File System Programming Guide

Developer

Contents

Figures, Tables, and Listings

About Files and Directories

The file system is an important part of any operating system. After all, it's where users keep their stuff. The organization of the file system plays an important role in helping the user find files. The organization also makes it easier for apps and the system itself to find and access the resources they need to support the user.

This document is intended for developers who are writing software for OS X, iOS, and iCloud. It shows you how to use the system interfaces to access files and directories, move files to and from iCloud, provides guidance on how best to work with files, and it shows you where you should be placing any new files you create.

Important: When you adopt App Sandbox in your OS X app, the behavior of many file system features changes. For example, to obtain access to locations outside of your app's container directory, you must request appropriate entitlements. To obtain persistent access to a file system resource, you must employ the security-scoped bookmark features of the `NSURL` class or the `CFURLRef` opaque type. There are changes to the location of app support files (which are relative to your container rather than to the user's home folder) and to the behavior of Open and Save dialogs (which are presented by Powerbox, not AppKit). For details on all these changes, read *App Sandbox Design Guide*.

At a Glance

To effectively use the file system, know what to expect from the file system itself and which technologies are available for accessing it.

The File System Imposes a Specific Organization

The file systems in iOS and OS X are structured to help keep files organized, both for the user and for apps. From the perspective of your code, a well-organized file system makes it easier to locate files your app needs. Of course, you also need to know where you should put any files you create.

Relevant chapters and appendixes: "File System Basics" (page 11), "OS X Library Directory Details" (page 91), "File System Details" (page 94)

Access Files Safely

On a multi-user system like OS X, it is possible that more than one app may attempt to read or write a file at the same time as another app is using it. The `NSFileCoordinator` and `NSFilePresenter` classes allow you to maintain file integrity and ensure that if files are made available to other apps (for example, emailing the current TextEdit document) that the most current version is sent.

Relevant Chapter: "The Role of File Coordinators and Presenters" (page 44)

How You Access a File Depends on the File Type

Different files require different treatments by your code. For file formats defined by your app, you might read the contents as a binary stream of bytes. For more common file formats, though, iOS and OS X provide higher-level support that make reading and writing those files easier.

Relevant section: "Choose the Right Way to Access Files" (page 29)

System Interfaces Help You Locate and Manage Your Files

Hard-coded pathnames are fragile and liable to break over time, so the system provides interfaces that search for files in well-known locations. Using these interfaces makes your code more robust and future proof, ensuring that regardless of where files move to, you can still find them.

Relevant chapter: "Accessing Files and Directories" (page 29)

Users Interact with Files Using the Standard System Panels

For files that the user creates and manages, your code can use the standard Open and Save panels to ask for the locations of those files. The standard panels present the user with a navigable version of the file system that matches what the Finder presents. You can use these panels without customization, modify the default behavior, or augment them with your own custom content. Even sandboxed apps can use the panels to access files because they work with the underlying security layer to allow exceptions for files outside of your sandbox that the user explicitly chooses.

Relevant chapter: "Using the Open and Save Panels" (page 58)

Read and Write Files Asynchronously

Because file operations require accessing a disk or network server, you should always access files from a secondary thread of your app. Some of the technologies for reading and writing files run asynchronously without you having to do anything, while others require you to provide your own execution context. All of the technologies do essentially the same thing but offer different levels of simplicity and flexibility.

As you read and write files, you should also use file coordinators to ensure that any actions you take on a file do not cause problems for other apps that care about the file.

Relevant chapter: "Techniques for Reading and Writing Files Without File Coordinators" (page 73)

Move, Copy, Delete, and Manage Files Like the Finder

The system interfaces support all of the same types of behaviors that the Finder supports and many more. You can move, copy, create, and delete files and directories just like the user can. Using the programmatic interfaces, you can also iterate through the contents of directories and work with invisible files much more efficiently. More importantly, most of the work can be done asynchronously to avoid blocking your app's main thread.

Relevant chapter: "Managing Files and Directories" (page 67)

Optimize Your File-Related Operations

The file system is one of the slowest parts of a computer so writing efficient code is important. Optimizing your file-system code is about minimizing the work you do and making sure that the operations you do perform are done efficiently.

Relevant chapter: "Performance Tips" (page 87)

See Also

For information about more advanced file-system tasks that you can perform, see *File System Advanced Programming Topics*.

File System Basics

The file systems in OS X and iOS handle the persistent storage of data files, apps, and the files associated with the operating system itself. Therefore, the file system is one of the fundamental resources used by all processes.

The file systems in OS X and iOS are both based on the UNIX file system. All of the disks attached to the computer—whether they are physically plugged into the computer or are connected indirectly through the network—contribute space to create a single collection of files. Because the number of files can easily be many millions, the file system uses directories to create a hierarchical organization. Although the basic directory structures are similar for iOS and OS X, there are differences in the way each system organizes apps and user data.

Before you begin writing code that interacts with the file system, you should first understand a little about the organization of file system and the rules that apply to your code. Aside from the basic tenet that you cannot write files to directories for which you do not have appropriate security privileges, apps are also expected to be good citizens and put files in appropriate places. Precisely where you put files depends on the platform, but the overarching goal is to make sure that the user's files remain easily discoverable and that the files your code uses internally are kept out of the user's way.

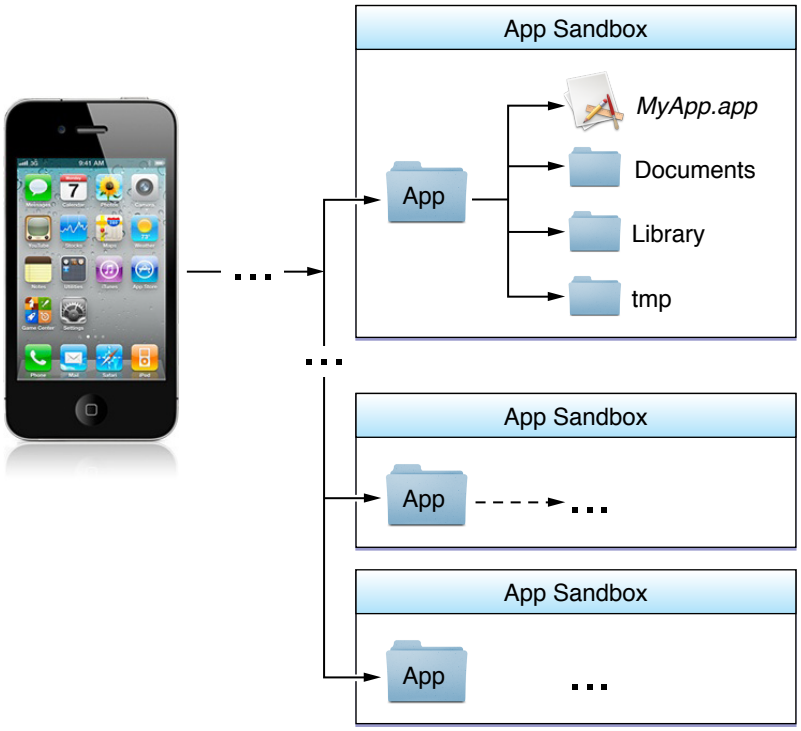
About the iOS File System

The iOS file system is geared toward apps running on their own. To keep the system simple, users of iOS devices do not have direct access to the file system and apps are expected to follow this convention.

Every App Is an Island

An iOS app's interactions with the file system are limited mostly to the directories inside the app's sandbox. During installation of a new app, the installer code creates a home directory for the app, places the app in that directory, and creates several other key directories. These directories constitute the app's primary view of the file system. Figure 1-1 shows a representation of the sandbox for an app.

Figure 1-1 Each iOS app operates within its own sandbox



Because it is in a sandbox, an app is generally prohibited from accessing or creating files in directories outside of its home directory. One exception to this rule occurs when an app uses public system interfaces to access things such as the user's contacts or music. In those cases, the system frameworks handle any file-related operations needed to read from or modify the appropriate data stores.

For a detailed look at the directories inside an app's sandbox, see "iOS Standard Directories: Where Files Reside" (page 13)

iOS Standard Directories: Where Files Reside

For security purposes, an iOS app has limited a number of places where it can write its data. When an app is installed on a device, iTunes creates a home directory for the app. This directory represents the universe for that app and contains everything the app can access directly. Table 1-1 lists some of the important subdirectories of this home directory and describes their intended usage. This table also describes any additional access restrictions for each subdirectory and points out whether the directory's contents are backed up by iTunes.

Table 1-1 Commonly used directories of an iOS app

Directory	Description
<*Application_Home*>`/`*AppName*`.app`	This is the bundle directory containing the app itself. Do not write anything to this directory. To prevent tampering, the bundle directory is signed at installation time. Writing to this directory changes the signature and prevents your app from launching again. In iOS 2.1 and later, the contents of this directory are not backed up by iTunes. However, iTunes does perform an initial sync of any apps purchased from the App Store.
<*Application_Home*>`/Documents/`	Use this directory to store critical user documents and app data files. Critical data is any data that cannot be recreated by your app, such as user-generated content. The contents of this directory can be made available to the user through file sharing. The contents of this directory are backed up by iTunes.
<*Application_Home*>`/Documents/Inbox`	Use this directory to access files that your app was asked to open by outside entities. Specifically, the Mail program places email attachments associated with your app in this directory; document interaction controllers may also place files in it. Your app can read and delete files in this directory but cannot create new files or write to existing files. If the user tries to edit a file in this directory, your app must silently move it out of the directory before making any changes. The contents of this directory are backed up by iTunes.

Directory	Description
<Application_Home>`/Library/`	This directory is the top-level directory for files that are not user data files. You typically put files in one of several standard subdirectories but you can also create custom subdirectories for files you want backed up but not exposed to the user. You should not use this directory for user data files. The contents of this directory (with the exception of the `Caches` subdirectory) are backed up by iTunes. For additional information about the Library directory, see "The Library Directory Stores App-Specific Files" (page 22).
<Application_Home>`/tmp/`	Use this directory to write temporary files that do not need to persist between launches of your app. Your app should remove files from this directory when it determines they are no longer needed. (The system may also purge lingering files from this directory when your app is not running.) In iOS 2.1 and later, the contents of this directory are not backed up by iTunes.

An iOS app may create additional directories in the `Documents`, `Library`, and `tmp` directories. You might want to do this to organize the files in those locations better.

For information about how to get references to the preceding directories from your iOS app, see "Locating Items in the Standard Directories" (page 34). For tips on where to put files, see "Where You Should Put Your App's Files" (page 14).

Where You Should Put Your App's Files

To prevent the syncing and backup processes on iOS devices from taking a long time, be selective about where you place files inside your app's home directory. Apps that store large files can slow down the process of backing up to iTunes or iCloud. These apps can also consume a large amount of a user's available storage, which may encourage the user to delete the app or disable backup of that app's data to iCloud. With this in mind, you should store app data according to the following guidelines:

- Put user data in the *<Application_Home>*`/Documents/`. User data is any data that cannot be recreated by your app, such as user documents and other user-generated content.
- Handle support files—files your application downloads or generates and can recreate as needed—in one of two ways:

- In iOS 5.0 and earlier, put support files in the *<Application_Home>*`/Library/Caches` directory to prevent them from being backed up
- In iOS 5.0.1 and later, put support files in the *<Application_Home>*`/Library/Application Support` directory and apply the `com.apple.MobileBackup` extended attribute to them. This attribute prevents the files from being backed up to iTunes or iCloud. If you have a large number of support files, you may store them in a custom subdirectory and apply the extended attribute to just the directory.

- Put data cache files in the *<Application_Home>*`/Library/Caches` directory. Examples of files you should put in this directory include (but are not limited to) database cache files and downloadable content, such as that used by magazine, newspaper, and map apps. Your app should be able to gracefully handle situations where cached data is deleted by the system to free up disk space.
- Put temporary data in the *<Application_Home>*`/tmp` directory. Temporary data comprises any data that you do not need to persist for an extended period of time. Remember to delete those files when you are done with them so that they do not continue to consume space on the user's device.

About the OS X File System

The OS X file system is designed for Macintosh computers, where both users and software have access to the file system. Users access the file system directly through the Finder, which presents a user-oriented view of the file system by hiding or renaming some files and directories. Apps access the file system using the system interfaces, which show the complete file system precisely as it appears on disk.

Domains Determine the Placement of Files

In OS X, the file system is divided into multiple domains, which separate files and resources based on their intended usage. This separation provides simplicity for the user, who only needs to worry about a specific subset of files. Arranging files by domain also lets the system apply blanket access privileges to files in that domain, preventing unauthorized users from changing files intentionally or inadvertently.

The **user domain** contains resources specific to the users who log in to the system. Although it technically encompasses all users, this domain reflects only the home directory of the current user at runtime. User home directories can reside on the computer's boot volume (in the `/Users` directory) or on a network volume. Each user (regardless of privileges) has access to and control over the files in his or her own home directory.

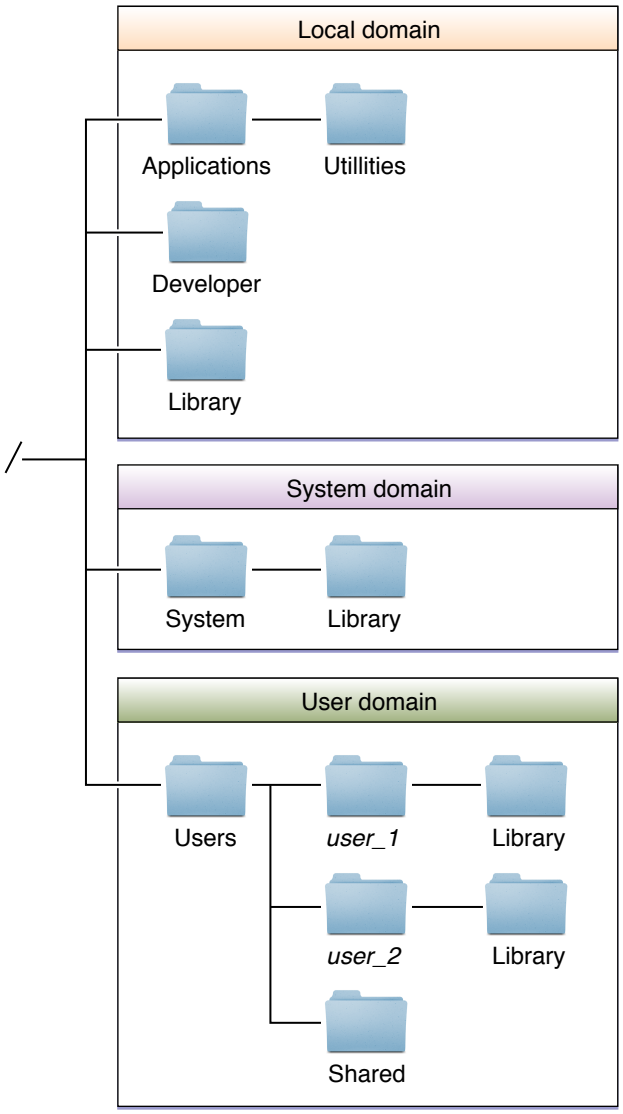
The **local domain** contains resources such as apps that are local to the current computer and shared among all users of that computer. The local domain does not correspond to a single physical directory, but instead consists of several directories on the local boot (and root) volume. This domain is typically managed by the system, but users with administrative privileges may add, remove, or modify items in this domain.

The **network domain** contains resources such as apps and documents that are shared among all users of a local area network. Items in this domain are typically located on network file servers and are under the control of a network administrator.

The **system domain** contains the system software installed by Apple. The resources in the system domain are required by the system to run. Users cannot add, remove, or alter items in this domain.

Figure 1-2 shows how the local, system, and user domains map to the local file system of an OS X installation. (The network domain is not shown but is similar in many ways to the local domain.) This figure shows the visible directories that the user might see. Depending on the user's system, other directories may be visible or some of the ones shown here may be hidden.

Figure 1-2 The local OS X file system



For information about the contents of the directories in OS X, see "OS X Standard Directories: Where Files Reside" (page 18). For information about the directories that OS X normally hides from the user (and why), see "Hidden Files and Directories: Simplifying the User Experience" (page 20).

OS X Standard Directories: Where Files Reside

Whether provided by the system or created by your app, every file has its place in OS X. Table 1-2 lists some of the top-level directories in an OS X installation and the types of content that each one contains.

Table 1-2 Commonly used directories in OS X

Directory	Usage
`/Applications`	This directory is where you install apps intended for use by all users of a computer. The App Store installs apps purchased by the user in this directory automatically. The `Utilities` subdirectory contains a subset of apps that are intended for use in managing the local system. This directory is part of the local domain.
`/Developer`	This directory contains the Xcode apps, tools, and developer-related resources. The default name of this directory is `Developer` but the user may change that name. A common reason to change the name is to support multiple installations of the Xcode tools. Regardless of the name, this directory is part of the local domain.
`Library`	There are multiple `Library` directories on the system, each one associated with a different domain or specific user. Apps should use the `Library` directory to store app-specific (or system-specific) resources. For detailed information about the contents of this directory and how you use it to support your apps, see "The Library Directory Stores App-Specific Files" (page 22).
`/Network`	This directory contains the list of computers in the local area network. There is no guarantee that files located on network file servers will have the `/Network` directory at the beginning of their path. Path names vary depending on several factors, including how the network volume was mounted. For example, if the user uses the Connect to Server command to mount a volume, paths begin with the `/Volumes` directory. When writing code, assume that files on any volume other than the boot volume could be located on a network-based server.
`/System`	This directory contains the system resources required by OS X to run. These resources are provided by Apple and must not be modified. This directory comprises the contents of the system domain.

Directory	Usage
`/Users`	This directory contains one or more user home directories. The user home directory is where user-related files are stored. A typical user's home directory includes the following subdirectories: `Applications`—Contains user-specific apps. `Desktop`—Contains the items on the user's desktop. `Documents`—Contains user documents and files. `Downloads`—Contains files downloaded from the Internet. `Library`—Contains user-specific app files (hidden in OS X 10.7 and later). `Movies`—Contains the user's video files. `Music`—Contains the user's music files. `Pictures`—Contains the user's photos. `Public`—Contains content the user wants to share. `Sites`—Contains web pages used by the user's personal site. (Web Sharing must be enabled to display these pages.) The preceding directories are for storing user documents and media only. Apps must not write files to the preceding directories unless explicitly directed to do so by the user. The sole exception to this rule is the `Library` directory, which apps may use to store data files needed to support the current user. Of the subdirectories, only the `Public` directory is accessible by other users on the system. Access to the other directories is restricted by default.

Important: The files in the user's `Documents` and `Desktop` directories should reflect only the documents that the user created and works with directly. Similarly, the media directories should contain only the user's media files. Those directories must never be used to store data files that your app creates and manages automatically. If you need a place to store automatically generated files, use the `Library` directory, which is designated specifically for that purpose. For information on where to put files in the Library directory, see "The Library Directory Stores App-Specific Files" (page 22).

Although the directories in Table 1-2 (page 18) are the ones seen by OS X users, they are not the only directories present in the file system. OS X hides many directories to prevent users from accessing files that they don't need to.

Sandboxed OS X App File Containers

OS X apps that are sandboxed have all their `Application Support`, `Cache`, temporary directories and other related documents stored within a directory located at a system-defined path that you can obtain by calling the `NSHomeDirectory` function.

For more information see *App Sandbox Design Guide*.

Hidden Files and Directories: Simplifying the User Experience

To simplify the experience for users, the Finder, and some specific user-facing interfaces (such as the Open and Save panels), hide many files and directories that the user should never have to use. Many of the hidden items are system- or app-specific resources that users cannot (or should not) access directly. Among the files and directories that are hidden are the following:

- **Dot directories and files**—Any file or directory whose name starts with a period (`.`) character is hidden automatically. This convention is taken from UNIX, which used it to hide system scripts and other special types of files and directories. Two special directories in this category are the `.` and `..` directories, which are references to the current and parent directories respectively.
- **UNIX-specific directories**—The directories in this category are inherited from traditional UNIX installations. They are an important part of the system's BSD layer but are more useful to software developers than end users. Some of the more important directories that are hidden include:
  - `/bin`—Contains essential command-line binaries. Typically, you execute these binaries from command-line scripts.
  - `/dev`—Contains essential device files, such as mount points for attached hardware.
  - `/etc`—Contains host-specific configuration files.
  - `/sbin`—Contains essential system binaries.
  - `/tmp`—Contains temporary files created by apps and the system.
  - `/usr`—Contains non-essential command-line binaries, libraries, header files, and other data.
  - `/var`—Contains log files and other files whose content is variable. (Log files are typically viewed using the Console app.)
- **Explicitly hidden files and directories**—The Finder may hide specific files or directories that should not be accessed directly by the user. The most notable example of this is the `/Volumes` directory, which contains a subdirectory for each mounted disk in the local file system from the command line. (The Finder provides a different user interface for accessing local disks.) In OS X 10.7 and later, the Finder also hides the `~/Library` directory—that is, the `Library` directory located in the user's home directory.

- **Packages and bundles**—Packages and bundles are directories that the Finder presents to the user as if they were files. Bundles hide the internal workings of executables such as apps and just present a single entity that can be moved around the file system easily. Similarly, packages allow apps to implement complex document formats consisting of multiple individual files while still presenting what appears to be a single document to the user.

Although the Finder and other system interfaces hide files and directories from the user, Cocoa interfaces such as `NSFileManager` do not filter out files or directories that are normally invisible to users. Thus, code that uses these interfaces theoretically has a complete view of the file system and its contents. (Of course, a process really has access to only those files and directories for which it has appropriate permissions.)

Files and Directories Can Have Alternate Names

In some situations, the Finder presents users with file or directory names that do not match the actual names as they appear in the file system. These names are known as **display names** and are used only by the Finder and specific system components (such as the Open and Save panels) when presenting file and directory information to the user. Display names improve the user experience by presenting the user with content in a more friendly way. For example, OS X uses display names in the following situations:

- **Localized names**—The system provides localized names for many system directories, such as `Applications`, `Library`, `Music`, `Movies`. An app may similarly provide localized names for itself and for any directories it creates.
- **Filename extension hiding**—The system hides filename extensions for all files by default. The user may change option, but when filename extension hiding is in effect, the characters after the last period in a filename (and the period itself) are not displayed.

Display names do not affect the actual name of the file in the file system. Code that accesses a file or directory programmatically must specify the item's actual name when opening or manipulating the item using the file system interfaces. The only time your app should ever use display names is when displaying the name of a file or directory to the user. You can get the display name for any file or directory using the `displayNameAtPath:` method of `NSFileManager`.

> **Important:** Your code should not allow users to modify display names directly. When you want the user to specify the name of a file, use a Save panel.

For information on how to localize the directories your app creates, see *File System Advanced Programming Topics*. For more information about localizing app content, see *Internationalization Programming Topics*.

The Library Directory Stores App-Specific Files

The `Library` directory is where apps and other code modules store their custom data files. Regardless of whether you are writing code for iOS or OS X, understanding the structure of the `Library` directory is important. You use this directory to store data files, caches, resources, preferences, and even user data in some specific situations.

There are several `Library` directories throughout the system but only a few that your code should ever need to access:

- `Library` in the current home directory—This is the version of the directory you use the most because it is the one that contains all user-specific files. In iOS, the home directory is the app's sandbox directory. In OS X, it is the app's sandbox directory or the current user's home directory (if the app is not in a sandbox).
- `/Library` (OS X only)—Apps that share resources between users store those resources in this version of the `Library` directory. Sandboxed apps are not permitted to use this directory.
- `/System/Library` (OS X only)—This directory is reserved for use by Apple.

After selecting which version of the Library directory to use, you still need to know where to store your files. The Library directory itself contains several subdirectories that subdivide app-specific content into a few well-known categories. Table 1-3 lists the most common subdirectories that you might use. Although Library directories in OS X contain many more subdirectories than the ones listed, most are used only by the system. If you want a more complete list of subdirectories, though, see "OS X Library Directory Details" (page 91).

Table 1-3 Key subdirectories of the `Library` directory

Directory	Usage
`Application Support`	Use this directory to store all app data files except those associated with the user's documents. For example, you might use this directory to store app-created data files, configuration files, templates, or other fixed or modifiable resources that are managed by the app. An app might use this directory to store a modifiable copy of resources contained initially in the app's bundle. A game might use this directory to store new levels purchased by the user and downloaded from a server. All content in this directory should be placed in a custom subdirectory whose name is that of your app's bundle identifier or your company. In iOS, the contents of this directory are backed up by iTunes.

Directory	Usage
`Caches`	Use this directory to write any app-specific support files that your app can re-create easily. Your app is generally responsible for managing the contents of this directory and for adding and deleting files as needed. In iOS 2.2 and later, the contents of this directory are not backed up by iTunes. In addition, iTunes removes files in this directory during a full restoration of the device. On iOS 5.0 and later, the system may delete the `Caches` directory on rare occasions when the system is very low on disk space. This will never occur while an app is running. However, you should be aware that iTunes restore is not necessarily the only condition under which the Caches directory can be erased.
`Frameworks`	In OS X, frameworks that must be shared by multiple apps can be installed in either the local or user domain. The Frameworks directory in the system domain stores the frameworks you use to create your OS X apps. In iOS, apps cannot install custom frameworks.
`Preferences`	This directory contains app-specific preference files. You should not create files in this directory yourself. Instead, use the `NSUserDefaults` class or CFPreferences API to get and set preference values for your app. In iOS, the contents of this directory are backed up by iTunes.

The iCloud File Storage Container

iCloud provides a structured system for storing files for apps that make use of iCloud:

- Apps have a primary iCloud container directory for storing their native files. They can also access secondary iCloud container directories listed in their app entitlements.
- Inside each container directory, files are segregated into "documents" and data. Every file or file package located in the `Documents` subdirectory (or one of its subdirectories) is presented to the user (via the iCloud UI in OS X and iOS) as a separate document that can be deleted individually. Anything not in `Documents` or one of its subdirectories is treated as data and shown as a single entry in the iCloud UI.

Documents that the user creates and sees in an app's user interface—for example the document browsers in Pages, Numbers, and Keynote should be stored in the `Documents` directory. Another example of files that might go in the `Documents` directory are saved games, again because they are something that an app could potentially provide some sort of method for selecting.

Anything that the app does not want the user to see or modify directly should be placed outside of the `Documents` directory. Apps can create any subdirectories inside the container directory, so they can arrange private files as desired.

Apps create files and directories in iCloud container directories in exactly the same way as they create local files and directories. And all the file's attributes are saved, if they add extended attributes to a file, those attributes are copied to iCloud and to the user's other devices too.

iCloud containers also allow the storage of key-value pairs that can be easily accessed without having to create a document format.

How the System Identifies the Type of Content in a File

There are two primary techniques for identifying the type of content in a file:

- Uniform Type Identifiers (UTIs)
- Filename extensions

A **uniform type identifier** is a string that uniquely identifies a class of entities considered to have a "type." UTIs provide consistent identifiers for data that all apps and services can recognize and rely upon. They are also more flexible than most other techniques because you can use them to represent any type of data, not just files and directories. Examples of UTIs include:

`public.text`—A public type that identifies text data.

`public.jpeg`—A public type that identifies JPEG image data.

`com.apple.bundle`—An Apple type that identifies a bundle directory.

`com.apple.application-bundle`—An Apple type that identifies a bundled app.

Whenever a UTI-based interface is available for specifying file types, you should prefer that interface over any others. Many OS X interfaces allow you to specify UTIs corresponding to the files or directories you want to work with. For example, in the Open panel, you can use UTIs as file filters and limit the types of files the user selects to ones your app can handle. Several AppKit classes, including `NSDocument`, `NSPasteboard`, and `NSImage`, support UTIs. In iOS, UTIs are used to specify pasteboard types only.

One way the system determines the UTI for a given file is by looking at its filename extension. A **filename extension** is a string of characters appended to the end of a file and separated from the main filename with a period. Each unique string of characters identifies a file of a specific type. For example, the `.strings` extension identifies a resource file with localizable string data while the `.png` extension identifies a file with image data in the portable network graphics format.

> **Note:** Because period characters are valid characters in OS X and iOS filenames, only the characters after the last period in a filename are considered part of the filename extension. Everything to the left of the last period is considered part of the filename itself.

If your app defines custom file formats, you should register those formats and any associated filename extensions in your app's `Info.plist` file. The `CFBundleDocumentTypes` key specifies the file formats that your app recognizes and is able to open. Entries for any custom file formats should include both a filename extension and UTI corresponding to the file contents. The system uses that information to direct files with the appropriate type to your app.

For more information about UTIs and how you use them, see *Uniform Type Identifiers Overview*. For more information about the `CFBundleDocumentTypes` key, see *Information Property List Key Reference*.

Security: Protect the Files You Create

Because all user data and system code are stored on disk somewhere, protecting the integrity of files and the file system is an important job. For that reason, there are several ways to secure content and prevent it from being stolen or damaged by other processes.

For general information about secure coding practices when working with files, see *Secure Coding Guide*.

Sandboxes Limit the Spread of Damage

In iOS and in OS X v10.7 and later, sandboxes prevent apps from writing to parts of the file system that they should not. Each sandboxed app receives its own container directory in which it can write files. An app cannot write to other apps' containers or to most directories outside of the sandbox. These restrictions limit the potential damage that can be done in the event that an app's security is breached.

Developers writing apps for OS X v10.7 and later are encouraged to put their apps in sandboxes to enhance security. Developers of iOS apps do not have to explicitly put their app in a sandbox because the system does it for them automatically at install time.

For more information about sandboxes and the types of restrictions they impose on file system access, see *Mac App Programming Guide* and *App Sandbox Design Guide*.

Permissions and Access Control Lists Govern All Access to Files

Access to files and directories is governed by a mixture of access control lists (ACLs) and BSD permissions. Access control lists are a set of fine-grained controls that define exactly what can and cannot be done to a file or directory and by whom. With access control lists, you can grant individual users different levels of access to

a given file or directory. By contrast, BSD permissions only allow you to give access to three classes of users: the file's owner, a single group of users that you specify, and all users. See *Security Overview* for more information.

> **Note:** For a file on a network server, do not make any assumptions about the ACLs and BSD permissions associated with the file. Some network file systems provide only a summarized version of this information.

Because iOS apps always run in a sandbox, the system assigns specific ACLs and permissions to files created by each app. However, OS X apps can use Identity Services to manage access control lists for files to which they have access. For information about how to use Identity Services (and the Collaboration framework), see *Identity Services Programming Guide*.

Files Can Be Encrypted On Disk

Both OS X and iOS provide support for encrypting files on disk:

- iOS—An iOS app can designate files that it wants to be encrypted on disk. When the user unlocks a device containing encrypted files, the system creates a decryption key that allows the app to access its encrypted files. When the user locks the device, though, the decryption key is destroyed to prevent unauthorized access to the files.
- OS X—Users can encrypt the contents of a volume using the Disk Utility app. (They can also encrypt just the boot volume from the Security & Privacy system preference.) The contents of an encrypted disk are available to apps only while the computer is running. When the user puts the computer to sleep or shuts it down, the decryption keys are destroyed to prevent unauthorized access to the disk's contents.

In iOS, apps that take advantage of disk-based encryption need to be discontinue the use of encrypted files when the user locks the device. Because locking the device destroys the decryption keys, access to encrypted files is limited to when the device is unlocked. If your iOS app can run in the background while the device is locked, it must do so without access to any of its encrypted files. Because encrypted disks in OS X are always accessible while the computer is running, OS X apps do not need to do anything special to handle disk-level encryption.

For more information about working with encrypted files in iOS, see *iOS App Programming Guide*.

Synchronization Ensures Robustness in Your File-Related Code

The file system is a resource shared by third-party apps and system apps. Because multiple apps are able to access files and directories at the same time, the potential arises for one app to make changes that render a second app's view of the file system obsolete. If the second app is not prepared to handle such changes, it could enter an unknown state or even crash. In cases where your app relies on the presence of specific files, you can use synchronization interfaces to be notified of changes to those files.

File system synchronization is primarily an issue in OS X, where the user can manipulate files directly with the Finder or with any number of other apps at the same time. Fortunately, OS X provides the following interfaces to help with synchronization issues:

- File coordinators—In OS X 10.7 and later, file coordinators are a way to incorporate fine-grained synchronization support directly into the objects of your app; see "The Role of File Coordinators and Presenters" (page 44).
- FSEvents—In OS X 10.5 and later, file system events allow you to monitor changes to a directory or its contents; see *File System Events Programming Guide*.

Files, Concurrency, and Thread Safety

Because file-related operations involve interacting with the hard disk and are therefore slow compared to most other operations, most of the file-related interfaces in iOS and OS X are designed with concurrency in mind. Several technologies incorporate asynchronous operation into their design and most others can execute safely from a dispatch queue or secondary thread. Table 1-4 lists some of the key technologies discussed in this document and whether they are safe to use from specific threads or any thread. For specific information about the capabilities of any interface, see the reference documentation for that interface.

Table 1-4 Thread safety of key classes and technologies

Class/Technology	Notes
`NSFileManager`	For most tasks, it is safe to use the default `NSFileManager` object simultaneously from multiple background threads. The only exception to this rule is tasks that interact with the file manager's delegate. When using a file manager object with a delegate, it is recommended that you create a unique instance of the `NSFileManager` class and use your delegate with that instance. You should then use your unique instance from one thread at a time.
Grand Central Dispatch	GCD itself is safe to use from any thread. However, you are still responsible for writing your blocks in a way that is thread safe.

Class/Technology	Notes
`NSFileHandle`, `NSData`, Cocoa streams	Most of the Foundation objects you use to read and write file data can be used from any single thread but should not be used from multiple threads simultaneously.
Open and Save panels	Because they are part of your user interface, you should always present and manipulate the Open and Save panels from your app's main thread.
POSIX routines	The POSIX routines for manipulating files are generally designed to operate safely from any thread. For details, see the corresponding man pages.
`NSURL` and `NSString`	The immutable objects you use to specify paths are safe to use from any thread. Because they are immutable, you can also refer to them from multiple threads simultaneously. Of course, the mutable versions of these objects should be used from only one thread at a time.
`NSEnumerator` and its subclasses	Enumerator objects are safe to use from any single thread but should not be used from multiple threads simultaneously.

Even if you use an thread-safe interface for manipulating a file, problems can still arise when multiple threads or multiple processes attempt to act on the same file. Although there are safeguards to prevent multiple clients from modifying a file at the same time, those safeguards do not always guarantee exclusive access to the file at all times. (Nor should you attempt to prevent other processes from accessing shared files.) To make sure your code knows about changes made to shared files, use file coordinators to manage access to those files. For more information about file coordinators, see "The Role of File Coordinators and Presenters" (page 44)

Accessing Files and Directories

Before you can open a file, you first have to locate it in the file system. The system frameworks provide many routines for obtaining references to many well-known directories, such as the `Library` directory and its contents. You can also specify locations manually by building a URL or string-based path from known directory names.

When you know the location of a file, you can then start planning the best way to access it. Depending on the type of file, you may have several options. For known file types, you typically use built-in system routines to read or write the file contents and give you an object that you can use. For custom file types, you may need to read the raw file data yourself.

Choose the Right Way to Access Files

Although you can open any file and read its contents as a stream of bytes, doing so is not always the right choice. OS X and iOS provide built-in support that makes opening many types of standard file formats (such as text files, images, sounds, and property lists) much easier. For these standard file formats, you should use the higher-level options for reading and writing the file contents. Table 2-1 lists the common file types supported by the system along with information about how you access them.

Table 2-1 File types with specialized routines

File Type	Examples	Description
Resource files	Nib files Image files Sound files Strings files Localized resources	Apps use resource files to store data that is independent of the code that uses it. Resource files are commonly used to store localizable content such as strings and images. The process for reading data from a resource file depends on the resource type. For information about how to read the contents of resource files, see *Resource Programming Guide*.
Text files	Plain text file UTF-8 formatted file UTF-16 formatted file	A text file is an unstructured sequence of ASCII or Unicode characters. You typically load the contents of a text file into an `NSString` object but may also read and write a text file as a raw stream of characters. For information about using the `NSString` class to load text from a file, see *String Programming Guide*.

File Type	Examples	Description
Structured data files	XML file Property list file Preference file	A structured data file usually consists of string-based data arranged using a set of special characters. For information about parsing XML, see *Event-Driven XML Programming Guide*.
Archive files	Binary files created using a keyed archiver object	An archive is a file format used to store a persistent version of your app's runtime objects. An archiver object encodes the state of the objects into a stream of bytes that can be written to disk all at once. An unarchiver reverses the process, using the stream of bytes to re-create the objects and restore them to their previous state. Archives are often a convenient alternative to implementing custom binary file formats for your documents or other data files. For information on how to create and read archive files, see *Archives and Serializations Programming Guide*.
File packages	Custom document formats	A file package is a directory that contains any number of custom data files but which is presented to the user as if it were a single file. Apps can use file packages to implement complex file formats that contain multiple distinct files, or a mixture of different types of files. For example, you might use a file package if your file format includes both a binary data file and one or more image, video, or audio files. You access the contents of a file package using `NSFileWrapper` objects.
Bundles	Apps Plug-ins Frameworks	Bundles provide a structured environment for storing code and the resources used by that code. Most of the time, you do not work with the bundle itself but with its contents. However, you can locate bundles and obtain information about them as needed. For information about bundles and how you access them, see *Bundle Programming Guide*
Code files	Binary code resources Dynamic shared libraries	Apps that work with plug-ins and shared libraries need to be able to load the associated code for that item to take advantage of its functionality. For information about how to load code resources into memory, see *Code Loading Programming Topics*.

File Type	Examples	Description
A file wrapper	A collection of files that appear as a single file	Apps use file wrappers to store files in a manner that can be written in a serialized manner that can use used with the pasteboard or stored as part of your data record. See "Using FileWrappers as File Containers" (page 84) for more information on file wrappers.

In situations where the standard file formats are insufficient, you can always create your own custom file formats. When reading and writing the content of custom files, you read and write data as a stream of bytes and apply those bytes to your app's file-related data structures. You have complete control over how you read and write the bytes and how you manage your file-related data structures. For more information about the techniques for reading and writing files that use custom file formats, see "Techniques for Reading and Writing Files Without File Coordinators" (page 73).

Specifying the Path to a File or Directory

The preferred way to specify the location of a file or directory is to use the `NSURL` class. Although the `NSString` class has many methods related to path creation, URLs offer a more robust way to locate files and directories. For apps that also work with network resources, URLs also mean that you can use one type of object to manage items located on a local file system or on a network server.

> **Note:** In addition to `NSURL`, you can also use the `CFURLRef` opaque type to manipulate paths as URLs. The `NSURL` class is toll-free bridged with the `CFURLRef` type, which means you can use them interchangeably in your code. For more information about how to create and manipulate URLs using Core Foundation, see *CFURL Reference*.

For most URLs, you build the URL by concatenating directory and file names together using the appropriate `NSURL` methods until you have the path to the item. A URL built in that way is referred to as a **path-based URL** because it stores the names needed to traverse the directory hierarchy to locate the item. (You also build string-based paths by concatenating directory and file-names together, with the results stored in a slightly different format than that used by the `NSURL` class.) In addition to path-based URLs, you can also create a **file reference URL**, which identifies the location of the or directory using a unique ID.

All of the following entries are valid references to a file called `MyFile.txt` in a user's `Documents` directory:

Path-based URL: `file://localhost/Users/steve/Documents/MyFile.txt`

File reference URL: `file:///.file/id=6571367.2773272/`

String-based path: `/Users/steve/Documents/MyFile.txt`

Different file systems rely on different separator characters. Because of these changes, you should create your URL objects using the methods provided by the NSURL class.

You create URL objects the `NSURL` methods and convert them to file reference URLs only when needed. Path-based URLs are easier to manipulate, easier to debug, and are generally preferred by classes such as `NSFileManager`. An advantage of file reference URLs is that they are less fragile than path-based URLs while your app is running. If the user moves a file in the Finder, any path-based URLs that refer to the file immediately become invalid and must be updated to the new path. However, as long as the file moved to another location on the same disk, its unique ID does not change and any file reference URLs remain valid.

> **Important:** Although they are safe to use while your app is running, file reference URLs are not safe to store and reuse between launches of your app because a file's ID may change if the system is rebooted. If you want to store the location of a file persistently between launches of your app, create a bookmark as described in "Locating Files Using Bookmarks."

Of course, there are still times when you might need to use strings to refer to a file. Fortunately, the `NSURL` class provides methods to convert path-based URLs to and from `NSString` objects. You might use a string-based path when presenting that path to the user or when calling a system routine that accepts strings instead of URLs. The conversion between `NSURL` objects and `NSString` objects is done using the NSURL class's method `absoluteString`.

Because `NSURL` and `NSString` describe only the location of a file or directory, you can create them before the actual file or directory exists. Neither class attempts to validate the actual existence of the file or directory you specify. In fact, you must create the path to a nonexistent file or directory before you can create it on disk.

For more information about the methods you use to create and manipulate URLs and strings, see *NSURL Class Reference* and *NSString Class Reference*.

Locating Items in the File System

Before you can access a file or directory, you need to know its location. There are several ways to locate files and directories:

- Find the file yourself.
- Ask the user to specify a location.
- Locating files in the standard system directories, in both sandboxed and non-sandboxed apps.
- Using bookmarks.

The file systems of iOS and OS X impose specific guidelines on where you should place files, so most of the items your app creates or uses should be stored in a well-known place. Both systems provide interfaces for locating items in such well-known places, and your app can use these interfaces to locate items and build paths to specific files. An app should prompt the user to specify the location of a file or directory only in a limited number of situations that are described in "Using the Open and Save Panels" (page 58).

Asking the User to Locate an Item

In OS X, user interactions with the file system should always be through the standard Open and Save panels. Because these panels involve interrupting the user, you should use them only in a limited number of situations:

- To open user documents
- To ask the user where to save a new document
- To associate user files (or directories of files) with an open window

You should never use the Open and Save panels to access any files that your app created and uses internally. Support files, caches, and app-generated data files should be placed in one of the standard directories dedicated to app-specific files.

For information on how to present and customize the Open and Save panels, see "Using the Open and Save Panels" (page 58).

Locating Items in Your App Bundle

Apps that need to locate resource files inside their bundle directory (or inside another known bundle) must do so using an `NSBundle` object. Bundles eliminate the need for your app to remember the location of individual files by organizing those files in a specific way. The methods of the `NSBundle` class understand that organization and use it to locate your app's resources on demand. The advantage of this technique is that you can generally rearrange the contents of your bundle without rewriting the code you use to access it. Bundles also take advantage of the current user's language settings to locate an appropriately localized version of a resource file.

The following code shows how to retrieve a URL object for an image named `MyImage.png` that is located in the app's main bundle. This code determines only the location of the file; it does not open the file. You would pass the returned URL to a method of the `NSImage` class to load the image from disk so that you could use it.

```
NSURL* url = [[NSBundle mainBundle] URLForResource:@"MyImage" withExtension:@"png"];
```

For more information about bundles, including how to locate items in a bundle, see *Bundle Programming Guide*. For specific information about loading and using resources in your app, see *Resource Programming Guide*.

Locating Items in the Standard Directories

When you need to locate a file in one of the standard directories, use the system frameworks to locate the directory first and then use the resulting URL to build a path to the file. The Foundation framework includes several options for locating the standard system directories. By using these methods, the paths will be correct whether your app is sandboxed or not:

- The `URLsForDirectory:inDomains:` method of the `NSFileManager` class returns a directory's location packaged in an `NSURL` object. The directory to search for is an `NSSearchPathDirectory` constant. These constants provide URLs for the user's home directory, as well as most of the standard directories.
- The `NSSearchPathForDirectoriesInDomains` function behaves like the `URLsForDirectory:inDomains:` method but returns the directory's location as a string-based path. You should use the `URLsForDirectory:inDomains:` method instead.
- The `NSHomeDirectory` function returns the path to either the user's or app's home directory. (Which home directory is returned depends on the platform and whether the app is in a sandbox.) When an app is sandboxed the home directory points to the app's sandbox, otherwise it points to the User's home directory on the file system. If constructing a file to a subdirectory of a user's home directory, you should instead consider using the `URLsForDirectory:inDomains:` method instead.

You can use the URL or path-based string you receive from the preceding routines to build new objects with the locations of the files you want. Both the `NSURL` and `NSString` classes provide path-related methods for adding and removing path components and making changes to the path in general. Listing 2-1 shows an example that searches for the standard `Application Support` directory and creates a new URL for a directory containing the app's data files.

Listing 2-1 Creating a URL for an item in the app support directory

```
- (NSURL*)applicationDataDirectory {
    NSFileManager* sharedFM = [NSFileManager defaultManager];
   NSArray* possibleURLs = [sharedFM URLsForDirectory:NSApplicationSupportDirectory
                                    inDomains:NSUserDomainMask];
    NSURL* appSupportDir = nil;
    NSURL* appDirectory = nil;
```

```
    if ([possibleURLs count] >= 1) {
        // Use the first directory (if multiple are returned)
        appSupportDir = [possibleURLs objectAtIndex:0];
    }

    // If a valid app support directory exists, add the
    // app's bundle ID to it to specify the final directory.
    if (appSupportDir) {
        NSString* appBundleID = [[NSBundle mainBundle] bundleIdentifier];
        appDirectory = [appSupportDir URLByAppendingPathComponent:appBundleID];
    }

    return appDirectory;
}
```

Locating Files Using Bookmarks

If you want to save the location of a file persistently, use the bookmark capabilities of `NSURL`. A **bookmark** is an opaque data structure, enclosed in an `NSData` object, that describes the location of a file. Whereas path- and file reference URLs are potentially fragile between launches of your app, a bookmark can usually be used to re-create a URL to a file even in cases where the file was moved or renamed.

To create a bookmark for an existing URL, use the `bookmarkDataWithOptions:includingResourceValuesForKeys:relativeToURL:error:` method of `NSURL`. Specifying the `NSURLBookmarkCreationSuitableForBookmarkFile` option creates an `NSData` object suitable for saving to disk. Listing 2-2 shows a simple example implementation that uses this method to create a bookmark data object.

Listing 2-2 Converting a URL to a persistent form

```
- (NSData*)bookmarkForURL:(NSURL*)url {
    NSError* theError = nil;
    NSData* bookmark = [url
bookmarkDataWithOptions:NSURLBookmarkCreationSuitableForBookmarkFile
                                          includingResourceValuesForKeys:nil
                                          relativeToURL:nil
                                          error:&theError];
    if (theError || (bookmark == nil)) {
```

```
        // Handle any errors.
        return nil;
    }
    return bookmark;
}
```

If you write the persistent bookmark data to disk using the `writeBookmarkData:toURL:options:error:` method of `NSURL`, what the system creates on disk is an alias file. Aliases are similar to symbolic links but are implemented differently. Normally, users create aliases from the Finder when they want to create links to files elsewhere on the system.

To transform a bookmark data object back into a URL, use the `URLByResolvingBookmarkData:options:relativeToURL:bookmarkDataIsStale:error:` method of `NSURL`. Listing 2-3 shows the process for converting a bookmark back into a URL.

Listing 2-3 Returning a persistent bookmark to its URL form

```
- (NSURL*)urlForBookmark:(NSData*)bookmark {
    BOOL bookmarkIsStale = NO;
    NSError* theError = nil;
    NSURL* bookmarkURL = [NSURL URLByResolvingBookmarkData:bookmark
                                        options:NSURLBookmarkResolutionWithoutUI
                                           relativeToURL:nil
                                           bookmarkDataIsStale:&bookmarkIsStale
                                           error:&theError];

    if (bookmarkIsStale || (theError != nil)) {
        // Handle any errors
        return nil;
    }
    return bookmarkURL;
}
```

The Core Foundation framework provides a set of C-based functions that parallel the bookmark interface provided by `NSURL`. For more information about using those functions, see *CFURL Reference*.

Enumerating the Contents of a Directory

When you want to discover what files are in a given directory, you can enumerate the contents of that directory. Cocoa supports enumerating a directory one file at a time or all at once. Regardless of which option you choose, you should enumerate directories sparingly because doing so can involve touching many files in the file system, which is expensive.

Enumerating a Directory One File at a Time

Enumerating a directory one file at a time is recommended when you want to search for a specific file and stop enumerating when you find it. File-by-file enumeration uses the `NSDirectoryEnumerator` class, which defines the methods for retrieving items. Because `NSDirectoryEnumerator` itself is an abstract class, however, you do not create instances of it directly. Instead, you use either the `enumeratorAtURL:includingPropertiesForKeys:options:errorHandler:` or `enumeratorAtPath:` method of `NSFileManager` object to obtain a concrete instance of the class for use in your enumeration.

Enumerator objects return the paths of all files and directories contained within the enumerated directory. Enumerations are recursive and cross device boundaries, so the number of files and directories returned may be more than what is present in the starting directory. You can skip the contents of any directory you are not interested in, by calling the enumerator object's `skipDescendents` method. Enumerator objects do not resolve symbolic links or attempt to traverse symbolic links that point to directories.

Listing 2-4 shows how to use the `enumeratorAtURL:includingPropertiesForKeys:options:errorHandler:` method to list all the user-visible subdirectories of a given directory, noting whether they are directories or file packages. The `keys` array tells the enumerator object to prefetch and cache information for each item. Prefetching this information improves efficiency by touching the disk only once. The options argument specifies that the 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.

Listing 2-4 Enumerating the contents of a directory

```
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 code in Listing 2-5 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.

Listing 2-5 Looking for files that have been modified recently

```
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);
        }
    }
}
```

Getting the Contents of a Directory in a Single Batch Operation

If you know that you want to look at every item in a directory, you can retrieve a snapshot of the items and iterate over them at your convenience. Retrieving the contents of a directory in a batch operation is not the most efficient way to enumerate a directory, because the file manager must walk the entire directory contents every time. However, if you plan to look at all the items anyway, it is a much simpler way to retrieve the items.

There are two options for doing a batch enumeration of a directory using `NSFileManager`:

- To perform a shallow enumeration, use the `contentsOfDirectoryAtURL:includingPropertiesForKeys:options:error:` or `contentsOfDirectoryAtPath:error:` method.
- To perform a deep enumeration and return only subdirectories, use the `subpathsOfDirectoryAtPath:error:` method.

Listing 2-6 shows an example that uses the `contentsOfDirectoryAtURL:includingPropertiesForKeys:options:error:` method to enumerate the contents of a directory. One of the benefits of using URLs is that you can also efficiently retrieve additional information about each item. This example retrieves the localized name, creation date, and type information for each item in the directory and stores that information in the corresponding `NSURL` object. When the method returns, you can proceed to iterate over the items in the `array` variable and do what you need with them.

Listing 2-6 Retrieving the list of items in a directory all at once

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

Determining Where to Store Your App-Specific Files

The `Library` directory is the designated repository for files your app creates and manages on behalf of the user. You should consider that these directories may be in different locations if your app is sandboxed. As a result, you should always use the `NSFileManager` method `URLsForDirectory:inDomains:` with the `NSUserDomainMask` as the domain to locate the specific directory that you should use to store this data.

- Use the `Application Support` directory constant `NSApplicationSupportDirectory`, appending your <*bundle_ID*> for:
  - Resource and data files that your app creates and manages for the user. You might use this directory to store app state information, computed or downloaded data, or even user created data that you manage on behalf of the user.
  - Autosave files.
- Use the `Caches` directory constant `NSCachesDirectory`, appending your <*bundle_ID*> directory for cached data files or any files that your app can re-create easily.
- Read and write preferences using the `NSUserDefaults` class. This class automatically writes preferences to the appropriate location.
- Use the `NSFileManager` method `URLsForDirectory:inDomains:` method to get the directory for storing temporary files. Temporary files are files you intend to use immediately for some ongoing operation but then plan to discard later. You should delete temporary files as soon as you are done with them. If you do not delete temporary files after three days, the system may delete them for you whether you are using them or not.

Note: The files that you store in these locations are rarely shared amongst apps. As such, they do not require the overhead of using creating and configuring an NSFileCoordination instance.

Tips for Accessing Files and Directories

Because the file system is a resource shared by all processes, including system processes, you should always use it carefully. Even on systems with solid-state disk drives, file operations tend to be a little slower because of the latency involved in retrieving data from the disk. And when you do access files, it is important that you do so in a way that is secure and does not interfere with other processes.

Perform File Operations Lazily

Operations involving the file system should be performed only when they are absolutely necessary. Accessing the file system usually takes a lot of time relative to other computer-wide tasks. So make sure you really need to access the disk before you do. Specifically:

- **Write data to disk only when you have something valuable to save.** The definition of what is valuable is different for each app but should generally involve information that the user provides explicitly. For example, if your app creates some default data structures at launch time, you should not save those structures to disk unless the user changes them.

- **Read data from disk only when you need it.** In other words, load data that you need for your user interface now but do not load an entire file just to get one small piece of data from that file. For custom file formats, use file mapping or read only the few chunks of a file that you need to present your user interface. Read any remaining chunks as needed in response to user interactions with the data. For structured data formats, use Core Data to manage and optimize the reading of data for you.

Use Secure Coding Practices

There are several principles you can follow to help ensure that you do not have file-based security vulnerabilities in your program:

- Check the result codes for any functions or methods you call. Result codes are there to let you know that there is a problem, so you should pay attention to them. For example, if you try to delete a directory that you think is empty and get an error code, you might find that the directory is not empty after all.
- When working in a directory to which your process does not have exclusive access, you must check to make sure a file does not exist before you create it. You must also verify that the file you intend to read from or write to is the same file you created.
- Whenever possible use routines that operate on file descriptors rather than pathnames. In this way you can be sure you're always dealing with the same file.
- Intentionally create files as a separate step from opening them so that you can verify that you are opening a file you created rather than one that already existed
- Know whether the function you are calling follows symbolic links. For example, the `lstat` function gives you the status of a file regardless of whether it's a normal file or a symbolic link, but the `stat` function follows symbolic links and, if the specified file was a symbolic link, returns the status of the linked-to file. Therefore, if you use the `stat` function, you might be accessing a different file than you expected.
- Before you read a file, make sure that file has the owner and permissions you expect. Be prepared to fail gracefully (rather than hanging) if it does not.
- Set your process' file code creation mask (umask) to restrict access to files created by your process. The umask is a bitmask that alters the default permissions of a new file. You do not reset the umask for your app, your process inherits the umask from its parent process. For more information about setting the umask, see `umask(2) OS X Developer Tools Manual Page`.

For additional tips and coding practices, see "Race Conditions and Secure File Operations" in *Secure Coding Guide*.

Assume Case Sensitivity for Paths

When searching or comparing filenames, always assume that the underlying file system is case sensitive. OS X supports many file systems that use case to differentiate between files. Even on file systems (such as HFS+) that support case insensitivity, there are still times when case may be used to compare filenames. For example, the `NSBundle` class and CFBundle APIs consider case when searching bundle directories for named resources.

Include Filename Extensions for New Files

All files should have a filename extension that reflects the type of content contained in the file. Filename extensions help the system determine how to open files and also make it easier to exchange files with other computers or other platforms. For example, network transfer programs often use filename extensions to determine the best way to to transfer files between computers.

Use Display Names When Presenting Items

Whenever you need to present the name of a file or directory in your user interface, always use the item's display name. Using the display name ensures that what your app presents matches what the Finder and other apps are presenting. For example, if your app shows the path to a file, using the display name ensures that the path is localized in accordance with the current user's language settings.

For more information about display names, see "Files and Directories Can Have Alternate Names" (page 21).

Accessing Files Safely From Background Threads

In general, the objects and functions used to access and manipulate files can be used from any thread of your app. However, as with all threaded programming, make sure you manipulate files safely from your background threads:

- Avoid using the same file descriptor from multiple threads. Doing so could cause each thread to disrupt the state of the other.
- Always use file coordinators to access files. File coordinators provide notifications when other threads in your program (or other processes) access the same file. You can use these notifications to clean up your local data structures or perform other relevant tasks.
- Create a unique instance of `NSFileManager` for each thread when copying, moving, linking, or deleting files. Although most file manager operations are thread-safe, operations involving a delegate require a unique instance of `NSFileManager`, especially if you are performing those operations on background threads.

The Role of File Coordinators and Presenters

Because the file system is shared by all running processes, problems can occur when two processes (or two threads in the same process) try to act on the same file at the same time. Two different processes acting on the same file might make changes that the other is not expecting, which could lead to more serious problems like crashes or data corruption. And even within a single program, two threads acting on the file simultaneously can corrupt it and render it unusable. To avoid this type of file contention, OS X 10.7 and later includes support for **file coordinators**, which allow you to coordinate file access safely between different processes or different threads.

The job of a file coordinator is to notify interested parties whenever a file they care about is acted on by another process or thread. The interested parties in this case are the objects of your app. Any object of your app can designate itself as a **file presenter**—that is, an object that works directly with a file and needs to be notified when actions are initiated on that file by other objects. Any object in your app can be a file presenter and a file presenter can monitor individual files or whole directories of files. For example, an app that works with images would designate one or more objects as file presenters for the corresponding image files. If the user deleted an image file from the Finder while the app was running, the file presenter for that image file would be notified and given an opportunity to accommodate the deletion of the file.

Using NSDocument and UIDocument to Handle File Coordinators and Presenters

The `UIDocument` class on iOS and the `NSDocument` class on OS X encapsulate the data for a user document. These classes automatically support:

- The `NSFileCoordinator` implementation.
- The `NSFilePresenter` implementation.
- The Autosave implementation.

Whenever possible you should explore using these classes to store your user data. Your app need only work directly with file coordinators and file presenters when dealing with files other than the user's data files.

Apps that use the document classes don't need to worry about using file coordinators when reading and writing private files in the `Application Support`, `Cache`, or temporary directories, as these should be considered private.

Ensuring Safe Read and Write Operations Using File Coordinators

When your app wants to modify a file, it must create a file coordinator object to notify any interested file presenters of its intentions. A file coordinator communicates with all of the relevant file presenters about your app's intentions toward the file. After the file presenters have all responded, the file coordinator executes a block containing the actual actions you want to perform. All of these takes place asynchronously so that your app threads do not have to wait explicitly.

The steps for adding support for file coordinators to your code is as follows:

1. For each object that manages a file or directory, make the corresponding class adopt the `NSFilePresenter` protocol. The system uses the methods of this protocol to notify your object when changes occur to the monitored file.
2. When you initialize your file presenter object at runtime, register it immediately by calling the `addFilePresenter:` class method of `NSFileCoordinator`.
3. When your file presenter object performs its own actions on a file or directory, create an `NSFileCoordinator` object and call the method that corresponds to the action you plan to take.
4. In the `dealloc` method of your file presenter object, call the `removeFilePresenter:` class method of `NSFileCoordinator` to unregister it as a file presenter.

Choosing Which Files to Monitor with File Presenters

Not every file touched by your app needs to be monitored. Apps should always use file presenters when monitoring the following types of items:

- User documents
- Directories containing media files (music, photos, movies, and so on) that your app manages
- Files located on remote servers

You generally do not need to monitor the files in your app bundle or any files that are private to your app and stored in app-specific subdirectories of the `~/Library` directory. If your app runs in a sandbox, you also do not need to monitor any files you create inside your sandbox directory.

Implementing Your File Presenter Objects

Most of your app's file presenter objects are going to be either controller objects or data model objects. Controller objects are a natural choice because they generally coordinate the creation and modification of other objects, including data model objects. However, if your app monitors hundreds or thousands of files, you might want to move the support for managing individual files down into your data model objects.

To implement a file presenter object, make the class for that object conform to the `NSFilePresenter` protocol. The methods in this protocol indicate the types of operations that can be performed on the file or directory. Your implementation of the various methods is where your object responds and takes any necessary actions. You do not have to implement every method of the protocol but should implement all methods that might affect how your object interacts with the file or directory. You use the properties and methods of the protocol to do the following:

- Provide the URL of the file or directory being monitored. (All file presenters must do this by implementing the `presentedItemURL` property.)
- Relinquish control of the file or directory temporarily so that another object can write to it or read from it.
- Save any unsaved changes before another object tries to read from the file.
- Track changes to the contents or attributes of a file or directory.
- Update your data structures when the file or directory is moved or deleted.
- Provide a dispatch queue on which to execute your file presenter methods.

If you are implementing a document-based app, you do not need to incorporate file presenter semantics into your `NSDocument` subclasses. The `NSDocument` class already conforms to the `NSFilePresenter` protocol and implements the appropriate methods. Thus, all of your documents automatically register themselves as presenters of their corresponding file and do things like save changes and track changes to the document.

All of your file presenter methods should be lightweight and execute quickly. In cases where your file presenter is responding to changes (such as in the `presentedItemDidChange`, `presentedItemDidMoveToURL:`, or `accommodatePresentedItemDeletionWithCompletionHandler:` methods), you might want to avoid incorporating changes directly from your file presenter method. Instead, dispatch a block asynchronously to a dispatch queue and process the changes at a later time. This lets you process the changes at your app's convenience without causing unnecessary delays to the file coordinator that initiated the change. Of course, when saving or relinquishing control of a file (such as in the `relinquishPresentedItemToReader:`, `relinquishPresentedItemToWriter:`, or `savePresentedItemChangesWithCompletionHandler:` methods) you should perform all necessary actions immediately and not defer them.

For details about how to implement the methods of the `NSFilePresenter` protocol, see *NSFilePresenter Protocol Reference*.

Registering File Presenters with the System

Every file presenter object created by your app must be registered with the system. Registering your objects tells the system that they are currently interested in a particular file and want to receive notifications about actions on it. You register a file presenter using the `addFilePresenter:` class method of `NSFileCoordinator`.

Any object you register as a file presenter must also be unregistered using the `removeFilePresenter:` method at appropriate times. Specifically, if you release a file presenter object in your code, you must unregister it before it is actually deallocated. Failure to do this is a programmer error.

Initiating File Changes with a File Coordinator

Before your file presenter object makes any changes to its monitored file, it must create an `NSFileCoordinator` object and indicate the type of change it is about to make. File coordinators are the mechanism by which file presenters system-wide are notified of changes. This object works with the system to notify other threads and processes and give the file presenters in those threads and processes a chance to respond. After all of the interested file presenters have responded, the file coordinator executes the block of code you provide to perform the actual actions. To use a file coordinator object, do the following:

1. Create an instance of the `NSFileCoordinator` class.
2. Call the instance method that corresponds to the action you want to take on the file or directory:
    - Call the `coordinateReadingItemAtURL:options:error:byAccessor:` method to read the contents of a file or directory.
    - Call the `coordinateWritingItemAtURL:options:error:byAccessor:` method to write to a file or to change the contents of a directory.
    - Call the `coordinateReadingItemAtURL:options:writingItemAtURL:options:error:byAccessor:` method to read and write from a file or directory.
    - Call the `itemAtURL:didMoveToURL:` method to move a file or directory to a new location.

    When you call these methods, you provide a block to do the actual reading or writing of the file or directory. If you need to read or write multiple items in a batch, you should call the `prepareForReadingItemsAtURLs:options:writingItemsAtURLs:options:error:byAccessor:` method instead and use the block passed to that method to coordinate the reading and writing of individual files. Using that method to process batches of files is much more efficient than trying to read or write files individually.
3. Release the file coordinator object as soon as you are done.

For more information about how to use the methods of `NSFileCoordinator`, see *NSFileCoordinator Class Reference*.

iCloud File Management

The iCloud storage APIs let your app write user documents and data to a central location and access those items from all of a user's computers and iOS devices. Making a user's documents ubiquitous using iCloud means that a user can view or edit those documents from any device without having to sync or transfer files explicitly. Storing documents in a user's iCloud account also provides a layer of security for that user. Even if a user loses a device, the documents on that device are not lost if they are in iCloud storage.

> **Important:** The iCloud APIs do not work with garbage collection in OS X. If your existing code uses garbage collection, you should update your code to use ARC instead.

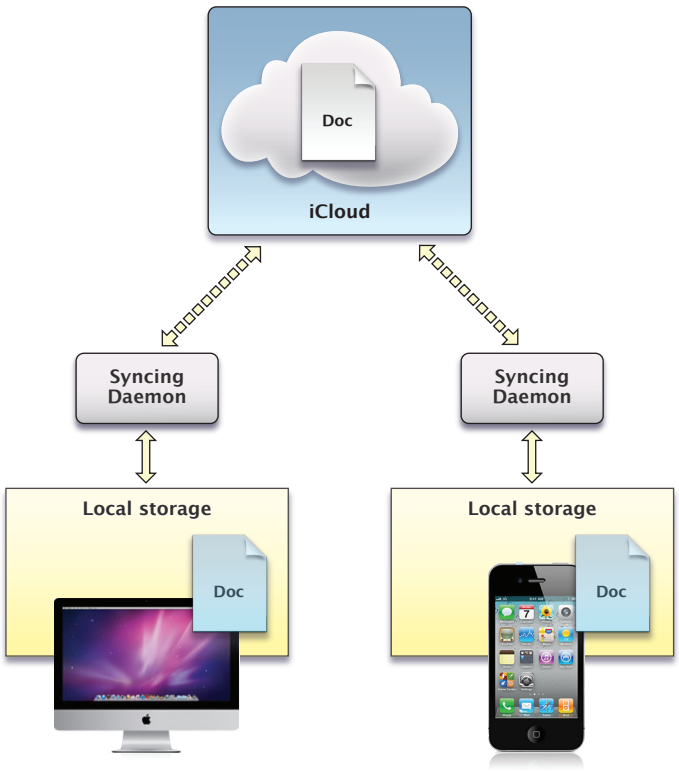
Storing and Using Documents in iCloud

Documents in iCloud provide the central location from which updates can be delivered to a user's computers and iOS devices. All documents must be created on a local disk initially and moved to a user's iCloud account later. A document targeted for iCloud storage is not moved to iCloud immediately, though. First, it is moved from its current location in the file system to a local system-managed directory where it can be monitored by the iCloud service. After that transfer, the file is transferred to iCloud and to the user's other devices as soon as possible.

While in iCloud storage, changes made on one device are stored locally and then pushed to iCloud using a local daemon, as shown in Figure 4-1. To prevent large numbers of conflicting changes from occurring at the same time, apps are expected to use file coordinator objects to perform all changes. File coordinators mediate

changes between your app and the daemon that facilitates the transfer of the document to and from iCloud. In this way, the file coordinator acts like a locking mechanism for the document, preventing your app and the daemon from modifying the document simultaneously.

Figure 4-1 Pushing document changes to iCloud



From an implementation perspective, the easiest way to manage documents in iCloud is to use the `NSDocument` class. This class does most of the heavy lifting required to read and write files that are stored in iCloud. Specifically, the `NSDocument` class handles the creation and use of file coordinators to modify the document. This class also seamlessly integrates document changes coming from other devices. The class even helps handle the potential conflicts that can arise when two devices do manage to update the same file in conflicting ways. You are not required to use the `NSDocument` class to manage your app's documents, but using it requires less effort on your part.

iCloud Storage APIs

Most apps will use iCloud document storage to share documents from a user's iCloud account. This is the feature that users think of when they think of iCloud storage. A user cares about whether documents are shared across devices and can see and manage those documents from a given device. In contrast, the iCloud key-value

data store is not something a user would see. It is a way for your app to share very small amounts of data (tens of kilobytes) with other instances of itself. Apps can use this feature to store important state information. A magazine app might save the issue and page that the user read last, while a stocks app might store the stock symbols the user is tracking.

The sections that follow provide more details about how to implement different aspects of iCloud storage for your app. For additional information about using specific classes and interfaces, see the corresponding reference documentation.

Working with Documents in iCloud

When your app needs to read or write a document in iCloud, it must do so in a coordinated manner. Your app might not be the only app trying to manipulate the local file at any given moment. The daemon that transfers the document to and from iCloud also needs to manipulate it periodically. To prevent your app from interfering with the daemon (and vice versa), the system provides a coordinated locking mechanism that works with the files and directories you target for iCloud storage.

At the heart of the iCloud locking mechanism are file coordinators and file presenters. Whenever you need to read and write a file, you do so using a **file coordinator**, which is an instance of the `NSFileCoordinator` class. The job of a file coordinator is to coordinate the reads and writes performed by your app and the sync daemon on the same document. For example, your app and the daemon may both read the document at the same time but only one may write to the file at any single time. Also, if one process is reading the document, the other process is prevented from writing to the document until the reader is finished.

In addition to coordinating operations, file coordinators also work with file presenters to notify apps when changes are about to occur. A **file presenter** is any object that conforms to the `NSFilePresenter` protocol and takes responsibility for managing a specific file (or directory of files) in an app. The job of a file presenter is to protect the integrity of its own data structures. It does this by listening for messages from other file coordinators and using those messages to update its internal data structures. In most cases, a file presenter may not have to do anything. However, if a file coordinator declares that it is about to move a file to a new URL, the file presenter would need to replace its old URL with the new one provided to it by the file coordinator. The `NSDocument` class is an example of a file presenter that tracks changes to its underlying file or file package.

Here is a checklist of the things your app must do to work with documents in iCloud:

- Manage each document in iCloud using a file presenter. The recommended way to do this is to use the `NSDocument` class, but you may define custom file presenters if you prefer. You can use a single file presenter to manage a file package or a directory of files.
- After creating a file presenter, register it by calling the `addFilePresenter:` class method of `NSFileCoordinator`. Registration is essential. The system can notify only registered presenter objects.
- Before deleting a file presenter, unregister it by calling the `removeFilePresenter:` method of `NSFileCoordinator`.

All file-related operations must be performed through a file coordinator object. To read or write a document, or move or delete it, follow these steps:

1. Create an instance of the `NSFileCoordinator` class and initialize it with the file presenter object that is about to perform the file operation.
2. Use the methods of the `NSFileCoordinator` object to read or write the file:
    - To read all or part of a single file, use the `coordinateReadingItemAtURL:options:error:byAccessor:` method
    - To write to a file or delete it, call the `coordinateWritingItemAtURL:options:error:byAccessor:` method.
    - To perform a sequence of read or write operations, use the `coordinateReadingItemAtURL:options:writingItemAtURL:options:error:byAccessor:` method.
    - To write to multiple files, or to move a file to a new location, use the `coordinateWritingItemAtURL:options:writingItemAtURL:options:error:byAccessor:` method.

    You perform the actual file-related operations in the block that you pass to these methods. You should perform operations as quickly as possible to avoid blocking other apps that might be trying to access the file at the same time.

3. When you are done with the operations, release the file coordinator object.

For more information about using file coordinators and file presenters, see "The Role of File Coordinators and Presenters" (page 44).

Warning: When working with iCloud files and directories, use the alphanumeric character set as much as possible and avoid special punctuation or other special characters. You should assume that filenames are case sensitive and should not change the letter case of any file. Always access the files using the same letter case as the original file. Keeping your filenames simple helps ensure that those files can be handled correctly on different types of file systems.

Moving a Document to iCloud Storage

To move a document to iCloud storage:

1. Create and save the file in an appropriate local directory.
2. If you are not using the `NSDocument` class to manage your file, create a file presenter to be responsible for it. For information on file presenters, see "Working with Documents in iCloud."
3. Create an `NSURL` object that specifies the destination of the file in a user's iCloud storage.

   You must put files in one of the container directories associated with your app. Call the `URLForUbiquityContainerIdentifier:` method of `NSFileManager` to get the root URL for the directory, and then append any additional directory and filenames to that URL. (Apps may put documents in any container directory for which they have the appropriate entitlement.)
4. Call the `setUbiquitous:itemAtURL:destinationURL:error:` method of `NSFileManager` to move the file to the specified destination in iCloud.

When moving documents to iCloud, you can create additional subdirectories inside the container directory to manage your files. It is strongly recommended that you create a `Documents` subdirectory and use that directory for storing user documents. In iCloud, the contents of the `Documents` directory are made visible to the user so that individual documents can be deleted. Everything outside of the `Documents` directory is grouped together and treated as a single entity that a user can keep or delete. You create subdirectories in a user's iCloud storage using the methods of the `NSFileManager` class just as you would any directory.

After you move a document to iCloud, it is not necessary to save a URL to the document's location persistently. If you manage a document using a `NSDocument` object, that object automatically updates its local data structures with the document's new URL. However, it does not save that URL to disk, and neither should your custom file presenters. Instead, because documents can move while in a user's iCloud storage, you should use

an `NSMetadataQuery` object to search for documents. Searching guarantees that your app has the correct URL for accessing the document. For information on how to search for documents in iCloud, see "iCloud Storage APIs."

For more information about the methods of the `NSFileManager` class, see *NSFileManager Class Reference*.

Searching for Documents in iCloud

To locate documents in iCloud storage, your app must search using an `NSMetadataQuery` object. Searching is a guaranteed way to locate documents in a user's iCloud storage. You should always use query objects instead of saving URLs persistently because the user can delete files from iCloud storage when your app is not running. Using a query to search is the only way to ensure an accurate list of documents.

The `NSMetadataQuery` class supports the following search scopes for your documents:

- Use the `NSMetadataQueryUbiquitousDocumentsScope` constant to search for documents in iCloud that reside somewhere inside a `Documents` directory. (For any given container directory, put documents that the user is allowed to access inside a `Documents` subdirectory.)
- Use the `NSMetadataQueryUbiquitousDataScope` constant to search for documents in iCloud that reside anywhere other than in a `Documents` directory. (For any given container directory, use this scope to store user-related data files that your app needs to share but that are not files you want the user to manipulate directly.)

To use a metadata query object to search for documents, create a new `NSMetadataQuery` object and do the following:

1. Set the search scope of the query to an appropriate value (or values). It's important to note that it is not possible to combine local file system searches with iCloud searches. The search must be run separately.
2. Add a predicate to narrow the search results. For example, to search for all files, specify a predicate with the format `NSMetadataItemFSNameKey == *`.
3. Register for the query notifications and configure any other query parameters you care about, such as sort descriptors, notification intervals, and so on.

   The `NSMetadataQuery` object uses notifications to deliver query results. At a minimum, you should register for the `NSMetadataQueryDidUpdateNotification` notification, but you might want to register for others in order to detect the beginning and end of the results-gathering process.
4. Call the `startQuery` method of the query object.
5. Run the current run loop so that the query object can generate the results.

If you started the query on your app's main thread, simply return and let the main thread continue processing events. If you started the query on a secondary thread, you must configure and execute a run loop explicitly. For more information about executing run loops, see *Threading Programming Guide*.

6. Process the results in your notification-handler methods.

   When processing results, always disable updates first. Doing so prevents the query object from modifying the result list while you are using it. When you are done processing the results, reenable updates again to allow new updates to arrive.

7. When you are ready to stop the search, call the `stopQuery` method of the query object.

For more information on how to create and run metadata queries, see *File Metadata Search Programming Guide*.

Handling File-Version Conflicts

When multiple instances of your app (running on different computers or iOS devices) try to modify the same document in iCloud, a conflict can occur. For example, if two iOS devices are not connected to the network and the user makes changes on both, both devices try to push their changes to iCloud when they are reconnected to the network. At this point, iCloud has two different versions of the same file and has to decide what to do with them. Its solution is to make the most recently modified file the **current file** and to mark any other versions of the file as **conflict versions**.

Your app is notified of conflict versions through its file presenter objects. It is the job of the file presenter to decide how best to resolve any conflicts that arise. Apps are encouraged to resolve conflicts quietly whenever possible, either by merging the file contents or by discarding the older version if the older data is no longer relevant. However, if discarding or merging the file contents is impractical or might result in data loss, your app might need to prompt the user for help in choosing an appropriate course of action. For example, you might let the user choose which version of the file to keep, or you might offer to save the older version under a new name.

Apps should always attempt to resolve conflict versions as soon as possible. When conflict versions exist, all of the versions remain in a user's iCloud storage (and locally on any computers and iOS devices) until your app resolves them. The current version of the file and any conflict versions are reported to your app using instances of the `NSFileVersion` class.

To resolve conflicts:

1. Get the current file version using the `currentVersionOfItemAtURL:` class method of `NSFileVersion`.
2. Get an array of conflict versions using the `unresolvedConflictVersionsOfItemAtURL:` class method of `NSFileVersion`.

3. For each conflict version object, perform whatever actions are needed to resolve the conflict. Options include:
    - Merging the conflict versions with the current file automatically, if it is practical to do so.
    - Ignoring the conflict versions, if doing so does not result in any data loss.
    - Prompting the user to select which version (current or conflict) to keep. This should always be your last option.
4. Update the current file as needed.
    - If the current file version remains the winner, you do not need to update the current file.
    - If a conflict version is chosen as the winner, use a coordinated write operation to overwrite the contents of the current file with the contents of the conflict version.
    - If the user chooses to save the conflict version under a different name, create the new file with the contents of the conflict version.
5. Set the `resolved` property of the conflict version objects to `YES`.

    Setting this property to `YES` causes the conflict version objects (and their corresponding files) to be removed from the user's iCloud storage.

For more information about file versions and how you use them, see *NSFileVersion Class Reference*.

Using iCloud Storage Responsibly

Apps that take advantage of iCloud storage features should act responsibly when storing data in there. The space available in each user's account is limited and is shared by all apps. In addition, users can see how much space is consumed by a given app and choose to delete documents and data associated with your app. For these reasons, it is in your app's interest to be responsible about what files you store. Here are some tips to help you manage documents appropriately:

- Rather than storing all documents, let a user choose which documents to store in an iCloud account. If a user creates a large number of documents, storing all of those documents in iCloud could overwhelm that user's available space. Providing a way for a user to designate which documents to store in iCloud gives that user more flexibility in deciding how best to use the available space.
- Remember that deleting a document removes it from a user's iCloud account and from all of that user's computers and devices. Make sure that users are aware of this fact and confirm any delete operations. For your app to refresh the local copy of a document, use the `evictUbiquitousItemAtURL:error:` method of `NSFileManager`.

- When storing documents in iCloud, place them in a `Documents` directory whenever possible. Documents inside a `Documents` directory can be deleted individually by the user to free up space. However, everything outside that directory is treated as data and must be deleted all at once.
- Never store caches or other files that are private to your app in a user's iCloud storage. A user's iCloud account should be used only for storing user data and content that cannot be re-created by your app.

Using the Open and Save Panels

When working with user documents and files in OS X, the user should decide where those files reside in the file system. The standard Open and Save panels provide you with an interface to use whenever you interact with the user's files. You present the Open panel when you want the user to select one or more existing files or directories. Present the Save panel when you have a new user document that you need to write to disk.

> **Important:** An iOS app should never use an open or save panel to prompt the user for the location of a file. Apps should always save files to known locations in the app sandbox and manage the selection of any user documents using a custom interface.

The Open Panel: Getting Existing Files or Directories

You use the `NSOpenPanel` class when you want to prompt the user to select one or more files or directories. Exactly how you use this class depends on what you plan to do with the selected items:

- To open a document in a new window, present the Open panel modally relative to the app.
- To choose files or directories and use them in conjunction with an already open document or window, present the panel as a sheet attached to the corresponding window.

Opening User Documents That You Plan to Display in a New Window

When you want to ask the user to select a document to display in a new window, display the standard Open panel modally. Document-based apps normally provide this behavior for you automatically, providing the infrastructure needed to respond to the Open command in the menu. To display this panel in other situations, you can implement your own method to display the panel. The steps for creating and presenting an open panel yourself are as follows:

1. Use the `openPanel` class method of `NSOpenPanel` to retrieve an open panel object.
2. Present the panel using the `beginWithCompletionHandler:` method.
3. Use your completion handler to process the results.

Listing 5-1 shows a custom method that presents the standard open panel to the user. This panel uses the default configuration options, which support the selection of a single file from all available file types. The `beginWithCompletionHandler:` method displays the panel in a detached window and returns immediately. The panel runs modally relative to the app and calls the supplied completion handler on the app's main thread when an item is selected. Upon the successful selection of a document, you would need to provide the code to open the document and present its window.

Listing 5-1 Presenting the open panel to the user

```
- (IBAction)openExistingDocument:(id)sender {
   NSOpenPanel* panel = [NSOpenPanel openPanel];

   // This method displays the panel and returns immediately.
   // The completion handler is called when the user selects an
   // item or cancels the panel.
   [panel beginWithCompletionHandler:^(NSInteger result){
      if (result == NSFileHandlingPanelOKButton) {
         NSURL*  theDoc = [[panel URLs] objectAtIndex:0];

         // Open  the document.
      }

   }];
}
```

Important: In OS X 10.6 and earlier, you must retain an open panel prior to displaying it and release it when you are done with it. Because the `openPanel` method returns an autoreleased object, the panel is normally released shortly after it appears on screen. Retaining the panel prevents it from being deallocated and dismissed prematurely. You do not need to retain the panel if it is attached to a window and you do not need to retain the panel in OS X 10.7 and when using ARC.

In a document-based app, requests to display the Open panel are handled by the `openDocument:` method of the app's document controller object, which is part of the app's default responder chain. Instead of implementing your own Open panel code, you might consider calling the `openDocument:` method instead. Doing so ensures that your custom code and the default app infrastructure always display the same Open panel. Customizing the panel at a later time also becomes easier because you have to make changes in only one place.

For more information about implementing a document-based app, see *Mac App Programming Guide*.

Choosing Files and Directories for an Already Open Window

To choose files or directories that relate to the current window, configure the standard Open panel as a Choose panel and use it to retrieve the user's selection. The main difference between the standard Open panel and a Choose panel is how you present it. Whereas you present the standard Open panel as a standalone modal panel, you almost always attach a Choose panel to one of your existing windows. Attaching the panel to a window makes it modal to the window but not to your whole app. Other differences relate to the configuration of the panel itself. A Choose panel can be configured to allow the selection of directories, multiple items, or hidden items among other options.

The `NSOpenPanel` object returned by the `openPanel` method is configured with the following default options:

- File selection: enabled
- Directory selection: disabled
- Resolve aliases: enabled
- Multiple selection: disabled
- Show hidden files: disabled
- File packages treated as directories: disabled
- Can create directories: disabled

Listing 5-2 shows a method you might implement in one of your `NSDocument` subclasses to import some files and associate them with the document. After configuring the panel, this method uses the `beginSheetModalForWindow:completionHandler:` method to present the panel as a sheet attached to the document's main window. If the user selects some files, the `URLs` method contains the corresponding file and directory references to incorporate. Because the panel is attached to the document window, you do not need to retain it explicitly in your code.

Listing 5-2 Associating an Open panel with a window

```
- (IBAction)importFilesAndDirectories:(id)sender {
    // Get the main window for the document.
    NSWindow* window = [[[self windowControllers] objectAtIndex:0] window];

    // Create and configure the panel.
    NSOpenPanel* panel = [NSOpenPanel openPanel];
    [panel setCanChooseDirectories:YES];
```

```
    [panel setAllowsMultipleSelection:YES];
    [panel setMessage:@"Import one or more files or directories."];

    // Display the panel attached to the document's window.
    [panel beginSheetModalForWindow:window completionHandler:^(NSInteger result){
       if (result == NSFileHandlingPanelOKButton) {
          NSArray* urls = [panel URLs];

          // Use the URLs to build a list of items to import.
       }

     }];
}
```

For single-window apps, associate the open panel with your app's main window. For document-based apps, associate it with the main window of the current document.

The Save Panel: Getting a New File Name

For document-based apps, requests to save the current document should be handled by the document object itself. The `NSDocument` class implements sophisticated infrastructure to manage both the Save panel and save-related operations, such as autosaving. As a result, the document object often customizes the default Save panel significantly and making additional customizations is not recommended except through the existing `NSDocument` methods. You could still use a custom Save panel to handle the exporting of files using different file formats.

If you are not using the `NSDocument` infrastructure, you can create your own Save panels and display them at appropriate times. The `NSSavePanel` class provides methods for creating and customizing the default Save panel. When implementing your own document infrastructure, restrict your use of a Save panel to documents the user creates and wants to save. Do not use a Save panel for files your app creates and manages implicitly. For example, iPhoto does not prompt the user to specify the location of its main library file; it creates the file in a well-known location without any user interactions. However, iPhoto does present a Save panel when the user exports an image to disk.

Listing 5-3 shows a method that a document object might call when exporting a file to a new type. This method assembles a new filename from the document's existing name and the new type being saved. It then presents the Save panel with the new name set as the default value and initiates the save operation as appropriate.

Listing 5-3 Saving a file with a new type

```
- (void)exportDocument:(NSString*)name toType:(NSString*)typeUTI
{
   NSWindow*       window = [[[self windowControllers] objectAtIndex:0] window];

   // Build a new name for the file using the current name and
   // the filename extension associated with the specified UTI.
   CFStringRef newExtension = UTTypeCopyPreferredTagWithClass((CFStringRef)typeUTI,
                                    kUTTagClassFilenameExtension);
   NSString* newName = [[name stringByDeletingPathExtension]
                        stringByAppendingPathExtension:(NSString*)newExtension];
   CFRelease(newExtension);

   // Set the default name for the file and show the panel.
   NSSavePanel*    panel = [NSSavePanel savePanel];
   [panel setNameFieldStringValue:newName];
   [panel beginSheetModalForWindow:window completionHandler:^(NSInteger result){
        if (result == NSFileHandlingPanelOKButton)
        {
            NSURL*  theFile = [panel URL];

            // Write the contents in the new format.
        }
    }];
}
```

For more information about implementing a document-based app, see *Mac App Programming Guide*.

Filters Limit the File Types That the User Can Select

If your app is able to open only specific file types, you can use filtering to restrict the user's selections to the subset of files your app actually supports. The Open panel does not restrict the types of files the user can select by default. To prevent the user from selecting files your app cannot handle, install one or more filters on the Open panel. Files that do not match the provided filters are dimmed by the Open panel and cannot be selected by the user. You can also install filters on the Save panel to dim unknown file types.

You specify filters as UTIs, filename extensions, or a combination of the two. After collecting your filter strings into an array, you assign them to the panel using the `setAllowedFileTypes:` method. The panel handles the actual filtration of files for you. You can change the filters while the panel is visible.

Listing 5-4 shows an example of how to configure the Open panel to allow the selection of image types only. The `NSImage` class provides a convenience method for retrieving the supported image types. Because the strings in the returned array are UTIs, the results are handed directly to the panel.

Listing 5-4 Filtering for image file types

```
- (IBAction)askUserForImage:(id)sender {
   NSOpenPanel*    panel = [[NSOpenPanel openPanel] retain];

   // Let the user select any images supported by
   // the NSImage class.
   NSArray* imageTypes = [NSImage imageTypes];
   [panel setAllowedFileTypes:imageTypes];

   [panel beginWithCompletionHandler:^(NSInteger result){
      if (result == NSFileHandlingPanelOKButton) {
         // Open the image.
      }

      [panel release];
   }];
}
```

Adding an Accessory View to the Open and Save Panels

To present the user with additional options when opening or saving files, add an accessory view to the standard Open and Save panels. Normally, the Open and Save panels process events and handle all interactions until the user cancels or accepts the results. An **accessory view** lets you add custom controls and views for gathering information or modify the behavior of the panel itself. For example, you might use custom controls to change the set of filters used by the panel or change the options your own code uses when opening or saving a file.

The basic process for adding an accessory view to an Open or Save panel is as follows:

1. Load the accessory view from a nib file. (You can create accessory views programmatically too, but using a nib file is often easier.)
2. Create the panel.
3. Associate your view with the panel using the `setAccessoryView:` method.
4. Show the panel.

Compare the preceding steps to displaying a panel normally and the only difference is loading your view and calling the `setAccessoryView:` method. All of the other work required to manage an accessory view and respond to events is your responsibility.

Defining the Contents of Your Accessory View

Because an accessory view is just another view, you define it the way you define other views in your app. An accessory view always consists of a single main view that may itself contain any number of subviews. The most common configuration for an accessory view is as a host for one or more controls. When the user interacts with the controls in your accessory view, those controls use the target-action design pattern to call your app's custom code. You can then use that code to store configuration options or take any relevant actions.

A nib file is the simplest way to define an accessory view. Before setting up the nib file, you need to know which object in your app is going to present the panel. For example, in a document-based app, your `NSDocument` subclass is usually responsible for displaying any Open and Save panels. In general, the object that presents the panel should also own the contents of the accessory view. With that in mind, the configuration of your nib file would be as follows:

1. Create a new nib file whose contents are a single view.
2. Set the File's Owner of the nib file to the object that presents the panel.
3. Configure the contents of the nib file's top-level view object.
    - Add any subviews, such as checkboxes, that you want to include in the accessory view.

- You can change the class of the top-level view object if you want, but doing so is not required. If you use a generic view as the main view, the underlying panel provides the appropriate background appearance for the rest of your content.

4. Connect actions (as appropriate) to your views to facilitate interactions with your custom code. You should implement the action methods themselves in the object you assigned as File's Owner.
5. Connect outlets (as appropriate) for any controls you use to gather data passively from the user. The completion handler for the panel can then use the outlets to access the data in the controls.
6. Size your accessory view to be as small as possible while still showing all subviews in their entirety.
7. Save the nib file.

When displaying an accessory view, the Open and Save panels shows your entire accessory view. If your accessory view is larger than the panel itself, the panel grows to accommodate your view. If the panel grows too large, it could look strange or could cause problems on smaller screens. Most accessory views should have only a few controls anyway. So if you find yourself adding more than ten controls, you might want to consider whether an accessory view is appropriate or if there is a better way to gather the information.

For more information about configuring your views and the rest of your nib file, see *Xcode User Guide*.

Loading and Configuring Your Accessory View at Runtime

Immediately prior to displaying an Open or Save panel, you need to load (or create) your accessory view and attach it to the panel using the `setAccessoryView:` method. If you create your accessory view programmatically, you would normally create it right before presenting the panel itself. If you stored your accessory view in a nib file, you need to load that nib file first.

Loading the nib file for an accessory view is relatively simple as long as the nib file itself is configured properly. The easiest way to configure the nib file is to assign the object that presents the panel as the File's Owner of the nib file itself. That way, any outlets and actions defined on the object are connected automatically and ready to use as soon as the nib file is loaded into memory. Listing 5-5 shows an example of how this works. The method in this example is implemented on an `NSDocument` object. When called, the method presents an Open panel with an accessory view attached to the document's main window. The accessory view itself contains a single checkbox that is connected to a custom `optionCheckbox` outlet that the document object defines. (The outlet itself is implemented using a property.) The handler block uses the value of the checkbox to determine how to open the file.

Listing 5-5 Loading a nib file with an accessory view

```
- (IBAction)openFileWithOptions:(id)sender {
    NSOpenPanel*    panel = [NSOpenPanel openPanel];
```

```
    NSWindow*       window = [[[self windowControllers] objectAtIndex:0] window];

    // Load the nib file with the accessory view and attach it to the panel.
    if ([NSBundle loadNibNamed:@"MyAccessoryView" owner:self])
        [panel setAccessoryView:self.accessoryView];

    // Show the open panel and process the results.
    [panel beginSheetModalForWindow:window completionHandler:^(NSInteger result){
        if (result == NSFileHandlingPanelOKButton) {
            BOOL option1 = ([self.optionCheckbox state] == NSOnState);

            // Open the selected file using the specified option...
        }
    }];
}
```

If you were creating your accessory view programmatically, you could replace the if statement containing the call to the `loadNibNamed:owner:` method with your custom view-creation code.

For more information on how to load objects from nib files, see *Resource Programming Guide*.

Managing Files and Directories

Some of the most basic operations involving files and directories are creating them and moving them around the file system. These operations are how your app builds the file system structure it needs to perform its tasks. For most operations, the `NSFileManager` class should offer the functionality you need to create and manipulate files. In the rare cases where it does not, you need to use BSD-level functions directly.

Creating New Files and Directories Programmatically

Creating files and directories is a basic part of file management. Typically, you create custom directories to organize the files created by your code. For example, you might create some custom directories in your app's `Application Support` directory to store any private data files managed by your app. And there are many ways to create files.

Creating Directories

When you want to create a custom directory, you do so using the methods of `NSFileManager`. A process can create directories anywhere it has permission to do so, which always includes the current home directory and may include other file system locations as well. You specify the directory to create by building a path to it and passing your `NSURL` or `NSString` object to one of the following methods:

- `createDirectoryAtURL:withIntermediateDirectories:attributes:error:` (OS X 10.7 and later only)
- `createDirectoryAtPath:withIntermediateDirectories:attributes:error:`

Listing 6-1 shows how to create a custom directory for app files inside the `~/Library/Application Support` directory. This method creates the directory if it does not exist and returns the path to the directory to the calling code. Because this method touches the file system every time, you would not want to call this method repeatedly to retrieve the URL. Instead, you might call it once and then cache the returned URL.

Listing 6-1 Creating a custom directory for app files

```
- (NSURL*)applicationDirectory
{
    NSString* bundleID = [[NSBundle mainBundle] bundleIdentifier];
```

```
    NSFileManager*fm = [NSFileManager defaultManager];
    NSURL*    dirPath = nil;

    // Find the application support directory in the home directory.
    NSArray* appSupportDir = [fm URLsForDirectory:NSApplicationSupportDirectory
                                     inDomains:NSUserDomainMask];
    if ([appSupportDir count] > 0)
    {
        // Append the bundle ID to the URL for the
        // Application Support directory
        dirPath = [[appSupportDir objectAtIndex:0]
URLByAppendingPathComponent:bundleID];

        // If the directory does not exist, this method creates it.
        // This method call works in OS X 10.7 and later only.
        NSError*    theError = nil;
        if (![fm createDirectoryAtURL:dirPath withIntermediateDirectories:YES
                    attributes:nil error:&theError])
        {
            // Handle the error.

            return nil;
        }
    }

    return dirPath;
}
```

If your code needs to run in OS X 10.6 and earlier, you can replace any calls to the `createDirectoryAtURL:withIntermediateDirectories:attributes:error:` method with a similar call to the `createDirectoryAtPath:withIntermediateDirectories:attributes:error:` method. The only change you have to make is to pass a string-based path instead of a URL as the first parameter. However, the `NSURL` class defines a `path` method that returns a string-based version of its path.

The `NSFileManager` methods are the preferred way to create new directories because of their simplicity. However, you can also use the `mkdir` function to create directories yourself. If you do so, you are responsible for creating intermediate directories and handling any errors that occur.

Creating New Files

There are two parts to creating a file: creating a record for the file in the file system and filling the file with content. All of the high-level interfaces for creating files perform both tasks at the same time, usually filling the file with the contents of an `NSData` or `NSString` object and then closing the file. You can also use lower-level functions to create an empty file and obtain a file descriptor that you can then use to fill the file with data. Some of the routines you can use to create files are:

- `createFileAtPath:contents:attributes:` (`NSFileManager`)
- `writeToURL:atomically:` (`NSData`)
- `writeToURL:atomically:` (`NSString`)
- `writeToURL:atomically:encoding:error:` (`NSString`)
- `writeToURL:atomically:` (Various collection classes define this method for writing out property lists.)
- `open` function with the `O_CREAT` option creates a new empty file

> **Note:** Any files you create inherit the permissions associated with the current user and process.

When writing the contents of a new file all at once, the system routines typically close the file after writing the contents to disk. If the routine returns a file descriptor, you can use that descriptor to continue reading and writing from the file. For information on how to read and write the contents of a file, see "Techniques for Reading and Writing Files Without File Coordinators" (page 73).

Copying and Moving Files and Directories

To copy items around the file system, you use the `NSFileManager` class provides the `copyItemAtURL:toURL:error:` and `copyItemAtPath:toPath:error:` methods. To move the files use the `moveItemAtURL:toURL:error:` or `moveItemAtPath:toPath:error:` methods.

You use the preceding methods to move or copy a single file or directory at a time. When moving or copying a directory, the directory and all of its contents are affected. The semantics for move and copy operations are the same as in the Finder. Move operations on the same volume do not cause a new version of the item to be

created. Move operations between volumes behave the same as a copy operation. And when moving or copying items, the current process must have permission to read the items and move or copy them to the new location.

Move and copy operations can potentially take a long time to complete, and the `NSFileManager` class performs these operations synchronously. Therefore, it is recommended that you execute any such operations on a concurrent dispatch queue and not on your app's main thread. Listing 6-2 shows an example that does just that by asynchronously creating a backup of a fictional app's private data. (For the sake of the example, the private data is located in the `~/Library/Application Support/`<*bundleID*>`/Data` directory, where <*bundleID*> is the actual bundle identifier of the app.) If the first attempt to copy the directory fails, this method checks to see whether a previous backup exists and removes it if it does. It then proceeds to try again and aborts if it fails a second time.

Listing 6-2 Copying a directory asynchronously

```
- (void)backupMyApplicationData {
   // Get the application's main data directory
   NSArray* theDirs = [[NSFileManager defaultManager]
URLsForDirectory:NSApplicationSupportDirectory
                                  inDomains:NSUserDomainMask];
   if ([theDirs count] > 0)
   {
      // Build a path to ~/Library/Application Support/<bundle_ID>/Data
      // where <bundleID> is the actual bundle ID of the application.
      NSURL* appSupportDir = (NSURL*)[theDirs objectAtIndex:0];
      NSString* appBundleID = [[NSBundle mainBundle] bundleIdentifier];
     NSURL* appDataDir = [[appSupportDir URLByAppendingPathComponent:appBundleID]
                              URLByAppendingPathComponent:@"Data"];

      // Copy the data to ~/Library/Application Support/<bundle_ID>/Data.backup
      NSURL* backupDir = [appDataDir URLByAppendingPathExtension:@"backup"];

      // Perform the copy asynchronously.
      dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0), ^{
        // It's good habit to alloc/init the file manager for move/copy operations,
         // just in case you decide to add a delegate later.
         NSFileManager* theFM = [[NSFileManager alloc] init];
```

```
        NSError* anError;

        // Just try to copy the directory.
        if (![theFM copyItemAtURL:appDataDir toURL:backupDir error:&anError]) {
         // If an error occurs, it's probably because a previous backup directory
           // already exists.  Delete the old directory and try again.
           if ([theFM removeItemAtURL:backupDir error:&anError]) {
              // If the operation failed again, abort for real.
            if (![theFM copyItemAtURL:appDataDir toURL:backupDir error:&anError])
 {
                 // Report the error....
              }
           }
        }

     });
   }
}
```

For details on how to use the `NSFileManager` methods, see *NSFileManager Class Reference*.

Deleting Files and Directories

To delete files and directories, use the following methods of the `NSFileManager` class:

- `removeItemAtURL:error:`
- `removeItemAtPath:error:`

When using these methods to delete files, realize that you are permanently removing the files from the file system; these methods do not move the file to the Trash where it can be recovered later.

For details on how to use the `NSFileManager` methods, see *NSFileManager Class Reference*.

Creating and Handling Invisible Files

In OS X, you make a file invisible to the user by prepending its name with a period character. Adding this character signals to the Finder and other display-related interfaces that they should treat the file as invisible under normal conditions. You should use this capability sparingly, though, and only in very specific cases. For example, the process for localizing a custom directory name involves placing hidden files with the localized names at the top-level of the directory. You should never use this technique to hide user documents or other data your app creates.

Even if you mark a file as invisible, there are still ways for the user to see it. Users can always view the true file system contents using the Terminal app. Unlike Finder, the command-line of the terminal displays the actual names of items in the file system and not their display names or localized names. You can also show hidden files in the Open and Save panels (using the `setShowsHiddenFiles:` method) in cases where you want the user to be able to select them for viewing or editing.

Regardless of whether a file contains a leading period, your app's code can always see hidden files. Methods that enumerate the contents of directories generally include hidden files in the enumeration. The only exceptions to this rule is the meta directories `.` and `..` that represent the current and parent directories. Therefore, any code that enumerates the contents of a directory should be prepared to handle hidden files and deal with them appropriately, usually by just ignoring them.

Techniques for Reading and Writing Files Without File Coordinators

Reading and writing files involves transferring a sequence of bytes between your code and the underlying disk. This is the lowest-level form of file management but is also the foundation for more sophisticated techniques as well. At some point, even the most sophisticated data structures have to be turned into a sequence of bytes before they can be stored on disk. Similarly, that same data must be read from the disk as a sequence of bytes before it can be used to reconstruct the more sophisticated data structures that it represents.

There are several different technologies for reading and writing the contents of files, nearly all of which are supported by both iOS and OS X. All of them do essentially the same thing but in slightly different ways. Some technologies require you to read and write file data sequentially, while others may allow you to jump around and operate on only part of a file. Some technologies provide automatic support for reading and writing asynchronously, while others execute synchronously so that you have more control over their execution.

Choosing from the available technologies is a matter of deciding how much control you want over the reading and writing process and how much effort you want to spend writing your file management code. Higher-level technologies like Cocoa streams limit your flexibility but provide an easy-to-use interface. Lower-level technologies like POSIX and Grand Central Dispatch (GCD) give you maximum flexibility and power but require you to write a little more code.

Reading and Writing Files Asynchronously

Because file operations involve accessing a disk (possibly one on a network server), performing those operations asynchronously is almost always preferred. Technologies such as Cocoa streams and Grand Central Dispatch (GCD) are designed to execute asynchronously at all times. This is not to say that other technologies cannot be used in an asynchronous way but that these technologies provide give you the asynchronous behavior automatically. This allows you to focus on reading and writing file data rather than worrying about where your code executes.

Processing an Entire File Linearly Using Streams

If you always read or write a file's contents from start to finish, streams provide a simple interface for doing so asynchronously. Streams are typically used for managing sockets and other types of sources where data may become available over time. However, you can also use streams to read or write an entire file in one or more bursts. There are two types of streams available:

- Use an `NSOutputStream` to write data sequentially to disk.
- Use an `NSInputStream` object to read data sequentially from disk.

Stream objects use the run loop of the current thread to schedule read and write operations on the stream. An input stream wakes up the run loop and notifies its delegate when there is data waiting to be read. An output stream wakes up the run loop and notifies its delegate when there is space available for writing data. When operating on files, this behavior usually means that the run loop is woken up several times in quick succession so that your delegate code can read or write the file data. It also means that your delegate code is called repeatedly until you close the stream object, or in the case of input streams until the stream reaches the end of the file.

For information and examples about how to set up and use stream objects to read and write data, see *Stream Programming Guide*.

Processing a File Using GCD

Grand Central Dispatch provides several different ways to read or write the content of files asynchronously:

- Create a dispatch I/O channel and use it to read or write data. (OS X 10.7 and later. Not supported in iOS.)
- Use the `dispatch_read` or `dispatch_write` convenience functions to perform a single asynchronous operation. (OS X 10.7 and later. Not supported in iOS.)
- Create a dispatch source to schedule the execution of a custom event handler block, in which you use standard POSIX calls to read or write data from the file.

Dispatch I/O channels are the preferred way to read and write files because they give you direct control over when file operations occur but still allow you to process the data asynchronously on a dispatch queue. A dispatch I/O channel is an `dispatch_io_t` structure that identifies the file whose contents you want to read or write. Channels can be configured for stream-based access or random access of files. A stream-based channel forces you to read or write file data sequentially, whereas a random-access channel lets you read or write at any offset from the beginning of the file.

If you do not want the trouble of creating and managing a dispatch I/O channel, you can use the `dispatch_read` or `dispatch_write` functions to perform a single read or write operation on a file descriptor. These methods are convenient for situations where you do not want or need the overhead of creating and managing a dispatch I/O channel. However, you should use them only when performing a single read or write operation on a file. If you need to perform multiple operations on the same file, creating a dispatch I/O channel is much more efficient.

Dispatch sources allow you to process files in a way that is similar to Cocoa stream objects. Like stream objects, they are used more often with sockets or data sources that send and receive data sporadically but they can still be used with files. A dispatch source schedules its associated event handler block whenever there is data waiting to be read or space available for writing. For files, this usually results in the block being scheduled repeatedly and in quick succession until you explicitly cancel the dispatch source or it reaches the end of the file it is reading.

For more information about creating and using dispatch sources, see *Concurrency Programming Guide*. For information about the `dispatch_read` or `dispatch_write` functions, or any other GCD functions, see *Grand Central Dispatch (GCD) Reference*.

Creating and Using a Dispatch I/O Channel

To create a dispatch I/O channel, you must provide either a file descriptor or the name of the file you want to open. If you already have an open file descriptor, passing it to the channel changes the ownership of that file descriptor from your code to the channel. A dispatch I/O channel takes control of its file descriptor so that it can reconfigure that file descriptor as needed. For example, the channel usually reconfigures the file descriptor with the `O_NONBLOCK` flag so that subsequent read and write operations do not block the current thread. Creating a channel using a file path causes the channel to create the necessary file descriptor and take control of it.

Listing 7-1 shows a simple example of how to open a dispatch I/O channel using an `NSURL` object. In this case, the channel is configured for random read-access and assigned to a custom property of the current class. The queue and block act to clean up the channel in the event that an error occurs during creation or at the end of the channel's lifecycle. If an error occurs during creation, you can use the error code to determine what happened. An error code of 0 indicates that the channel relinquished control of its file descriptor normally, usually as a result of calling the `dispatch_io_close` function, and that you can now dispose of the channel safely.

Listing 7-1 Creating a dispatch I/O channel

```
-(void)openChannelWithURL:(NSURL*)anURL {
   NSString* filePath = [anURL path];
   self.channel = dispatch_io_create_with_path(DISPATCH_IO_RANDOM,
                     [filePath UTF8String],   // Convert to C-string
                     O_RDONLY,                // Open for reading
                     0,                       // No extra flags
                     dispatch_get_main_queue(),
                     ^(int error){
                        // Cleanup code for normal channel operation.
```

```
                        // Assumes that dispatch_io_close was called elsewhere.
                        if (error == 0) {
                            dispatch_release(self.channel);
                            self.channel = NULL;
                        }
                    });
}
```

After creating a dispatch channel, you can store a reference to the resulting `dispatch_io_t` structure and use it to initiate read or write calls at your convenience. If you created a channel that supports random access, you can start reading or writing at any location. If you create a stream-based channel, any offset value you specify as a starting point is ignored and data is read or written at the current location. For example, to read the second 1024 bytes from a channel that supports random access, your read call might look similar to the following:

```
dispatch_io_read(self.channel,
                 1024,                      // 1024 bytes into the file
                 1024,                      // Read the next 1024 bytes
                 dispatch_get_main_queue(),   // Process the bytes on the main
thread
                 ^(bool done, dispatch_data_t data, int error){
                     if (error == 0) {
                        // Process the bytes.
                     }
                 });
```

A write operation requires you to specify the bytes you want written to the file, the location at which to begin writing (for random access channels), and a handler block with which to receive progress reports. You initiate write operations using the `dispatch_io_write` function, which is described in *Grand Central Dispatch (GCD) Reference*.

Manipulating Dispatch Data for an I/O Channel

All channel-based operations use `dispatch_data_t` structures to manipulate the data read or written using a channel. A `dispatch_data_t` structure is an opaque type that manages one or more contiguous memory buffers. The use of an opaque type allows GCD to use discontiguous buffers internally while still presenting the data to your app as if it were more or less contiguous. The actual implementation details of how dispatch data structures work is not important, but understanding how to create them or get data out of them is.

To write data to a dispatch I/O channel, your code must provide a `dispatch_data_t` structure with the bytes to write. You do this using the `dispatch_data_create` function, which takes a pointer to a buffer and the size of the buffer and returns a `dispatch_data_t` structure that encapsulates the data from that buffer. How the data object encapsulates the buffer depends on the destructor you provide when calling the `dispatch_data_create` function. If you use the default destructor, the data object makes a copy of the buffer and takes care of releasing that buffer at the appropriate time. However, if you do not want the data object to copy the buffer you provide, you must provide a custom destructor block to handle any needed cleanup when the data object itself is released.

> **Note:** If you have multiple data buffers that you want to write to a file as a single contiguous block of data, you can create a single dispatch data object that represents all of those buffers. Using the `dispatch_data_create_concat` function, you can append additional data buffers to a `dispatch_data_t` structure. The buffers themselves can all be independent and in different parts of memory but the dispatch data object collects them and represents them as a single entity. (You can even use the `dispatch_data_create_map` function to generate a contiguous version of your buffers.) Especially for disk-based operations, concatenating multiple buffers lets you write large amounts of data to a file using one call to the `dispatch_io_write` function, which is much more efficient than calling `dispatch_io_write` separately for each independent buffer.

To extract bytes from a dispatch data object, you use the `dispatch_data_apply` function. Because dispatch data objects are opaque, you use this function to iterate over the buffers in the object and process them using a block that you provide. For a dispatch data object with a single contiguous buffer, your block is called once. For a data object with multiple buffers, your block is called as many times as there are buffers. Each time your block is called, it is passed a data buffer and some information about that buffer.

Listing 7-2 shows an example that opens a channel and reads a UTF8 formatted text file, creating `NSString` objects for the contents of the file. This particular example reads 1024 bytes at a time, which is an arbitrary amount and may not yield the best performance. However, it does demonstrate the basic premise of how to use the `dispatch_io_read` function in combination with the `dispatch_data_apply` function to read the bytes and then convert them into a form that your app might want. In this case, the block that processes the bytes uses the dispatch data object's buffer to initialize a new string object. It then hands the string off to the custom `addString:toFile:` method, which in this case would retain the string and store it for use later.

Listing 7-2 Reading the bytes from a text file using a dispatch I/O channel

```
- (void)readContentsOfFile:(NSURL*)anURL {
   // Open the channel for reading.
   NSString*   filePath = [anURL path];
   self.channel = dispatch_io_create_with_path(DISPATCH_IO_RANDOM,
                        [filePath UTF8String],   // Convert to C-string
                        O_RDONLY,                // Open for reading
                        0,                       // No extra flags
                        dispatch_get_main_queue(),
                        ^(int error){
                           // Cleanup code
                           if (error == 0) {
                              dispatch_release(self.channel);
                              self.channel = nil;
                           }
                        });

   // If the file channel could not be created, just abort.
   if (!self.channel)
      return;

    // Get the file size.
    NSNumber* theSize;
    NSInteger fileSize = 0;
    if ([anURL getResourceValue:&theSize forKey:NSURLFileSizeKey error:nil])
        fileSize = [theSize integerValue];

   // Break the file into 1024 size strings.
   size_t chunkSize = 1024;
   off_t  currentOffset = 0;

   for (currentOffset = 0; currentOffset < fileSize; currentOffset += chunkSize)
{
      dispatch_io_read(self.channel, currentOffset, chunkSize,
dispatch_get_main_queue(),
```

```
                    ^(bool done, dispatch_data_t data, int error){
                        if (error)
                            return;

                        // Build strings from the data.
                        dispatch_data_apply(data,
(dispatch_data_applier_t)^(dispatch_data_t region,
                                                size_t offset, const void *buffer,
 size_t size){
                            NSAutoreleasePool* pool = [[NSAutoreleasePool alloc]
init];
                            NSString* aString = [[[NSString alloc]
initWithBytes:buffer
                                                 length:size encoding:NSUTF8StringEncoding]
 autorelease];

                            [self addString:aString toFile:anURL];  // Custom method.
                            [pool release];
                           return true;  // Keep processing if there is more data.
                        });

                    });
    }
}
```

For more information about the functions you use to manipulate dispatch data objects, see *Grand Central Dispatch (GCD) Reference*.

Reading and Writing Files Synchronously

The file-related interfaces that operate on data synchronously give you the flexibility to set your code's execution context yourself. Just because they execute synchronously does not mean that they are less efficient than their asynchronous counterparts. On the contrary, it just means that the interface itself does not provide the asynchronous execution context automatically. If you want to read and write data asynchronously using these technologies, and get all the same benefits, you must provide the asynchronous execution context yourself. Of course, the best way to do that is to execute your code using GCD dispatch queues or operation objects.

Building Your Content in Memory and Writing It to Disk All at Once

The simplest way to manage the reading and writing of file data is all at once. This works well for custom document types where you expect the size of the document's on-disk representation to remain reasonably small. You would not want to do this for multimedia files or for files whose size can grow to many megabytes in size.

For custom document types that use a binary or private file format, you can use the `NSData` or `NSMutableData` class to transfer your custom data to and from disk. You can create new data objects in many different ways. For example, you can use a keyed archiver object to convert a graph of objects into a linear stream of bytes enclosed in a data object. If you have a binary file format that is very structured, you can append bytes to an `NSMutableData` object and build your data object piece by piece. When you are ready to write the data object to disk, use the `writeToURL:atomically:` or `writeToURL:options:error:` method. These methods allow you to create the corresponding on-disk file in one step.

> **Note:** In iOS, one of the options you can pass to the `writeToURL:options:error:` method allows you to specify whether you want the contents of the file encrypted. If you specify one of these options, the contents are encrypted immediately to ensure the security of the file.

To read data back from disk, use the `initWithContentsOfURL:options:error:` method to obtain a data object based on the contents of your file. You can use this data object to reverse the process you used when creating it. Thus, if you used a keyed archiver to create the data object, you can use a keyed unarchiver to re-create your object graph. If you wrote the data out piece by piece, you can parse the byte stream in the data object and use it to reconstruct your document's data structures.

Apps that use the `NSDocument` infrastructure typically interact with the file system indirectly using `NSData` objects. When the user saves a document, the infrastructure prompts the corresponding `NSDocument` object for a data object to write to disk. Similarly, when the user opens an existing document, it creates the document object and passes it a data object with which to initialize itself.

For more information about the `NSData` and `NSMutableData` classes, see *Foundation Framework Reference*. For more information about using the document infrastructure in an OS X app, see *Mac App Programming Guide*.

Reading and Writing Files Using NSFileHandle

The use of the `NSFileHandle` class closely parallels the process for reading and writing files at the POSIX level. The basic process is that you open the file, issue read or write calls, and close the file when you are done. In the case of `NSFileHandle`, you open the file automatically when you create an instance of the class. The

file handle object acts as a wrapper for the file, managing the underlying file descriptor for you. Depending on whether you requested read access, write access, or both, you then call the methods of `NSFileHandle` to read or write actual bytes. When you are done, you release your file handle object to close the file.

Listing 7-3 shows a very simple method that reads the entire contents of a file using a file handle object. The `fileHandleForReadingFromURL:error:` method creates the file handle object as an autoreleased object, which causes it to be released automatically at some point after this method returns.

Listing 7-3 Reading the contents of a file using `NSFileHandle`

```
- (NSData*)readDataFromFileAtURL:(NSURL*)anURL {
    NSFileHandle* aHandle = [NSFileHandle fileHandleForReadingFromURL:anURL
error:nil];
    NSData* fileContents = nil;


    if (aHandle)
        fileContents = [aHandle readDataToEndOfFile];


    return fileContents;
}
```

For more information about the methods of the `NSFileHandle` class, see *NSFileHandle Class Reference*.

Managing Disk Reads and Writes at the POSIX Level

If you prefer to use C-based functions for your file management code, the POSIX layer offers standard functions for working with files. At the POSIX level, you identify a file using a file descriptor, which is an integer value that identifies an open file uniquely within your app. You pass this file descriptor to any subsequent functions that require it. The following list contains the main POSIX functions you use to manipulate files:

- Use the `open` function to obtain a file descriptor for your file.
- Use the `pread`, `read`, or `readv` function to read data from an open file descriptor. For information about these functions, see `pread`.
- Use the `pwrite`, `write`, or `writev` function to write data to an open file descriptor. For information about these functions, see `pwrite`.
- Use the `lseek` function to reposition the current file pointer and change the location at which you read or write data.
- Use the `pclose` function to close the file descriptor when you are done with it.

Important: On some file systems, there is no guarantee that a successful write call results in the actual writing of bytes to the file system. For some network file systems, written data might be sent to the server at some point after your write call. To verify that the data actually made it to the file, use the `fsync` function to force the data to the server or close the file and make sure it closed successfully—that is, the `close` function did not return `-1`.

Listing 7-4 shows a simple function that uses POSIX calls to read the first 1024 bytes of a file and return them in an `NSData` object. If the file has fewer than 1024 bytes, the method reads as many bytes as possible and truncates the data object to the actual number of bytes.

Listing 7-4 Reading the contents of a file using POSIX functions

```
- (NSData*)readDataFromFileAtURL:(NSURL*)anURL {
    NSString* filePath = [anURL path];
    fd = open([filePath UTF8String], O_RDONLY);
    if (fd == -1)
        return nil;

    NSMutableData* theData = [[[NSMutableData alloc] initWithLength:1024]
autorelease];
    if (theData) {
        void* buffer = [theData mutableBytes];
        NSUInteger bufferSize = [theData length];

        NSUInteger actualBytes = read(fd, buffer, bufferSize);
        if (actualBytes < 1024)
            [theData setLength:actualBytes];
    }

    close(fd);
    return theData;
}
```

Because there is a limit to the number of open file descriptors an app may have at any given time, you should always close file descriptors as soon as you are done using them. File descriptors are used not only for open files but for communications channels such as sockets and pipes. And your code is not the only entity creating file descriptors for your app. Every time you load a resource file or use a framework that communicates over

the network, the system creates a file descriptor on behalf of your code. If your code opens large numbers of sockets or files and never closes them, system frameworks may not be able to create file descriptors at critical times.

Getting and Setting File Metadata Information

Files contain a lot of useful information but so does the file system. For each file and directory, the file system stores meta information about things like the item's size, creation date, owner, permissions, whether a file is locked, or whether a file's extension is hidden. There are several ways to get and set this information but the most prominent ways are:

- Get and set most content metadata information (including Apple-specific attributes) using the `NSURL` class.
- Get and set basic file-related information using the `NSFileManager` class.

The `NSURL` class offers a wide range of file-related information, including information that is standard for the file system (such as file size, type, owner, and permissions) but also a lot of Apple-specific information (such as the assigned label, localized name, the icon associated with the file, whether the file is a package, and so on). In addition, some methods that take URLs as arguments allow you to cache attributes while you are performing other operations on the file or directory. Especially when accessing large numbers of files, this type of caching behavior can improve performance by minimizing the number of disk-related operations. Regardless of whether attributes are cached, you retrieve them from the NSURL object using its `getResourceValue:forKey:error:` method and set new values for some attributes using the `setResourceValue:forKey:error:` or `setResourceValues:error:` method.

Even if you are not using the `NSURL` class, you can still get and set some file-related information using the `attributesOfItemAtPath:error:` and `setAttributes:ofItemAtPath:error:` methods of the `NSFileManager` class. These methods let you retrieve information about file system items like their type, size, and the level of access currently supported. You can also use the `NSFileManager` class to retrieve more general information about the file system itself, such as its size, the amount of free space, the number of nodes in the file system, and so on. Note that you are only able to get a subset of resources for iCloud files.

For more information about the `NSURL` and `NSFileManager` classes, and the attributes you can obtain for files and directories, see *NSURL Class Reference* and *NSFileManager Class Reference*.

Using FileWrappers as File Containers

An `NSFileWrapper` instance holds a file's contents in dynamic memory. In this role it enables a document object to embed a file, treating it as a unit of data that can be displayed as an image (and possibly edited in place), saved to disk, or transmitted to another app. It can also store an icon for representing the file in a document or in a dragging operation.

Instances of this class are referred to as *file wrapper objects*, and when no confusion will result, merely as *file wrappers*. A file wrapper can be one of three specific types: a regular file wrapper, which holds the contents of a single actual file; a directory wrapper, which holds a directory and all of the files or directories within it; or a link wrapper, which simply represents a symbolic link in the file system.

Because the purpose of a file wrapper is to represent files in memory, it's very loosely coupled to any disk representation. A file wrapper doesn't record the path to the disk representation of its contents. This allows you to save the same file wrapper with different URLs, but it also requires you to record those URLs if you want to update the file wrapper from disk later.

> **Note:** When an `NSFileWrapper` instance is specified as the item for file coordination, all the files within the file wrapper are automatically part of that file coordination. It is not necessary to mange each file or directory individually.

Working with File Wrappers

You can create a file wrapper from data in memory using the `initWithSerializedRepresentation:` method or from data on disk using the `initWithURL:options:error:` method. Both create the appropriate type of file wrapper based on the nature of the serialized representation or of the file on disk.

Three convenience methods each create a file wrapper of a specific type: `initRegularFileWithContents:`, `initDirectoryWithFileWrappers:`, and `initSymbolicLinkWithDestination:`. Because each initialization method creates file wrappers of different types or states, they're all designated initializers for this class—subclasses must meaningfully override them all as necessary.

Some file wrapper methods apply only to a specific wrapper type, and an exception is raised if a method sent to a file wrapper of the wrong type. To determine the type of a file wrapper, use the `isRegularFile`, `isDirectory`, and `isSymbolicLink` methods.

A file wrapper stores file system information (such as modification time and access permissions), which it updates when reading from disk and uses when writing files to disk. The `fileAttributes` method returns this information in the format described in the `NSFileManager` method `attributesOfItemAtPath:error:`. You can also set the file attributes using the `setFileAttributes:` method.

The `NSFileWrapper` class allows you to set a preferred filename for save operations, and it records the last filename it was actually saved to; the `preferredFilename` and `filename` methods return these names. This feature is most important for directory wrappers, though, and so is discussed under "Working with Directory Wrappers."

When saving a file wrapper to disk, you typically determine the directory you want to save it in, then append the preferred filename to that directory URL, and use the `writeToURL:options:originalContentsURL:error:` method, which saves the file wrapper's contents and updates the file attributes. You can save a file wrapper under a different name if you like, but this doing so may result in the recorded filename differing from the preferred filename, depending on how you invoke the `writeToURL:options:originalContentsURL:error:` method.

Besides saving its contents to disk, a file wrapper can reread them from disk when necessary. The `matchesContentsOfURL:` method determines whether a disk representation may have changed, based on the file attributes stored the last time the file was read or written. If the file wrapper's modification time or access permissions are different from those of the file on disk, this method returns `YES`. You can then use `readFromURL:options:error:` to re-read the file from disk.

Finally, to transmit a file wrapper to another process or system (for example, using the pasteboard), you use the `serializedRepresentation` method to get an `NSData` object containing the file wrapper's contents in the `NSFileContentsPboardType` format. You can safely transmit this representation over whatever channel you choose. The recipient of the representation can then reconstitute the file wrapper using the `initWithSerializedRepresentation:` method.

Working with Directory Wrappers

A directory wrapper contains other file wrappers (of any type), and allows you to access them by keys derived from their preferred filenames. You can add any type of file wrapper to a directory wrapper with the `addFileWrapper:` method and remove it with the `removeFileWrapper:` method. The convenience methods `addRegularFileWithContents:preferredFilename:` and `addSymbolicLinkWithDestination:preferredFilename:` allow you to add regular file and link wrappers while also setting their preferred names.

A directory wrapper stores its contents in an `NSDictionary` object, which you can retrieve using the `fileWrappers` method. The keys of this dictionary are based on the preferred filenames of each file wrapper contained in the directory wrapper. There exist, then, three identifiers for a file wrapper within a directory wrapper:

- **Preferred filename.** This identifier doesn't uniquely identify the file wrapper, but the other identifiers are always based on it.
- **Dictionary key.** This identifier is always equal to the preferred name when there are no other file wrappers of the same preferred name in the same directory wrapper. Otherwise, it's a string made by adding a unique prefix to the preferred filename. Note that the same file wrapper can have a different dictionary key for each directory wrapper that contains it. You use the dictionary key to retrieve the file wrapper object in memory, in order to get its contents or its filename (that is, to update it from disk). You can get a file wrapper's dictionary key by sending a `keyForFileWrapper:` message to the directory wrapper that contains it.
- **Filename**. This identifier is usually based on the preferred filename, but isn't necessarily the same as it or the dictionary key. You use the filename to update a single file wrapper relative to the path of the directory wrapper that contains it. Note that the filename may change whenever you save a directory wrapper containing the file wrapper. Particularly if the file wrapper has been added to several different directory wrappers. Thus, you should always retrieve the filename from the file wrapper itself each time you need it rather than caching it.

When working with the contents of a directory wrapper, you can use a dictionary enumerator to retrieve each file wrapper and perform whatever operation you need. With the exceptions of saving and updating, a directory file wrapper defines no recursive operations for its contents. To set the file attributes for all contained file wrappers, or to perform any other such operation, you must define a recursive method that examines the type of each file wrapper and invokes itself again for any directory wrapper it encounters.

Performance Tips

If your app works with a lot of files, the performance of its file-related code is very important. Relative to other types of operations, accessing files on disk is one of the slowest operations a computer can perform. Depending on the size and number of files, it can take anywhere from a few milliseconds to several minutes to read files from a disk-based hard drive. Therefore, you should make sure your code performs as efficiently as possible under even light to moderate work loads.

If your app slows down or becomes less responsive when it starts working with files, use the Instruments app to gather some baseline metrics. Instruments can show you how much time your app spends operating on files and help you monitor various file-related activity. As you fix each problem, be sure to run your code in Instruments again and record the results so that you can verify whether your changes worked.

Things to Look For in Your Code

If you are not sure where to start looking for potential fixes to your file-related code, here are some tips on where to start looking.

- **Look for places where your code is reading lots of files (of any type) from disk.** Remember to look for places where you are loading resource files too. Are you actually using the data from all of those files right away? If not, you might want to load some of the files more lazily.
- **Look for places where you are using older file-system calls.** Most of your calls should be using Objective-C interfaces or block-based interfaces. You can use BSD-level calls too but should not use older Carbon-based functions that operate on `FSRef` or `FSSpec` data structures. Xcode generates warnings when it detects your code using deprecated methods and functions, so make sure you check those warnings.
- **Look for places where you are using callback functions or methods to process file data.** If a newer API is available that takes a block object, you might want to update your code to use that API instead.
- **Look for places where you are performing many small read or write operations on the same file.** Can you group those operations together and perform them all at once? For the same amount of data, one large read or write operation is usually more efficient than many small operations.

Use Modern File System Interfaces

When deciding which routines to call, choose ones that let you specify paths using NSURL objects over those that specify paths using strings. Most of the URL-based routines were introduced in OS X 10.6 and later and were designed from the beginning to take advantage of technologies like Grand Central Dispatch. This gives your code an immediate advantage on multicore computers while not requiring you to do much work.

You should also prefer routines that accept block objects over those that accept callback functions or methods. Blocks are a convenient and more efficient way to implement callback-type behaviors. In practice, blocks often require much less code to implement because they do not require you to define and manage a context data structure for passing data. Some routines might also execute your block by scheduling it in a GCD queue, which can also improve performance.

General Tips

What follows are some basic recommendations for reducing the I/O activity of your program. These may help improve your file-system related performance, but as with all tips be sure to measure before and after so that you can verify any performance gains.

- **Minimize the number of file operations you perform.** Moving data from a local file system into memory takes a significant amount of time. File-system access times are generally measured in milliseconds, which corresponds to several millions of clock cycles spent waiting for data to be fetched from disk. And if the target file system is located on a server halfway around the world, network latency increases the delay in retrieving the data.
- **Reuse path objects.** If you take the time to create an NSURL for a file, reuse that object as much as you can rather than create it each time you need it. Locating files and building URLs or pathname information takes time and can be expensive. Reusing the objects created from those operations saves time and minimizes your app's interactions with the file system.
- **Choose an appropriate read buffer size.** When reading data from the disk to a local buffer, the buffer size you choose can have a dramatic effect on the speed of the operation. If you are working with relatively large files, it does not make sense to allocate a 1K buffer to read and process the data in small chunks. Instead, create a larger buffer (say 128K to 256K in size) and read much or all of the data into memory before processing it. The same rules apply for writing data to the disk: write data as sequentially as you can using a single file-system call.
- **Read data sequentially instead of jumping around in a file.** The kernel transparently clusters I/O operations, which makes sequential reads much faster.

- **Avoid skipping ahead in an empty file before writing data.** The system might have to write zeroes into the intervening space to fill the gap. You should always have a good reason for including "holes" in your files at write time and should know that doing so might incur a performance penalty. For more information, see "Zero-Fill Delays Provide Security at a Cost" (page 90).
- **Defer I/O operations until your app needs the data.** The golden rule of being lazy applies to disk performance as well as many other types of performance.
- **Do not abuse the preferences system.** Use the preferences system to capture only user preferences (such as window positions, view settings, and user provided preferences) and not data that can be inexpensively recomputed. Recomputing simple values is significantly faster than reading the same value from disk.
- **Do not assume that caching files in memory will speed up your app.** Caching files in memory increases memory usage, which can decrease performance in other ways. Plus, the system may cache some file data for you automatically, so creating your own caches might make things even worse; see "The System Has its Own File Caching Mechanism" (page 89).

The System Has its Own File Caching Mechanism

Disk caching can be a good way to accelerate access to file data, but its use is not appropriate in every situation. Caching increases the memory footprint of your app and if used inappropriately can be more expensive than simply reloading data from the disk.

Caching is most appropriate for files you plan to access multiple times. If you have files you only intend to use once, you should either disable the caches or map the file into memory.

Disabling File-System Caching

When reading data that you are certain you won't need again soon, such as streaming a large multimedia file, tell the file system not to add that data to the file-system caches. By default, the system maintains a buffer cache with the data most recently read from disk. This disk cache is most effective when it contains frequently used data. If you leave file caching enabled while streaming a large multimedia file, you can quickly fill up the disk cache with data you won't use again. Even worse is that this process is likely to push other data out of the cache that might have benefited from being there.

Apps can call the BSD `fcntl` function with the `F_NOCACHE` flag to enable or disable caching for a file. For more information about this function, see `fcntl`.

Note: When reading uncached data, it is recommended that you use 4K-aligned buffers. This gives the system more flexibility in how it loads the data into memory and can result in faster load times.

Using Mapped I/O Instead of Caching

If you intend to read data randomly from a file, you can improve performance in some situations by mapping that file directly into your app's virtual memory space. File mapping is a programming convenience for files you want to access with read-only permissions. It lets the kernel take advantage of the virtual memory paging mechanism to read the file data only when it is needed. You can also use file mapping to overwrite existing bytes in a file; however, you cannot extend the size of file using this technique. Mapped files bypass the system disk caches, so only one copy of the file is stored in memory.

Important: If you map a file into memory and the file becomes inaccessible—because the disk containing the file was ejected or the network server containing the file is unmounted—your app will crash with a SIGBUS error.

For more information about mapping files into memory, see *File System Advanced Programming Topics*.

Zero-Fill Delays Provide Security at a Cost

For security reasons, file systems are supposed to zero out areas on disk when they are allocated to a file. This behavior prevents data leftover from a previously deleted file from being included with the new file. The HFS Plus file system used by OS X has always implemented this zero-fill behavior.

For both reading and writing operations, the system delays the writing of zeroes until the last possible moment. When you close a file after writing to it, the system writes zeroes to any portions of the file your code did not touch. When reading from a file, the system writes zeroes to new areas only when your code attempts to read from that area or when it closes the file. This delayed-write behavior avoids redundant I/O operations to the same area of a file.

If you notice a delay when closing your files, it is likely because of this zero-fill behavior. Make sure you do the following when working with files:

- Write data to files sequentially. Gaps in writing must be filled with zeros when the file is saved.
- Do not move the file pointer past the end of the file and then close the file.
- Truncate files to match the length of the data you wrote. For scratch files you plan to delete, truncate the file to zero-length.

OS X Library Directory Details

The `Library` directories are where the system and your code store all of their related data and resources. In OS X, this directory can contain many different subdirectories, most of which are created automatically by the system. In iOS, the app installer creates only a few subdirectories in `~/Library` (such as `Caches` and `Preferences`) and your app is responsible for creating all others.

Table A-1 lists some of the common subdirectories you might find in a `Library` directory in OS X along with the types of files that belong there. You should always use these directories for their intended purposes. For information about the directories your app should be using the most, see "The Library Directory Stores App-Specific Files" (page 22).

Table A-1 Subdirectories of the Library directory

Subdirectory	Directory contents
`Application Support`	Contains all app-specific data and support files. These are the files that your app creates and manages on behalf of the user and can include files that contain user data. By convention, all of these items should be put in a subdirectory whose name matches the bundle identifier of the app. For example, if your app is named MyApp and has the bundle identifier `com.example.MyApp`, you would put your app's user-specific data files and resources in the `~/Library/Application Support/com.example.MyApp/` directory. Your app is responsible for creating this directory as needed. Resources required by the app to run must be placed inside the app bundle itself.
`Assistants`	Contains programs that assist users in configuration or other tasks.
`Audio`	Contains audio plug-ins, loops, and device drivers.
`Autosave Information`	Contains app-specific autosave data.

Subdirectory	Directory contents
`Caches`	Contains cached data that can be regenerated as needed. Apps should never rely on the existence of cache files. Cache files should be placed in a directory whose name matches the bundle identifier of the app. By convention, apps should store cache files in a subdirectory whose name matches the bundle identifier of the app. For example, if your app is named MyApp and has the bundle identifier `com.example.MyApp`, you would put user-specific cache files in the `~/Library/Caches/com.example.MyApp/` directory.
`ColorPickers`	Contains resources for picking colors according to a certain model, such as the HLS (Hue Angle, Saturation, Lightness) picker or RGB picker.
`ColorSync`	Contains ColorSync profiles and scripts.
`Components`	Contains system bundles and extensions.
`Containers`	Contains the home directories for any sandboxed apps. (Available in the user domain only.)
`Contextual Menu Items`	Contains plug-ins for extending system-level contextual menus.
`Cookies`	Contains data files with web browser cookies.
`Developer`	Contains data used by Xcode and other developer tools.
`Dictionaries`	Contains language dictionaries for the spell checker.
`Documentation`	Contains documentation files and Apple Help packages intended for the users and administrators of the computer. (Apple Help packages are located in the `Documentation/Help` directory.) In the local domain, this directory contains the help packages shipped by Apple (excluding developer documentation).
`Extensions`	Contains device drivers and other kernel extensions.
`Favorites`	Contains aliases to frequently accessed folders, files, or websites. (Available in the user domain only.)
`Fonts`	Contains font files for both display and printing.
`Frameworks`	Contains frameworks and shared libraries. The `Frameworks` directory in the system domain is for Apple-provided frameworks only. Developers should install their custom frameworks in either the local or user domain.
`Internet Plug-ins`	Contains plug-ins, libraries, and filters for web-browser content.

Subdirectory	Directory contents
`Keyboards`	Contains keyboard definitions.
`LaunchAgents`	Specifies the agent apps to launch and run for the current user.
`LaunchDaemons`	Specifies the daemons to launch and run as root on the system.
`Logs`	Contains log files for the console and specific system services. Users can also view these logs using the Console app.
`Mail`	Contains the user's mailboxes. (Available in the user domain only.)
`PreferencePanes`	Contains plug-ins for the System Preferences app. Developers should install their custom preference panes in the local domain.
`Preferences`	Contains the user's preferences. You should never create files in this directory yourself. To get or set preference values, you should always use the `NSUserDefaults` class or an equivalent system-provided interface.
`Printers`	In the system and local domains, this directory contains print drivers, PPD plug-ins, and libraries needed to configure printers. In the user domain, this directory contains the user's available printer configurations.
`QuickLook`	Contains QuickLook plug-ins. If your app defines a QuickLook plug-in for viewing custom document types, install it in this directory (user or local domains only).
`QuickTime`	Contains QuickTime components and extensions.
`Screen Savers`	Contains screen saver definitions. See *Screen Saver Framework Reference* for a description of the interfaces used to create screen saver plug-ins.
`Scripting Additions`	Contains scripts and scripting resources that extend the capabilities of AppleScript.
`Sounds`	Contains system alert sounds.
`StartupItems`	(Deprecated) Contains system and third-party scripts and programs to be run at boot time. (See *Daemons and Services Programming Guide* for more information about starting up processes at boot time.)
`Web Server`	Contains web server content. This directory contains the CGI scripts and webpages to be served. (Available in the local domain only.)

File System Details

This appendix includes information about the file systems supported by OS X and iOS.

Supported File Systems

OS X supports a variety of file systems and volume formats, including those listed in Table B-1. Although the primary volume format is HFS Plus, OS X can also boot from a disk formatted with the UFS file system. Future versions of OS X may be bootable with other volume formats as well.

Table B-1 File systems supported by OS X

File System	Description
HFS	Mac OS Standard file system. Standard Macintosh file system for older versions of Mac OS. File systems of this type are treated as read only as of OS X v10.6.
HFS Plus	Mac OS Extended file system. Standard Macintosh file system for OS X.
WebDAV	Used for directly accessing files on the web. For example, iDisk uses WebDAV for accessing files.
UDF	Universal Disk Format. The standard file system for all forms of DVD media (video, ROM, RAM and RW) and some writable CD formats.
FAT	The MS-DOS file system, with 16- and 32-bit variants. Fat 12-bit is not supported.
ExFAT	An interchange format used by digital cameras and other peripherals.
SMB/CIFS	Used for sharing files with Microsoft Windows SMB file servers and clients.
AFP	Apple Filing Protocol. The primary network file system for all versions of Mac OS.
NFS	Network File System. A commonly-used UNIX file sharing standard. OS X supports NFSv2 and NFSv3 over TCP and UDP. OS X 10.7 also supports NFSv4 over TCP.
FTP	A file system wrapper for the standard Internet File Transfer Protocol.
Xsan	Apple's 64-bit cluster file system used in storage area networks.
NTFS	A standard file system for computers running the Windows operating system.

File System	Description
CDDAFS	A file system used to mount audio CDs and present audio tracks on disc to users as AIFF-C encoded files.
ISO 9660	The file system format used by CD-ROMs.

Many file systems require specific separator characters to denote paths. It is best to use the NSURL file construction methods to construct strings rather than doing so manually.

The Finder

Because the Finder is the user's main access to the file system in OS X, it helps to understand a little about how the Finder presents and works with files.

Filename Sorting Rules

The Finder's sort order for file and directory names is based on the Unicode Collation Algorithm (Technical Standard UTS #10) defined by the Unicode Consortium. That standard provides a complete and unambiguous sort ordering for all Unicode characters and is available on the Unicode Consortium website (http://www.unicode.org). The Finder alters the default sorting behavior of this algorithm slightly by taking advantage of some sanctioned alternatives, specifically:

- Punctuation and symbols are significant for sorting.
- Sub-strings of digits are sorted according to their numeric value, as opposed to sorting the actual characters in the number.
- Case is not considered during sorting.

Presentation Rules for Files and Directories

The Finder uses several pieces of information to determine how to present files and directories. The file's bundle bit, type code, creator code, and filename extension all help determine the icon. User settings also play a role. The following steps explain the process used to choose icons for files and directory:

- For files:
    1. The Finder asks Launch Services to provide an appropriate icon. (File icons are nominally provided by the app that defines the appropriate file type. The system apps also provide default icons for many known file types.)
    2. If no icon is available, the finder displays the generic file icon.

- For non-bundled directories:
    1. For specific system directories, the Finder displays a custom icon. These custom icons are typically a generic folder icon overlaid with an image that indicates the purpose of the directory.
    2. For all other directories, the system displays the generic folder icon.
- For bundled directories:
    1. The system presents the directory as a file and does not let the user navigate further down into the directory by default. (Users can still view the contents of the bundle directory by Control-clicking the directory and selecting Show Package Contents from the contextual menu that appears.)
    2. If the bundled directory has the extension `.app` in its filename, the Finder applies the icon associated with the app bundle. If no icon is provided.
    3. For a bundled directory, the Finder looks up the type code, creator code, and filename extension in the Launch Services database and uses that information to locate the appropriate custom icon.
    4. If no custom icon is available for either a file or directory, the Finder displays the default icon appropriate for the given item type.

       The default icon can differ based on whether the item is a document, unbundled directory, app, plug-in, or generic bundle, among others.

File Types and Creator Codes

File type and creator codes are an older way of identifying the type of a file and the app that created it. A file type code (a 32-bit value usually specified as a sequence of four characters) identifies the type of content contained in a file. A creator code (similarly identified using a sequence of four characters) identifies the app that created the file and acts as the primary editor for that specific file. Although these codes are generally deprecated, you may see them in legacy files and apps and in some places in the system.

OS X File System Security

OS X provides file system security policies that limit access to files and directories (including special files and directories such as mount points for volumes, block and character special device files that represent hardware devices, symbolic links, named pipes, UNIX domain sockets, and so on). This appendix explains these policies and how they affect apps.

Security Schemes

OS X provides three file system security schemes: UNIX (BSD) permissions, POSIX access control lists (ACLs), and sandbox entitlements. In addition, the BSD layer provides several per-file flags that override UNIX permissions. These schemes are described in the sections that follow.

In addition, OS X allows admin users to disable ownership and permissions checking for removable volumes on a per-volume basis by choosing Get Info on the volume in Finder, then checking the "Ignore ownership on this volume" checkbox.

These permissions models fit together as follows:

1. If the app's sandbox forbids the requested access, the request is denied. See "Sandbox Entitlements" (page 97) for details.
2. If ownership checking has been disabled for the volume in question by the system administrator (with a checkbox in its Finder Get Info window), the request is granted.
3. If an access control entry exists on the file, it is evaluated and used to determine access rights. See "POSIX ACLs" (page 97) for details.
4. If a file flag prohibits the operation, the operation is denied. See "BSD File Flags" (page 103) for details.
5. Otherwise, if the user ID matches the owner of the file, the "user" permissions (also called "owner" permissions) are used. See "UNIX Permissions" (page 105) for details.
6. Otherwise, if the group ID matches the group for the file, the "group" permissions are used. See "UNIX Permissions" (page 105) for details.
7. Otherwise, the "other" permissions are used. See "UNIX Permissions" (page 105) for details.

Sandbox Entitlements

OS X supports the use of a sandbox to limit an app's ability to access files. These limits override any permissions the app might otherwise have. Sandbox limits are subtractive, not additive. Therefore, the file system permissions represent the maximum access an app might be allowed if its sandbox also permits that access.

POSIX ACLs

Starting with OS X v10.4, the Mach and BSD permissions policies are supplemented by support in the kernel for **ACLs** (access control lists), which are data structures that provide much more detailed control over permissions than does BSD. For example, ACLs allow the system administrator to specify that a specific user can delete a file but cannot write to it. ACLs also provide compatibility with Active Directory and with the SMB/CIFS networks used by the Windows operating system. For more information on ACL support in OS X for different network file systems, see "Network File Systems" (page 109).

An ACL consists of an ordered list of **ACEs** (access control entries), each of which associates a user or group with a set of permissions and specifies whether each permission is allowed or denied. ACEs also include attributes related to inheritance (see "Inheritance of Permissions" (page 100)).

> **Note:** File system ACLs are not related to the ACLs used by keychains, as described in *Keychain Services Programming Guide*.

File System Access Control Policy

You can use file system ACLs to implement more detailed and complex access control policies than are possible using only BSD permissions. They do so by using many more permission bits than the three used by BSD and by implementing both allow and deny associations for each permission for each user or group. Table B-2 shows the permission bits used by ACLs. Compare these to the BSD permission bits shown in Table B-3 (page 106).

Table B-2 File permission bits using ACLs

Bit	File	Directory
read	Open file for read	List directory contents
write	Open file for write	Add a file entry to the directory
execute	Execute file	Search through the directory (to access files or directories within it)
delete	Delete file	Delete directory
append	Append to file	Add subdirectory to directory
delete child	—	Remove a file or subdirectory entry from the directory
read attributes	Read basic attributes	Read basic attributes
write attributes	Write basic attributes	Write basic attributes
read extended	Read extended (named) attributes	Read extended (named) attributes
write extended	Write extended (named) attributes	Write extended (named) attributes
read permissions	Read file permissions (ACL)	Read directory permissions (ACL)
write permissions	Write file permissions (ACL)	Write directory permissions (ACL)

Bit	File	Directory
take ownership	Take ownership	Take ownership

Notice that the right to change permissions is itself controlled by a permission.

ACLs and User IDs

One of the main reasons for implementing ACLs in OS X is to support network file systems such as SMB/CIFS (see "SMB/CIFS" (page 110)). In order to be able to identify users and groups throughout the network, each file or directory must have universally unique identifiers (UUIDs) in addition to the locally-unique UID and GID used by BSD. Each file or directory that has associated ACLs, therefore, has four associated identities, two to support BSD and two to support ACLs:

- user ID (UID)
- group ID (GID)
- owner UUID
- group UUID

Unlike BSD, which specifies three permissions for each file (one for the file's owner, one for members of the file's group, and one for everyone else), an ACL can specify different permissions for each ACE. Another contrast between ACLs and BSD is that, whereas in BSD the file owner must be an individual, in the ACL permission scheme the file owner can be either a user or a group. If a file is owned by a group, its GID (used by BSD) and group UUID are always coherent (that is, there is always a simple, 1:1 mapping between them). However, because BSD does not support the concept of a group as owner of a file, in this case the system assigns a special UID that identifies the file as owned by "not a user" and the owner UUID represents a group. If the file is owned by a single individual, its UID and owner UUID are coherent.

The owner of a file using an ACL has certain irrevocable permissions (read and write permissions) regardless of the contents of the ACL. If the file is owned by an individual, the group UUID associates a group with a file system object and affects the inheritance of certain ACEs (see "Inheritance of Permissions" (page 100)) but does not confer any special permissions on the group.

Evaluating Access Control Lists

Each ACE in an ACL either allows or a denies some set of permissions. It is very important to understand that a deny ACE is not the same as the absence of an allow ACE. Rather, the system evaluates the ACEs in sequence until either all requested permissions are allowed or any requested permission is denied. A request for authorization includes a credential (which identifies the requesting entity) and the permissions required for the operation. OS X v10.4 and later evaluates permissions using the following algorithm (also, see "Inheritance of Permissions" (page 100) for a discussion of inherited permissions):

1. If the requested permissions would change the object, and the file system is read-only or the object is marked as immutable, the operation is denied.
2. If the entity making the request is the root user, the operation is allowed.
3. If the entity making the request is the object's owner, the requestor is given Read Permissions and Write Permissions access. If that is sufficient to satisfy the request, the operation is allowed.
4. If the object has an ACL, the ACEs in the ACL are scanned in order. (Those with deny associations are usually placed before those with allow associations.) Each ACE is evaluated according to the following criteria until either a required permission has been denied, all required permissions have been allowed, or the end of the ACL is reached:
    a. The ACE is checked for applicability. The ACE is not considered applicable if it does not refer to any of the requested permissions. In addition, the requesting entity must be the same as the entity named in the ACE, or the requestor must be a member of a group named in the ACE. (Groups may be nested and an external directory service may be used to resolve group membership.) Non-applicable ACEs are ignored.
    b. If the ACE denies any of the requested permissions, then the request is denied. (Note that Read Permissions and Write Permissions are granted to the object's owner, regardless of whether allowed or denied by ACEs.)
    c. If the ACE allows any of the requested permissions, the system adds this permission to the list of granted permissions. If the granted permissions include all the requested permissions, the request is allowed and the process stops. If the list is not complete, the system goes on to check the next ACE.
5. If the end of the ACL is reached without finding all of the required permissions, and if the object also has BSD permissions, then the system checks the unsatisfied permissions against the BSD permissions. If these are sufficient to grant all required permissions, the request is allowed. If the permission requested has no BSD equivalent (such as "take ownership"), then it is considered still outstanding and the request is denied.
6. If the file system object has no ACL, then permissions are evaluated according to the BSD security policies, as described in "UNIX Permissions" (page 105) and "BSD File Flags" (page 103).

The credential of the requesting entity is equivalent to the effective UID (that is, the EUID) of the program attempting to open or execute a file. The EUID is normally the same as the UID of the user or process that executes the process., but it can differ in special circumstances (involving the setuid bit) as described in "Owner or Root Security Policy" in *Security Overview*.

Inheritance of Permissions

BSD permissions are assigned only on a per-file basis, so that the permissions assigned to a directory do not affect the permissions of a new file or subdirectory created in that directory. Although you can apply the permissions of a directory to enclosed items, doing so is a one-time operation. Any newly created files or subdirectories are not affected—they are created with default permissions.

With ACLs, by contrast, newly created files and subdirectories can inherit permissions from their enclosing directory. Each ACE on a directory can contain any combination of the following inheritance flags:

- Inherited (this ACE was inherited).
- File Inherit (this ACE should be inherited by files created within this directory.
- Directory Inherit (this ACE should be inherited by directories created within this directory).
- Inherit Only (this ACE should not be checked during authorization).
- No Propagate Inherit (this ACE should be inherited only by direct children; that is, the ACE should lose any Directory Inherit or File Inherit bit when inherited).

When it creates a new file, the kernel goes through the entire access control list of the parent directory and copies to the file's ACL any ACEs that are marked for file inheritance. Similarly, when it creates a new subdirectory, the kernel copies to the subdirectory's ACL any ACEs that are marked for directory inheritance.

If a file is copied and pasted into a directory, the kernel replicates the contents of the source file into a new file at the destination. Because it is creating a new file, the system checks the ACL of the parent directory and adds any inherited ACEs to whatever ACEs were in the original file. If a file is moved into a directory, on the other hand, the original file is not replicated and no ACEs are inherited. In this case, the parent directory's ACEs are added to the moved file only if the administrator specifically propagates ACEs from the parent directory through contained files and subdirectories. Similarly, once a file has been created, changing the ACL of the parent directory does not affect the ACL of contained files and subdirectories unless the administrator specifically propagates the change.

In BSD, applying a directory's permissions to enclosed files and subdirectories completely replaces the permissions of the enclosed objects. With ACLs, in contrast, inherited ACEs are added to other ACEs already on the file or directory.

The order in which ACEs are placed in an ACL—and therefore the order in which they are evaluated to determine permissions—is as follows:

1. Explicitly specified deny associations
2. Explicitly specified allow associations
3. Inherited associations, in the same order in which they appeared in the parent

Therefore, any explicitly specified ACEs take precedence over all inherited ACEs. For more information on how ACEs are evaluated, see "Evaluating Access Control Lists" (page 99).

Because ACEs can be inherited, administrators can control the fine-grained permissions of files created in a directory by assigning inheritable ACEs to the directory. Doing so saves the work of assigning ACEs to each file individually. In addition, because ACEs can apply to groups of users, administrators can assign permissions

to groups rather than having to specify permissions for each individual. Applying access security to directories and groups rather than to files and individuals saves administrator time and gives better file system performance in many circumstances.

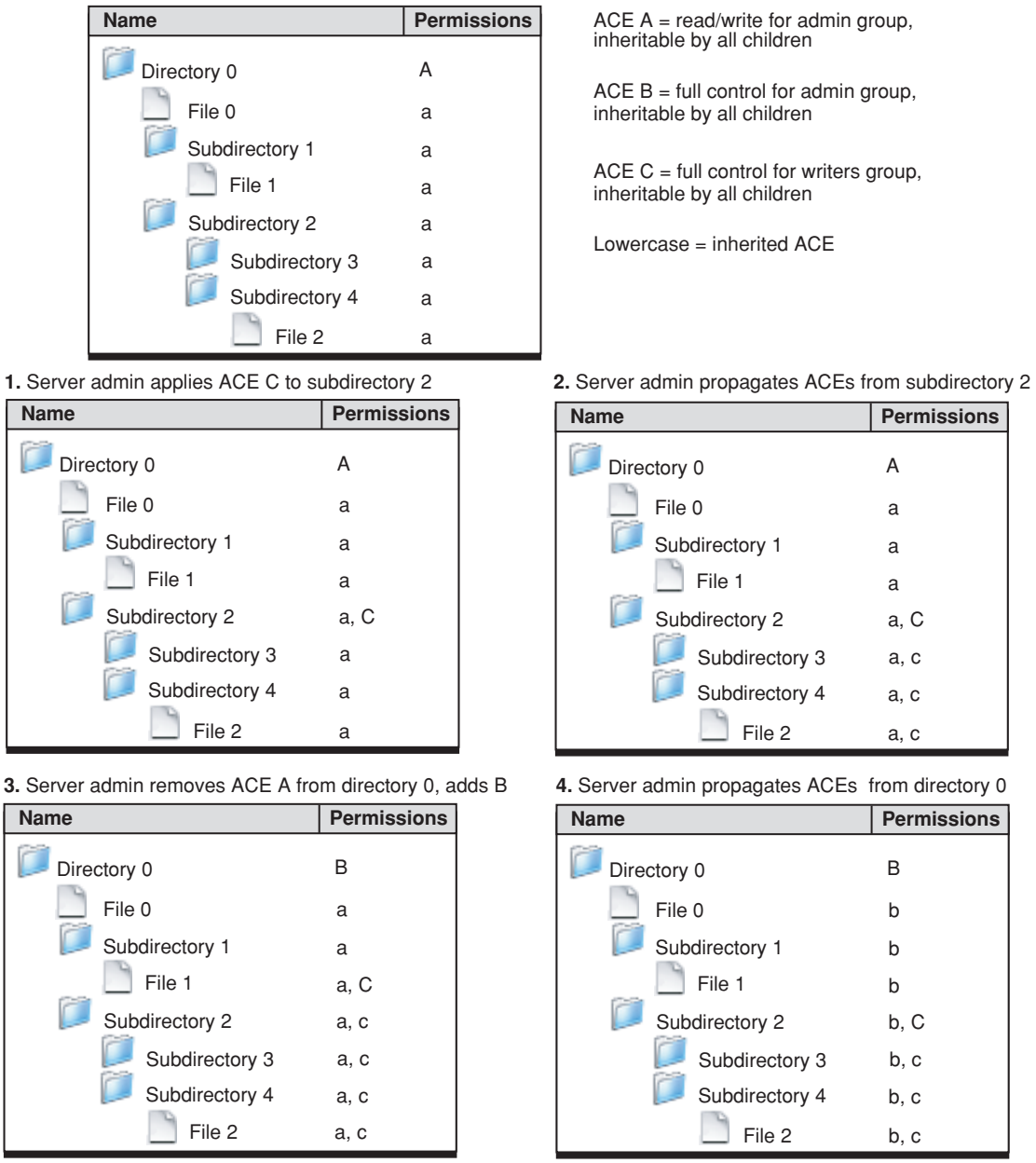
For app programmers, it is important to note that the automatic inheritance of permissions from directories means that it is not necessary for an app to create an ACL for each new file it creates or to maintain inherited ACEs when a file is saved, because the kernel creates the ACL for the file using inherited ACEs. (Note that assignment and inheritance of BSD permissions are not affected by ACLs. If ACLs are not supported, the BSD permissions are used. For more information on the way permissions are evaluated when both ACLs and BSD permissions are set, see "Security Schemes" (page 97).)

In OS X Server v10.4, the server administrator can perform the following operations:

- Copy permissions from a parent directory to all files and directories below it in the hierarchy. This makes permissions uniform in the directory tree and should be used only for BSD permissions.
- Propagate permissions from a parent directory to all files and directories below it in the hierarchy. In this case, explicitly specified ACEs are unchanged and ACEs inherited from them are unchanged. Files and subdirectories inherit ACEs as if they had been newly created in place under the directories that have explicitly specified ACEs, as illustrated in Figure B-1.
- Apply inheritance from a parent directory to a specific directory or file.
- Make inherited ACEs in directories explicit.
- Remove all ACEs from directories and files.
- Enable or disable ACLs on a volume.

The server GUI cannot directly manipulate ACEs of files. There is no GUI in the Finder to set or change ACEs. ACEs can be read and set both on the server and client using the command-line tools `ls` and `chmod`.

Figure B-1 Propagating permissions



BSD File Flags

In addition to the standard UNIX file permissions, OS X supports several BSD file flags provided by the `chflags` API and the related `chflags` command. These flags override the UNIX permissions.

C flag name Command-line name	Hex value	Meaning
UF_NODUMP nodump	0x1	Do not back up the file when using the UNIX dump command. This flag is largely superfluous in OS X. This flag can be changed by either the file's owner or the superuser (root).
UF_IMMUTABLE uchg, uchange, or uimmutable	0x2	File cannot be moved, renamed, or deleted (except by root in single-user mode). This flag can be changed by either the file's owner or the superuser (root).
UF_APPEND uappnd or uappend	0x4	Software can only append to the file, not modify the existing data. This flag can be changed by either the file's owner or the superuser (root).
UF_OPAQUE opaque	0x8	Directory is opaque with respect to union mounts. This means that if a directory in the underlying file system exists and has the same name, its contents are not visible. This flag can be changed by either the file's owner or the superuser (root).
----------	0x10	Reserved.
UF_COMPRESSED (no command equivalent)	0x20	File is compressed at the file system level. This flag can be changed by either the file's owner or the superuser (root).
----------	0x40–0x4000	Reserved.
UF_HIDDEN hidden	0x8000	Hint that the file should be hidden in the GUI. This flag can be changed by either the file's owner or the superuser (root).
SF_ARCHIVED arch or archived	0x10000	File has been archived. This flag can be changed only by the superuser (root).
SF_IMMUTABLE schg, schange, or simmutable	0x20000	File cannot be moved, renamed, or deleted (except by root in single-user mode). This flag can be changed only by the superuser (root).

C flag name Command-line name	Hex value	Meaning
`SF_APPEND` `sappnd` or `sappend`	`0x40000`	Software can only append to the file, not modify the existing data. This flag can be changed only by the superuser (`root`).

Note: To disable a flag with `chflags`, add `no` before the flag name or drop the leading `no`, as appropriate.

UNIX Permissions

Each file system object has a set of UNIX permissions defined by three attributes:

- **UID**, short for user ID. Commonly referred to as the file's owner.
- **GID**, short for group ID.
- **Flags** that include permission bits and other related attributes.

The flags for a file or directory are a 16-bit value that is often represented as a three-digit or four-digit octal value (with the top four or seven bits dropped):

- Bits 12–15: Flags indicating the type of the file. These bits are immutable and are omitted when representing permissions.
- Bits 9–11: Special permissions bits described in Table B-4 (page 106). Usually 0; may be omitted if not set.
- Bits 6–8: Owner rights bits. These bits limit access by any process whose effective user ID (EUID) is equal to the UID of the file or directory).

  These bits have the highest precedence.
- Bits 3–5: Group rights bits. These bits limit access by any process with an effective group ID (EGID) matching the GID of the file or directory.

  These rights do *not* apply to any process whose EUID matches the UID of the file or directory. These bits have lower precedence than the Owner rights, but higher precedence than the Other rights.
- Bits 0–2: Other rights bits. These bits apply to any process that matches neither the UID nor GID of the file or directory.

The Owner, Group, and Other bit sets contain three bits: read, write, execute (`rwx` for short). The effect of these bits differs for files and directories, as shown in Table B-3.

Table B-3 File permission bits in BSD

Bit	File	Directory
read	Can open file for read	Can list directory contents
write	Can open file for write	Can modify directory contents (move, rename, or delete enclosed files or directories)
execute	Can treat file as a program to run	Can search through the directory (to access files or directories inside it)

In addition to the `r`, `w`, and `x` bits, each file system object also has three ancillary permission bits: `setuid`, `setgid`, and `sticky`.

Table B-4 Special file–system permissions bits

Bit	File	Directory
setuid	For a binary executable, when executed, the EUID of the resulting process is set to the file's UID instead of the EUID of the parent process. The RUID of the resulting process is still set to the EUID of the parent process as usual. This flag has no effect for interpreted scripts or non-executable files.	The UID of any file or directory created within the directory is set to the UID of the directory.
setgid	For a binary executable, when executed, the EGID of the resulting process is set to the file's GID instead of the EGID of the parent process. The RGID of the resulting process is still set to the EGID of the parent process as usual. This flag has no effect for interpreted scripts or non-executable files.	The GID of any file or directory created within the directory is set to the GID of the directory.
sticky	No effect in OS X. For portability, you should avoid setting this bit, as it does have an effect in other UNIX variants.	Restricts deletion of enclosed files or directories to the following three users: • The file or directory's owner (EUID = file UID) • `root` (EUID = 0) • The owner of the sticky directory (EUID = directory UID)

For example, if the owner of a binary executable file is the root user and the setuid bit is set, the program always runs with an EUID of 0. Because such a program runs with root privileges when executed by someone other than `root`, it can create a security vulnerability. Therefore, it is important to restrict the creation and use of setuid and setgid programs.

A user can change the permissions only on files owned by that user. Therefore, only the root user can set the setuid bit on a program owned by `root`.

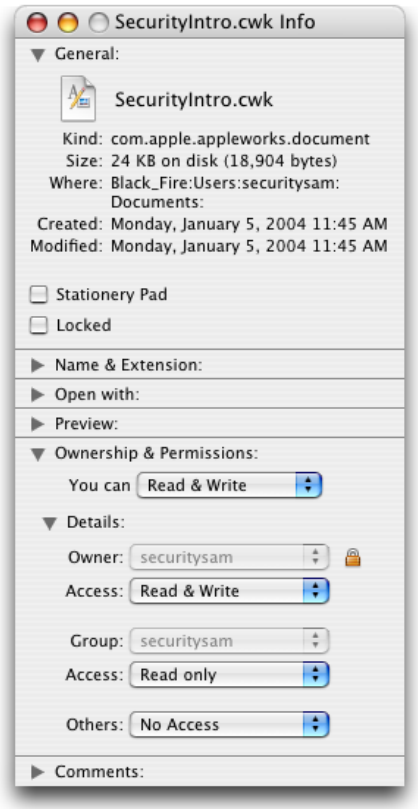
UNIX permissions are visible to users in Terminal and in the Finder. In Terminal, they look like this:

```
$ ls -ld filename dirname
drwxr-xr-x  2 username  groupname  68 Jun 16 13:40 dirname
-rw-r--r--  1 username  groupname   0 Jun 16 13:40 filename
```

The format of the permissions (at left) is described further in "Shell Script Security" in *Shell Scripting Primer*.

In the Finder, UNIX permissions take the form of the Ownership and Permissions information in a file or folder's Info dialog (Figure B-2).

Figure B-2 Ownership and Permissions information



Special Users And Groups

This section lists specific users and groups in OS X that have elevated privileges or the right to obtain elevated privileges.

The Root User

The **root user** owns many of the primary system processes and has unlimited access to the file system objects on the devices attached to the computer. For example, the root user can:

- Read, write, and execute any file
- Copy, move, and rename any file or folder
- Transfer ownership and reset permissions for any file

A major difference between standard BSD permission semantics and the OS X implementation is that in OS X the root user is disabled after system installation. In most cases, it is not necessary for an administrator to run as `root` (see "The Admin Group" (page 109)). You may also assume root power by using the `sudo` utility. Although the `sudo` utility does not require you to enable the root user, you can use it only from the Terminal app; that is, you must have physical access to the machine to use it. See the `sudo` man page for more information on its use.

The root user should not be enabled on user systems. If your app needs to perform operations as the root user, you must use Authorization Services. For more information, see *Authorization Services C Reference* and *Authorization Services Programming Guide* in Security Documentation.

> **Note:** In almost all cases you can run as a member of the `admin` group or use `sudo` rather than enabling the root user. If you absolutely must enable the root user, run the NetInfo Manager utility and authenticate yourself as the local administrator. Then choose Enable Root User from the Security menu. This menu item is enabled only if you are a member of the local `admin` group—a group with special administrative privileges—and you have been previously authenticated in the local domain. Once you've enabled the root user, the password is blank, so you should give the root user a password by selecting Change Root Password from the Security menu. After you've completed the task requiring root access, you should relinquish root user privileges by choosing Disable Root User from the Security menu.

Whereas most user permissions apply across networks, `setuid` and `setgid` are often ignored on network volumes, as is the concept of a root user.

For example, when accessing remote volumes over NFS, by default, the root user is mapped to **nobody**—a special user with very little access. This prevents the root user on one computer from becoming the root user on another computer.

The Wheel Group

There is a special group in BSD called the **wheel group**. Membership in the wheel group confers on users the ability to become the root user by using the `su` utility on the command line. Users who are not in the wheel group can't become the root user, even if they have the correct password.

In OS X v 10.3 and later, the wheel group is not used. Its functions have been assumed by the `admin` group.

The Admin Group

OS X provides the `admin` group in place of the root user. A member of the **admin group** (referred to as an **administrator**) can perform almost all functions the root user can, and can do them using the Finder—that is, without resorting to the command line. The only thing the administrator is prevented from doing is directly adding, modifying, or deleting files in the system domain. An administrator can use special apps such as Installer or Software Update for this purpose, however.

The user who installs OS X on a system becomes automatically the first administrator for the system. Thereafter, this user (or any other administrator) can use Accounts preferences to create accounts on the local system for new users and can grant administrative privileges to any user on the system.

Network File Systems

This section discusses the use of permissions by network file server protocols. OS X supports four network file server protocols:

- **AFP**—Apple Filing Protocol, the principal file-sharing protocol in Mac OS 9 systems, used by AppleShare servers and clients.
- **NFS**—Network File System, the main file-sharing protocol used by UNIX systems.
- **SMB/CIFS**—Server Message Block/Common Internet File System, a file-sharing protocol used on Windows and UNIX systems
- **WebDAV**—Web-based Distributed Authoring and Versioning, an extension of HTTP that allows collaborative file management on the web.

AFP

If the AppleShare client and server both support AFP 3.0, the actual BSD permissions are transported over the connection. If the file or directory on the AFP server has an ACL, the ACL is transported over the connection and the effective permissions are displayed by the Finder. However, enforcement of permissions is done only on the server, not on the client. See "POSIX ACLs" (page 97) for more information on the OS X implementation of ACLs.

If the connection is using AFP 2.x, be aware of the differences in how permissions work:

- BSD supports permissions on files, whereas AFP 2.x does not.
- BSD implements a "best match" permissions policy. If you're the owner, you get the owner permissions. If you're not the owner but you're in the file's group, you get the group permissions. Otherwise you get the other permissions. AFP implements a cumulative permissions policy: your permissions are the union of the permissions you derive from the owner, group, and other permissions. For example, if a folder is writable by the group but not by the owner, AFP permissions let the owner modify the folder but BSD permissions do not.
- BSD interprets the `rwx` bits for folders as shown in Table B-3. AFP permissions define them as "See Files", "See Folders", and "Make Changes". When dealing with an AppleShare 2.x server, the OS X AppleShare client maps between these privilege models. A similar mapping applies when you connect to an OS X server using an AppleShare 2.x client.
- ACLs are not supported by AFP 2.x.

AFP excludes a process having an EUID of 0 (that is, one running as `root`) from accessing any data over the network.

NFS

In general, NFS is not a secure protocol, because most NFS servers trust their clients. That is, if a client says that this file operation is done on behalf of user Bob, the server does the operation on behalf of user Bob. However, if you have root access on the client, you can pretend to be user Bob and access any of Bob's files on the NFS server. To maintain some security, most NFS servers map the root user to a special user, `nobody`, which owns no files or directories. For this reason, if your EUID is 0 you can, in general, access only those files on an NFS server that allow access to "other".

SMB/CIFS

SMB is a networking protocol for file sharing commonly used on Windows networks. CIFS is often used as a synonym for SMB. **Samba** is software that implements an SMB/CIFS server on UNIX. Therefore, this file sharing protocol is variously referred to as SMB, CIFS, SMB/CIFS, Samba, and Windows file sharing.

OS X v10.4 and later implements SMB/CIFS-compatible access control lists (ACLs). Although individual users cannot set or alter ACLs, server administrators can do so. (Administrators can use the SMB server command line to manipulate ACLs, but only if both the client and server are bound to the same Active Directory domain.) However, enforcement of permissions is done only on the server, not on the client. See "POSIX ACLs" (page 97) for more information on the OS X implementation of ACLs.

For OS X v10.3 and earlier, all of the SMB access controls in OS X are implemented on the server, not the client. Consequently, when an OS X user mounts an SMB file server, the volume, directory, or file mounted appears in the Finder to allow read, write, and execute access and to be owned by the user. However, when the user attempts to open a folder or file, the server evaluates the user's access permissions and either allows access or prompts the user for a new user name and password before granting access.

For more information on SMB/CIFS permissions and to learn how to modify their behavior, see the man page for SMB (`man 5 smb.conf`).

WebDAV

The **WebDAV** protocol is an extension to the HTTP protocol that allows users to write and edit web content remotely; that is, over a network connection. The OS X WebDAV file system uses WebDAV and HTTP requests to access resources on a WebDAV-enabled HTTP server as files and directories.

The WebDAV protocol does not support users and groups. Furthermore, a WebDAV client cannot determine access permissions for files and directories on a WebDAV server before attempting to access them. Therefore, the WebDAV file system in OS X sets the user and group IDs to `unknown` for all files and directories and the permissions default to `read`, `write`, and `execute` for everyone: user, group, and other.

When the WebDAV file system sends a request to a WebDAV-enabled HTTP server, the server determines whether authorization is required. If no authorization is required, the server accepts the request. If authorization is required, the server checks for authentication credentials (such as a user name and password) and, if they are present and correct, the server authorizes the client and allows access. If authorization is required and no credentials were sent or the credentials are not correct, the server rejects the request with a challenge for authentication. If the user cannot supply the correct credentials, the WebDAV file system refuses access.

For more information on the protocols used by the WebDAV file system, see the following documents:

- *Hypertext Transfer Protocol–HTTP/1.1* http://www.ietf.org/rfc/rfc2616.txt
- *HTTP Authentication: Basic and Digest Access Authentication* http://www.ietf.org/rfc/rfc2617.txt
- *HTTP Extensions for Distributed Authoring–WEBDAV* http://www.ietf.org/rfc/rfc2518.txt

iOS File System Security

Although the underlying permissions model in iOS is the same as in OS X, in practice, iOS apps are limited by their sandbox such that an app can generally only access files created by that app. (Apps can access certain other files such as address book data and photos, but only through APIs specifically designed for that purpose.)

In addition, if file protection is enabled on an iOS device, apps can choose to prevent access to specific files to when the device is locked. You might do this for files that contain private user data or sensitive information.

As with the keychain, encrypted files in iOS are also encrypted in any backups of the mobile device. In addition to enabling encryption, you can also cause a file to be excluded from appearing in backups entirely.

> **Important:** Any app that supports file protection must be prepared to handle situations in which the app is running but the protected file is unavailable.

For more information about file protection APIs, see the information in "Advanced App Tricks" in *iOS App Programming Guide*.

Document Revision History

This table describes the changes to *File System Programming Guide*.

Date	Notes
2012-03-08	Corrected a statement about where iOS apps can create new directories.
2012-03-01	Added a note in the Overview about changes to file system behavior when you adopt App Sandbox in an OS X app.
2012-01-09	Added information about supporting iCloud. Added iCloud copying, moving, and deleting information. Added information about file coordinators and file presenters. Added permissions information that was previously available in *Security Overview* and added an overview to show how the various permissions models interact.
2011-06-06	New document that describes how to create and manage files, directories, and other content in the file system. This document covers information that was previously available in *File System Overview*, *Low-Level File Management Programming Topics*, and *Application File Management*.

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