; the latter method accumulates data and sends a notification only after the sending process shuts down its end of the connection.
3. In a method implemented to respond to either of these notifications, the process extracts the transmitted or accumulated data from the notification’s `userInfo` dictionary by using the `NSFileHandleNotificationDataItem` key.
4. If you wish to keep getting notified you’ll need to again call `readInBackgroundAndNotify` in your observer method.

You close the communications channel in both directions by sending `closeFile` to the `NSFileHandle` object; either process can partially or totally close communication across the socket connection with a system-specific shutdown command.

HFS File Types

HFS type codes are replaced by file UTIs (see *Uniform Type Identifiers Overview*). This article is provided for those who still need to use HFS type codes.

HFS file type and creator codes are specified by a 32-bit unsigned integer (`OSType`), usually represented as four characters, such as `GIFf` or `MooV`. Compilers in Mac OS X automatically convert the 4-character representation to the integer representation when they encounter four characters within single quotes, such as `'GIFf'`.

```
OSType aFileType = 'GIFf';
```

You can manipulate HFS type and creator codes using `NSFileManager` as follows:

- To retrieve HFS type and creator codes, you use `attributesOfItemAtPath:error:` to retrieve a dictionary containing the file's attributes, then use the `NSDictionary` methods `fileHFSTypeCode` and `fileHFSCreatorCode` respectively.
- To set type and creator codes, use the `NSFileManager` method `setAttributesOfItemAtPath:error:`, passing a dictionary that contains the `NSFileHFSCreatorCode` and `NSFileHFSTypeCode` keys. The values of these keys should be `NSNumber` objects that hold `OSType` values, for example:

```
NSNumber *aFileType = [NSNumber numberWithIntUnsignedLong:'GIFf'];
```

Other parts of Cocoa, such as `NSOpenPanel`, have traditionally been restricted to specifying file types by filename extension, such as `"gif"` or `"mov"`. Because of the conflicting data types—strings versus integers—these filename-extension APIs cannot use HFS file types directly. These methods can instead accept HFS file types that have been properly encoded into `NSString`.

The Foundation Kit defines the following functions to convert between a classic HFS file type and an encoded string:

<code>NSFileTypeForHFSTypeCode</code>	Returns a string encoding a file type code.
<code>NSHFSTypeCodeFromFileType</code>	Returns a file type code.
<code>NSHFSTypeOfFile</code>	Returns a string encoding a file type.

The first two functions convert between an HFS file type and an encoded HFS string. The final function returns the encoded HFS string for a specific file.

As an example, when specifying the file types that can be opened by an `NSOpenPanel` object, you can create the array of file types as follows to include any text documents with the appropriate HFS type code:

```
NSArray *fileTypes = [NSArray arrayWithObjects:@"txt", @"text",  
NSFileTypeForHFSTypeCode('TEXT'), nil];
```

When you need to find out whether the HFS file type or extension of a particular file is part of a set of HFS file types and extensions, you can use a function similar to this one:

```
BOOL FileIsValid(NSString *fullFilePath)
{
    // Create an array of strings specifying valid extensions and HFS file types.
    NSArray *fileTypes = [NSArray arrayWithObjects:
                           @"txt",
                           @"text",
                           NSFileTypeForHFSTypeCode('TEXT'),
                           nil];

    // Try to get the HFS file type as a string.
    NSString *fileType = NSHFSTypeOfFile(fullFilePath);

    if ([fileType isEqualToString:@""])
    {
        // No HFS type; get the extension instead.
        fileType = [fullFilePath pathExtension];
    }

    return [fileTypes containsObject:fileType];
}
```

For information about setting HFS file types with `NSDocument` subclasses, see [Saving HFS Type and Creator Codes](#).

Document Revision History

This table describes the changes to *Low-Level File Management Programming Topics*.

Date	Notes
2009-07-31	Updated for Mac OS X v10.6.
2009-03-05	Corrected a broken link.
2008-05-05	Added note that FSFindFolder is not available on iPhone.
2008-02-08	Removed references to deprecated Java classes.
2006-07-24	Added code sample showing how to inquire about the HFS file type of a file.
	Added code sample to " HFS File Types " (page 29).
2005-08-11	Corrected example code. Changed title from "Low-Level File Management."
2004-12-02	Corrected typo in "Locating Directories on the System."
2004-08-31	Removed link to missing article and made minor edits.
2002-11-12	Revision history was added to existing document.

