Bundle Programming Guide

Developer

Contents

Tables and Listings

Introduction

Bundles are a fundamental technology in OS X and iOS that are used to encapsulate code and resources. Bundles simplify the developer experience by providing known locations for needed resources while alleviating the need to create compound binary files. Instead, bundles use directories and files to provide a more natural type of organization—one that can also be modified easily both during development and after deployment.

To support bundles, both Cocoa and Core Foundation provide programming interfaces for accessing the contents of bundles. Because bundles use an organized structure, it is important that all developers understand the fundamental organizing principles of bundles. This document provides you with the foundation for understanding how bundles work and for how you use them during development to access your resource files.

Organization of This Document

This document contains the following chapters:

- "About Bundles" (page 7) introduces the concept of bundles and packages and how they are used by the system.
- "Bundle Structures" (page 13) describes the structure and contents of the standard bundle types.
- "Accessing a Bundle's Contents" (page 33) shows you how to use the Cocoa and Core Foundation interfaces to get information about a bundle and its contents.
- "Document Packages" (page 48) describes the notion of document packages (which are loosely related to bundles) and how you use them.

See Also

Although the information in this document applies to all types of bundles, if you are working with more specialized types of bundles (such as frameworks and plug-ins), you should also consult the following documents:

- *Framework Programming Guide* provides detailed information about creating and using custom frameworks.
- *Code Loading Programming Topics* provides information about writing plug-ins using the Objective-C language.

- *Plug-in Programming Topics* provides information about writing plug-ins using the C language.

About Bundles

Bundles are a convenient way to deliver software in OS X and iOS. Bundles provide a simplified interface for end users and at the same time provide support for development. This chapter provides an introduction to bundles and discusses the role they play in OS X and iOS.

Bundles and Packages

Although bundles and packages are sometimes referred to interchangeably, they actually represent very distinct concepts:

- A **package** is any directory that the Finder presents to the user as if it were a single file.
- A **bundle** is a directory with a standardized hierarchical structure that holds executable code and the resources used by that code.

Packages provide one of the fundamental abstractions that makes OS X easy to use. If you look at an application or plug-in on your computer, what you are actually looking at is a directory. Inside the package directory are the code and resource files needed to make the application or plug-in run. When you interact with the package directory, however, the Finder treats it like a single file. This behavior prevents casual users from making changes that might adversely affect the contents of the package. For example, it prevents users from rearranging or deleting resources or code modules that might prevent an application from running correctly.

> **Note** Even though packages are treated as opaque files by default, it is still possible for users to view and modify their contents. On the contextual menu for package directories is a Show Package Contents command. Selecting this command displays a new Finder window set to the top level of the package directory. The user can use this window to navigate the package's directory structure and make changes as if it were a regular directory hierarchy.

Whereas packages are there to improve the user experience, bundles are geared more toward helping developers package their code and to helping the operating system access that code. Bundles define the basic structure for organizing the code and resources associated with your software. The presence of this structure also helps facilitate important features such as localization. The exact structure of a bundle depends on whether you are creating an application, framework, or plug-in. It also depends on other factors such as the target platform and the type of plug-in.

The reason bundles and packages are sometimes considered to be interchangeable is that many types of bundles are also packages. For example, applications and loadable bundles are packages because they are usually treated as opaque directories by the system. However, not all bundles are packages and vice versa.

How the System Identifies Bundles and Packages

The Finder considers a directory to be a package if any of the following conditions are true:

- The directory has a known filename extension: `.app`, `.bundle`, `.framework`, `.plugin`, `.kext`, and so on.
- The directory has an extension that some other application claims represents a package type; see "Document Packages" (page 48).
- The directory has its package bit set.

The preferred way to specify a package is to give the package directory a known filename extension. For the most part, Xcode takes care of this for you by providing templates that apply the correct extension. All you have to do is create an Xcode project of the appropriate type.

Most bundles are also packages. For example, applications and plug-ins are typically presented as a single file by the Finder. However, this is not true for all bundle types. In particular, a framework is a type of bundle that is treated as a single unit for the purposes of linking and runtime usage, but framework directories are transparent so that developers can view the header files and other resources they contain.

About Bundle Display Names

Display names give the user some control over how bundles and packages appear in the Finder without breaking clients that rely on them. Whereas a user can rename a file freely, renaming an application or framework might cause related code modules that refer to the application or framework by name to break. Therefore, when the user changes the name of a bundle, the change is superficial only. Rather than change the bundle name in the file system, the Finder associates a separate string (known as the **display name**) with the bundle and displays that string instead.

Display names are for presentation to the user only. You never use display names to open or access directories in your code, but you do use them when displaying the name of the directory to the user. By default, a bundle's display name is the same as the bundle name itself. However, the system may alter the default display name in the following cases:

- If the bundle is an application, the Finder hides the `.app` extension in most cases.
- If the bundle supports localized display names (and the user has not explicitly changed the bundle name), the Finder displays the name that matches the user's current language settings.

Although the Finder hides the `.app` extension for applications most of the time, it may display it to prevent confusion. For example, if the user changes the name of an application and the new name contains another filename extension, the Finder shows the `.app`. extension to make it clear that the bundle is an application. For example, if you were to add the `.mov` extension to the `Chess` application, the Finder would display `Chess.mov.app` to prevent users from thinking `Chess.mov` is a QuickTime file.

For more information about display names and specifying localized bundle names, see *File System Overview*.

The Advantages of Bundles

Bundles provide the following advantages for developers:

- Because bundles are directory hierarchies in the file system, a bundle just contains files. Therefore, you can use all of the same file-based interfaces to open your bundle resources as you do to open other types of files.
- The bundle directory structure makes it easy to support multiple localizations. You can easily add new localized resources or remove unwanted ones.
- Bundles can reside on volumes of many different formats, including multiple fork formats like HFS, HFS+, and AFP, and single-fork formats like UFS, SMB, and NFS.
- Users can install, relocate, and remove bundles simply by dragging them around in the Finder.
- Bundles that are also packages, and are therefore treated as opaque files, are less susceptible to accidental user modifications, such as removal, modification, or renaming of critical resources.
- A bundle can support multiple chip architectures (PowerPC, Intel) and different address space requirements (32-bit/64-bit). It can also support the inclusion of specialized executables (for example, libraries optimized for a particular set of vector instructions).
- Most (but not all) executable code can be bundled. Applications, frameworks (shared libraries), and plug-ins all support the bundle model. Static libraries, dynamic libraries, shell scripts, and UNIX command line tools do not use the bundle structure.
- A bundled application can run directly from a server. No special shared libraries, extensions, and resources need to be installed on the local system.

Types of Bundles

Although all bundles support the same basic features, there are variations in the way you define and create bundles that define their intended usage:

- **Application** - An application bundle manages the code and resources associated with a launchable process. The exact structure of this bundle depends on the platform (iOS or OS X) that you are targeting. For information about the structure of application bundles, see "Application Bundles" (page 13).
- **Frameworks** - A framework bundle manages a dynamic shared library and its associated resources, such as header files. An application can link against one or more frameworks to take advantage of the code they contain. For information about the structure of framework bundles, see "Anatomy of a Framework Bundle" (page 28).
- **Plug-Ins** - OS X supports plug-ins for many system features. Plug-ins are a way for an application to load custom code modules dynamically. The following list identifies some of the key types of plug-ins you might want to develop:
  - **Custom plug-ins** are plug-ins you define for your own purposes; see "Anatomy of a Loadable Bundle" (page 29).
  - **Image Unit plug-ins** add custom image-processing behaviors to the Core Image technology; see *Image Unit Tutorial*.
  - **Interface Builder plug-ins** contain custom objects that you want to integrate into Interface Builder's library window; see *Interface Builder Plug-In Programming Guide*.
  - **Preference Pane plug-ins** define custom preferences that you want to integrate into the System Preferences application; see *Preference Pane Programming Guide*.
  - **Quartz Composer plug-ins** define custom patches for the Quartz Composer application; see *Quartz Composer Custom Patch Programming Guide*.
  - **Quick Look plug-ins** support the display of custom document types using Quick Look; see *Quick Look Programming Guide*.
  - **Spotlight plug-ins** support the indexing of custom document types so that those documents can be searched by the user; see *Spotlight Importer Programming Guide*.
  - **WebKit plug-ins** extend the content types supported by common web browsers; see *WebKit Plug-In Programming Topics*.
  - **Widgets** add new HTML-based applications to Dashboard; see *Dashboard Programming Topics*.

Although document formats can leverage the bundle structure to organize their contents, documents are generally not considered bundles in the purest sense. A document that is implemented as a directory and treated as an opaque type is considered to be a document package, regardless of its internal format. For more information about document packages, see "Document Packages" (page 48).

Creating a Bundle

For the most part, you do not create bundles or packages manually. When you create a new Xcode project (or add a target to an existing project), Xcode automatically creates the required bundle structure when needed. For example, the application, framework, and loadable bundle targets all have associated bundle structures. When you build any of these targets, Xcode automatically creates the corresponding bundle for you.

> **Note** Some Xcode targets (such as shell tools and static libraries) do not result in the creation of a bundle or package. This is normal and there is no need to create bundles specifically for these target types. The resulting binaries generated for those targets are intended to be used as is.

If you use make files (instead of Xcode) to build your projects, there is no magic to creating a bundle. A bundle is just a directory in the file system with a well-defined structure and a specific filename extension added to the end of the bundle directory name. As long as you create the top-level bundle directory and structure the contents of your bundle appropriately, you can access those contents using the programmatic support for accessing bundles. For more information on how to structure your bundle directory, see "Bundle Structures" (page 13).

Programmatic Support for Accessing Bundles

Programs that refer to bundles, or are themselves bundled, can take advantage of interfaces in Cocoa and Core Foundation to access the contents of a bundle. Using these interfaces you can find bundle resources, get information about the bundle's configuration, and load executable code. In Objective-C applications, you use the `NSBundle` class to get and manage bundle information. For C-based applications, you can use the functions associated with the `CFBundleRef` opaque type to manage a bundle.

> **Note** Unlike many other Core Foundation and Cocoa types, `NSBundle` and `CFBundleRef` are not toll-free bridged data types and cannot be used interchangeably. However, you can extract the bundle path information from either object and use it to create the other.

For information about how to use the programmatic support in Cocoa and Core Foundation to access bundles, see "Accessing a Bundle's Contents" (page 33).

Guidelines for Using Bundles

Bundles are the preferred organization mechanism for software in OS X and iOS. The bundle structure lets you group executable code and the resources to support that code in one place and in an organized way. The following guidelines offer some additional advice on how to use bundles:

- Always include an information-property list (`Info.plist`) file in your bundle. Make sure you include the keys recommended for your bundle type. For a list of all keys you can include in this file, see *Runtime Configuration Guidelines*.
- If an application cannot run without a specific resource file, include that file inside the application bundle. Applications should always include all of the images, strings files, localizable resources, and plug-ins that they need to operate. Noncritical resources should similarly be stored inside the application bundle whenever possible but may be placed outside the bundle if needed. For more information about the bundle structure of applications, see "Application Bundles" (page 13).
- If you plan to load C++ code from a bundle, you might want to mark the symbols you plan to load as `extern "C"`. Neither `NSBundle` nor the Core Foundation `CFBundleRef` functions know about C++ name mangling conventions, so marking your symbols this way can make it much easier to identify them later.
- You cannot use the `NSBundle` class to load Code Fragment Manager (CFM) code. If you need to load CFM-based code, you must use the functions for the `CFBundleRef` or `CFPlugInRef` opaque types. You may load CFM-based plugins from a Mach-O executable using this technique.
- You should always use the `NSBundle` class (as opposed to the functions associated with the `CFBundleRef` opaque type) to load any bundle containing Java code.
- When loading bundles containing Objective-C code, you may use either the `NSBundle` class or the functions associated with the `CFBundleRef` opaque type in OS X v10.5 and later, but there are differences in behavior for each. If you use the Core Foundation functions to load a plug-in or other loadable bundle (as opposed to a framework or dynamic shared library), the functions load the bundle privately and bind its symbols immediately; if you use `NSBundle`, the bundle is loaded globally and its symbols are bound lazily. In addition, bundles loaded using the `NSBundle` class cause the generation of `NSBundleDidLoadNotification` notifications, whereas those loaded using the Core Foundation functions do not.

Bundle Structures

Bundle structures can vary depending on the type of the bundle and the target platform. The following sections describe the bundle structures used most commonly in both OS X and iOS.

> **Note** Although bundles are one way of packaging executable code, they are not the only way that is supported. UNIX shell scripts and command-line tools do not use the bundle structure, neither do static and dynamic shared libraries.

Application Bundles

Application bundles are one of the most common types of bundle created by developers. The application bundle stores everything that the application requires for successful operation. Although the specific structure of an application bundle depends on the platform for which you are developing, the way you use the bundle is the same on both platforms. This chapter describes the structure of application bundles in both iOS and OS X.

What Files Go Into an Application Bundle?

Table 2-1 summarizes the types of files you are likely to find inside an application bundle. The exact location of these files varies from platform to platform and some resources may not be supported at all. For examples and more detailed information, see the platform-specific bundle sections in this chapter.

Table 2-1 Types of files in an application bundle

File	Description
`Info.plist` file	(Required) The **information property list** file is a structured file that contains configuration information for the application. The system relies on the presence of this file to identify relevant information about your application and any related files.
Executable	(Required) Every application must have an executable file. This file contains the application's main entry point and any code that was statically linked to the application target.

File	Description
Resource files	Resources are data files that live outside your application's executable file. Resources typically consist of things like images, icons, sounds, nib files, strings files, configuration files, and data files (among others). Resource files can be localized for a particular language or region or shared by all localizations. The placement of resource files in the bundle directory structure depends on whether you are developing an iOS or Mac app.
Other support files	Mac apps can embed additional high-level resources such as private frameworks, plug-ins, document templates, and other custom data resources that are integral to the application. Although you can include custom data resources in your iOS application bundles, you cannot include custom frameworks or plug-ins.

Although most of the resources in an application bundle are optional, this may not always be the case. For example, iOS applications typically require additional image resources for the application's icon and default screen. And although not explicitly required, most Mac apps include a custom icon instead of the default one provided by the system.

Anatomy of an iOS Application Bundle

The project templates provided by Xcode do most of the work necessary for setting up the bundle for your iPhone or iPad application. However, understanding the bundle structure can help you decide where you should place your own custom files. The bundle structure of iOS applications is geared more toward the needs of a mobile device. It uses a relatively flat structure with few extraneous directories in an effort to save disk space and simplify access to the files.

The iOS Application Bundle Structure

A typical iOS application bundle contains the application executable and any resources used by the application (for instance, the application icon, other images, and localized content) in the top-level bundle directory. Listing 2-1 shows the structure of a simple iPhone application called `MyApp`. The only files that are required to be in subdirectories are those that need to be localized; however, you could create additional subdirectories in your own applications to organize resources and other relevant files.

Listing 2-1 Bundle structure of an iOS application

```
MyApp.app
    MyApp
    MyAppIcon.png
```

```
MySearchIcon.png
Info.plist
Default.png
MainWindow.nib
Settings.bundle
MySettingsIcon.png
iTunesArtwork
en.lproj
   MyImage.png
fr.lproj
   MyImage.png
```

Table 2-2 describes the contents of the application shown in Listing 2-1 (page 14). Although the application itself is for demonstration purposes only, many of the files it contains represent specific files that iOS looks for when scanning an application bundle. Your own bundles would include some or all of these files depending on the features you support.

Table 2-2 Contents of a typical iOS application bundle

File	Description
`MyApp`	(Required) The executable file containing your application's code. The name of this file is the same as your application name minus the `.app` extension.
Application icons (`MyAppIcon.png`, `MySearchIcon.png`, and `MySettingsIcon.png`)	(Required/Recommended) Application icons are used at specific times to represent the application. For example, different sizes of the application icon are displayed in the Home screen, in search results, and in the Settings application. Not all of the icons are required but most are recommended. For information about application icons, see "Application Icon and Launch Images" (page 19).
`Info.plist`	(Required) This file contains configuration information for the application, such as its bundle ID, version number, and display name. See "The Information Property List File" (page 17) for further information.

File	Description
Launch images (`Default.png`)	(Recommended) One or more images that show the initial interface of your application in a specific orientation. The system uses one of the provided launch images as a temporary background until your application loads its window and user interface. If your application does not provide any launch images, a black background is displayed while the application launches. For information about application icons, see "Application Icon and Launch Images" (page 19).
`MainWindow.nib`	(Recommended) The application's main nib file contains the default interface objects to load at application launch time. Typically, this nib file contains the application's main window object and an instance of the application delegate object. Other interface objects are then either loaded from additional nib files or created programmatically by the application. (The name of the main nib file can be changed by assigning a different value to the `NSMainNibFile` key in the `Info.plist` file. See "The Information Property List File" (page 17) for further information.)
`Settings.bundle`	The Settings bundle is a special type of plug-in that contains any application-specific preferences that you want to add to the Settings application. This bundle contains property lists and other resource files to configure and display your preferences.
Custom resource files	Non-localized resources are placed at the top level directory and localized resources are placed in language-specific subdirectories of the application bundle. Resources consist of nib files, images, sound files, configuration files, strings files, and any other custom data files you need for your application. For more information about resources, see "Resources in an iOS Application" (page 20).

An iOS application should be internationalized and have a *language* `.lproj` folder for each language it supports. In addition to providing localized versions of your application's custom resources, you can also localize your application icons and launch images by placing files with the same name in your language-specific project directories. Even if you provide localized versions, however, you should always include a default version of these files at the top-level of your application bundle. The default version is used in situations where a specific localization is not available. For more information about localized resources, see "Localized Resources in Bundles" (page 31).

The Information Property List File

Every iOS application must have an information property list (`Info.plist`) file containing the application's confirmation information. When you create a new iOS application project, Xcode creates this file automatically and sets the value of some of the key properties for you. Table 2-3 lists some additional keys that you should set explicitly. (Xcode obscures actual key names by default, so the string displayed by Xcode is also listed in parenthesis where one is used. You can see the real key names for all keys by Control-clicking the Information Property List key in the editor and choosing Show Raw Keys/Values from the contextual menu that appears.)

Table 2-3 Required keys for the `Info.plist` file

Key	Value
`CFBundleDisplayName` (Bundle display name)	The bundle display name is the name displayed underneath the application icon. This value should be localized for all supported languages.
`CFBundleIdentifier` (Bundle identifier)	The bundle identifier string identifies your application to the system. This string must be a uniform type identifier (UTI) that contains only alphanumeric (`A-Z`,`a-z`,`0-9`), hyphen (`–`), and period (`.`) characters. The string should also be in reverse-DNS format. For example, if your company's domain is `Ajax.com` and you create an application named Hello, you could assign the string `com.Ajax.Hello` as your application's bundle identifier. The bundle identifier is used in validating the application signature.
`CFBundleVersion` (Bundle version)	The bundle version string specifies the build version number of the bundle. This value is a monotonically increased string, comprised of one or more period-separated integers. This value cannot be localized.
`CFBundleIconFiles`	An array of strings containing the filenames of the images used for the application's assorted icons. Although technically not required, it is strongly encouraged that you use it. This key is supported in iOS 3.2 and later.
`LSRequiresIPhoneOS` (Application requires iOS environment)	A Boolean value that indicates whether the bundle can run on iOS only. Xcode adds this key automatically and sets its value to true. You should not change the value of this key.

Key	Value
UIRequiredDevice–Capabilities	A key that tells iTunes and the App Store know which device-related features an application requires in order to run. iTunes and the mobile App Store use this list to prevent customers from installing applications on a device that does not support the listed capabilities. The value of this key is either an array or a dictionary. If you use an array, the presence of a given key indicates the corresponding feature is required. If you use a dictionary, you must specify a Boolean value for each key indicating whether the feature is required. In both cases, not including a key indicates that the feature is not required. For a list of keys to include in the dictionary, see *Information Property List Key Reference*. This key is supported in iOS 3.0 and later.

In addition to the keys in the preceding table, Table 2-4 lists some keys that are commonly used by iOS applications. Although these keys are not required, most provide a way to adjust the configuration of your application at launch time. Providing these keys can help ensure that your application is presented appropriately by the system.

Table 2-4 Keys commonly included in the Info.plist file

Para	Para
NSMainNibFile (Main nib file base name)	A string that identifies the name of the application's main nib file. If you want to use a nib file other than the default one created for your project, associate the name of that nib file with this key. The name of the nib file should not include the .nib filename extension.
UIStatusBarStyle	A string that identifies the style of the status bar as the application launches. This value is based on the UIStatusBarStyle constants declared in UIApplication.h header file. The default style is UIStatusBarStyleDefault. The application can change this initial status-bar style when it finishes launching. If you do not specify this key, iOS displays the default status bar.
UIStatusBarHidden	A Boolean value that determines whether the status bar is initially hidden when the application launches. Set it to true to hide the status bar. The default value is false.
UIInterface–Orientation	A string that identifies the initial orientation of the application's user interface. This value is based on the UIInterfaceOrientation constants declared in the UIApplication.h header file. The default style is UIInterfaceOrientationPortrait.

Para	Para
`UIPrerenderedIcon`	A Boolean value that indicates whether the application icon already includes gloss and bevel effects. The default value is `false`. Set it to `true` if you do not want the system to add these effects to your artwork.
`UIRequiresPersistent-WiFi`	A Boolean value that notifies the system that the application uses the Wi-Fi network for communication. Applications that use Wi-Fi for any period of time must set this key to `true`; otherwise, after 30 minutes, the device shuts down Wi-Fi connections to save power. Setting this flag also lets the system know that it should display the network selection dialog when Wi-Fi is available but not currently being used. The default value is `false`. Even if the value of this property is `true`, this key has no effect when the device is idle (that is, screen-locked). During that time, the application is considered inactive and, although it may function on some levels, it has no Wi-Fi connection.
`UILaunchImageFile`	A String containing the base filename used by the application's launch images. If you do not specify this key, the base name is assumed to be the string `Default`.

Application Icon and Launch Images

Application icons and launch images are standard graphics that must be present in every application. Every application must specify an icon to be displayed on the device's Home screen and in the App Store. And an application may specify several different icons for use in different situations. For example, applications can provide a small version of the application icon to use when displaying search results. Launch images provide visual feedback to the user that your application launched.

The image files used to represent icons and launch images must all reside in the root level of your bundle. How you identify these images to the system can vary, but the recommended way to specify your application icons is to use the `CFBundleIconFiles` key. For detailed information about how to specify the icons and launch images in your application, see the discussion of these items in "Advanced App Tricks" in *iOS Application Programming Guide*.

Note In addition to the icons and launch image at the top level of your bundle, you can also include localized versions of those images in your application's language-specific project subdirectories. For more information about including localized resources in your application, see "Localized Resources in Bundles" (page 31).

Resources in an iOS Application

In an iOS application, nonlocalized resources are located at the top-level of the bundle directory, along with the application's executable file and the `Info.plist` file. Most iOS applications have at least a few files at this level, including the application's icon, launch image, and one or more nib files. Although you should place most nonlocalized resources in this top-level directory, you can also create subdirectories to organize your resource files. Localized resources must be placed in one or more language-specific subdirectories, which are discussed in more detail in "Localized Resources in Bundles" (page 31).

Listing 2-2 shows a fictional application that includes both localized and nonlocalized resources. The nonlocalized resources include `Hand.png`, `MainWindow.nib`, `MyAppViewController.nib`, and the contents of the `WaterSounds` directory. The localized resources include everything in the `en.lproj` and `jp.lproj` directories.

Listing 2-2 An iOS application with localized and nonlocalized resources

```
MyApp.app/
    Info.plist
    MyApp
    Default.png
    Icon.png
    Hand.png
    MainWindow.nib
    MyAppViewController.nib
    WaterSounds/
        Water1.aiff
        Water2.aiff
    en.lproj/
        CustomView.nib
        bird.png
        Bye.txt
        Localizable.strings
    jp.lproj/
```

```
        CustomView.nib
        bird.png
        Bye.txt
        Localizable.strings
```

For information about finding resource files in your application bundle, see "Accessing a Bundle's Contents" (page 33). For information about how to load resource files and use them in your program, see *Resource Programming Guide*.

Anatomy of an OS X Application Bundle

The project templates provided by Xcode do most of the work necessary for setting up your Mac app bundle during development. However, understanding the bundle structure can help you decide where you should place your own custom files. OS X bundles use a highly organized structure to make it easier for the bundle-loading code to find resources and other important files in the bundle. The hierarchical nature also helps the system distinguish code bundles such as applications from the directory packages used by other applications to implement document types.

The Structure of an OS X Application Bundle

The basic structure of a Mac app bundle is very simple. At the top-level of the bundle is a directory named `Contents`. This directory contains everything, including the resources, executable code, private frameworks, private plug-ins, and support files needed by the application. While the `Contents` directory might seem superfluous, it identifies the bundle as a modern-style bundle and separates it from document and legacy bundle types found in earlier versions of Mac OS.

Listing 2-3 shows the high-level structure of a typical application bundle, including the immediate files and directories you are most likely to find inside the `Contents` directory. This structure represents the core of every Mac app.

Listing 2-3 The basic structure of a Mac app

```
MyApp.app/
   Contents/
      Info.plist
      MacOS/
      Resources/
```

Table 2-5 lists some of the directories that you might find inside the `Contents` directory, along with the purpose of each one. This list is not exhaustive but merely represents the directories in common usage.

Table 2-5 Subdirectories of the `Contents` directory

Directory	Description
`MacOS`	(Required) Contains the application's standalone executable code. Typically, this directory contains only one binary file with your application's main entry point and statically linked code. However, you may put other standalone executables (such as command-line tools) in this directory as well.
`Resources`	Contains all of the application's resource files. This contents of this directory are further organized to distinguish between localized and nonlocalized resources. For more information about the structure of this directory, see "The Resources Directory" (page 25)
`Frameworks`	Contains any private shared libraries and frameworks used by the executable. The frameworks in this directory are revision-locked to the application and cannot be superseded by any other, even newer, versions that may be available to the operating system. In other words, the frameworks included in this directory take precedence over any other similarly named frameworks found in other parts of the operating system. For information on how to add private frameworks to your application bundle, see *Framework Programming Guide*.
`PlugIns`	Contains loadable bundles that extend the basic features of your application. You use this directory to include code modules that must be loaded into your application's process space in order to be used. You would not use this directory to store standalone executables.
`SharedSupport`	Contains additional non-critical resources that do not impact the ability of the application to run. You might use this directory to include things like document templates, clip art, and tutorials that your application expects to be present but that do not affect the ability of your application to run.

Application bundles have evolved significantly over the years but the overall goal has been the same. The bundle organization makes it easier for the application to find its resources while making it harder for users to interfere with those resources. Because the Finder treats most bundles as opaque entities, it is difficult for casual users to move or delete the resources an application might need.

The Information Property List File

For the Finder to recognize an application bundle as such, you need to include an information property list (`Info.plist`) file. This file contains XML property-list data that identifies the configuration of your bundle. For a minimal bundle, this file would contain very little information, most likely just the name and identifier of the bundle. For more complex bundles, the `Info.plist` file includes much more information.

> **Important** Bundle resources are located using a case-sensitive search. Therefore, the name of your information property list file must start with a capital "I".

Table 2-6 lists the keys that you should always include in your `Info.plist` file. Xcode provides all of these keys automatically when you create a new project. (Xcode obscures actual key names by default, so the string displayed by Xcode is also listed in parenthesis. You can see the real key names for all keys by Control-clicking the Information Property List key in the editor and choosing Show Raw Keys/Values from the contextual menu that appears.)

Table 2-6 Expected keys in the `Info.plist` file

Key	Description
`CFBundleName` (Bundle name)	The short name for the bundle. The value for this key is usually the name of your application. Xcode sets the value of this key by default when you create a new project.
`CFBundleDisplayName` (Bundle display name)	The localized version of your application name. You typically include a localized value for this key in an `InfoPlist.strings` files in each of your language-specific resource directories.
`CFBundleIdentifier` (Bundle identifier)	The string that identifies your application to the system. This string must be a uniform type identifier (UTI) that contains only alphanumeric (`A-Z`,`a-z`,`0-9`), hyphen (`-`), and period (`.`) characters. The string should also be in reverse-DNS format. For example, if your company's domain is `Ajax.com` and you create an application named Hello, you could assign the string `com.Ajax.Hello` as your application's bundle identifier. The bundle identifier is used in validating the application signature.
`CFBundleVersion` (Bundle version)	The string that specifies the build version number of the bundle. This value is a monotonically increased string, comprised of one or more period-separated integers. This value can correspond to either released or unreleased versions of the application. This value cannot be localized.
`CFBundlePackageType` (Bundle OS Type code)	The type of bundle this is. For applications, the value of this key is always the four-character string `APPL`.

Key	Description
`CFBundleSignature` (Bundle creator OS Type code)	The creator code for the bundle. This is a four-character string that is specific to the bundle. For example, the signature for the TextEdit application is `ttxt`.
`CFBundleExecutable` (Executable file)	The name of the main executable file. This is the code that is executed when the user launches your application. Xcode typically sets the value of this key automatically at build time.

Table 2-7 lists the keys that you should also consider including in your `Info.plist` file.

Table 2-7 Recommended keys for the `Info.plist` file

Key	Description
`CFBundleDocumentTypes` (Document types)	The document types supported by the application. This type consists of an array of dictionaries, each of which provides information about a specific document type.
`CFBundleShortVersion-String` (Bundle versions string, short)	The release version of the application. The value of this key is a string comprised of three period-separated integers.
`LSMinimumSystemVersion` (Minimum system version)	The minimum version of OS X required for this application to run. The value for this key is a string of the form *n*.*n*.*n* where each *n* is a number representing either the major or minor version number of OS X that is required. For example, the value 10.1.5 would represent OS X v10.1.5.
`NSHumanReadableCopyright` (Copyright (human-readable))	The copyright notice for the application. This is a human readable string and can be localized by including the key in an `InfoPlist.strings` file in your language-specific project directories.
`NSMainNibFile` (Main nib file base name)	The nib file to load when the application is launched (without the `.nib` filename extension). The main nib file is an Interface Builder archive containing the objects (main window, application delegate, and so on) needed at launch time.
`NSPrincipalClass` (Principal class)	The entry point for dynamically loaded Objective-C code. For an application bundle, this is almost always the `NSApplication` class or a custom subclass.

The exact information you put into your `Info.plist` file is dependent on your bundle's needs and can be localized as necessary. For more information on this file, see *Runtime Configuration Guidelines*.

The Resources Directory

The `Resources` directory is where you put all of your images, sounds, nib files, string resources, icon files, data files, and configuration files among others. The contents of this directory are further subdivided into areas where you can store localized and nonlocalized resource files. Non-localized resources reside at the top level of the `Resources` directory itself or in a custom subdirectory that you define. Localized resources reside in separate subdirectories called language-specific project directories, which are named to coincide with the specific localization.

The best way to see how the `Resources` directory is organized is to look at an example. Listing 2-4 shows a fictional application that includes both localized and nonlocalized resources. The nonlocalized resources include `Hand.tiff`, `MyApp.icns` and the contents of the `WaterSounds` directory. The localized resources include everything in the `en.lproj` and `jp.lproj` directories or their subdirectories.

Listing 2-4 A Mac app with localized and nonlocalized resources

```
MyApp.app/
   Contents/
      Info.plist
      MacOS/
         MyApp
      Resources/
         Hand.tiff
         MyApp.icns
         WaterSounds/
            Water1.aiff
            Water2.aiff
         en.lproj/
            MyApp.nib
            bird.tiff
            Bye.txt
            InfoPlist.strings
            Localizable.strings
            CitySounds/
               city1.aiff
```

```
            city2.aiff
      jp.lproj/
         MyApp.nib
         bird.tiff
         Bye.txt
         InfoPlist.strings
         Localizable.strings
         CitySounds/
            city1.aiff
            city2.aiff
```

Each of your language-specific project directories should contain a copy of the same set of resource files, and the name for any single resource file must be the same across all localizations. In other words, only the content for a given file should change from one localization to another. When you request a resource file in your code, you specify only the name of the file you want. The bundle-loading code uses the current language preferences of the user to decide which directories to search for the file you requested.

For information about finding resource files in your application bundle, see "Accessing a Bundle's Contents" (page 33). For information about how to load resource files and use them in your program, see *Resource Programming Guide*.

The Application Icon File

One special resource that belongs in your top-level `Resources` directory is your application icon file. By convention, this file takes the name of the bundle and an extension of `.icns`; the image format can be any supported type, but if no extension is specified, the system assumes `.icns`.

Localizing the Information Property List

Because some of the keys in an application's `Info.plist` file contain user-visible strings, OS X provides a mechanism for specifying localized versions of those strings. Inside each language-specific project directory, you can include an `InfoPlist.strings` file that specifies the appropriate localizations. This file is a strings file (not a property list) whose entries consist of the `Info.plist` key you want to localize and the appropriate translation. For example, in the TextEdit application, the German localization of this file contains the following strings:

```
CFBundleDisplayName = "TextEdit";
NSHumanReadableCopyright = "Copyright © 1995–2009 Apple Inc.\nAlle Rechte
vorbehalten.";
```

Creating an Application Bundle

The simplest way to create an application bundle is using Xcode. All new application projects include an appropriately configured application **target**, which defines the rules needed to build an application bundle, including which source files to compile, which resource files to copy to the bundle, and so on. New projects also include a preconfigured `Info.plist` file and typically several other files to help you get started quickly. You can add any custom files as needed using the project window and configure those files using the Info or Inspector windows. For example, you might use the Info window to specify custom locations for resource files inside your bundle.

For information on how to configure targets in Xcode, see *Xcode Build System Guide* .

Framework Bundles

A framework is a hierarchical directory that encapsulates a dynamic shared library and the resource files needed to support that library. Frameworks provide some advantages over the typical dynamic shared library in that they provide a single location for all of the framework's related resources. For example, most frameworks include the header files that define the symbols exported by the framework. Grouping these files with the shared library and its resources makes it easier to install and uninstall the framework and to locate the framework's resources.

Just like a dynamic shared library, frameworks provide a way to factor out commonly used code into a central location that can be shared by multiple applications. Only one copy of a framework's code and resources reside in-memory at any given time, regardless of how many processes are using those resources. Applications that link against the framework then share the memory containing the framework. This behavior reduces the memory footprint of the system and helps improve performance.

> **Note** Only the code and read-only resources of a framework are shared. If a framework defines writable variables, each application gets its own copy of those variables to prevent it from affecting other applications.

Although you can create frameworks of your own, most developers' experience with frameworks comes from including them in their projects. Frameworks are how OS X delivers many key features to your application. The publicly available frameworks provided by OS X are located in the `/System/Library/Frameworks` directory. In iOS, the public frameworks are located in the `System/Library/Frameworks` directory of the appropriate iOS SDK directory. For information about adding frameworks to your Xcode projects, see *Xcode Build System Guide* .

Note The creation of custom frameworks is not supported in iOS.

For more detailed information about frameworks and framework bundles, see *Framework Programming Guide* .

Anatomy of a Framework Bundle

The structure of framework bundles differs from that used by applications and plug-ins. The structure for frameworks is based on a bundle format that predates OS X and supports the inclusion of multiple versions of the framework's code and resources in the framework bundle. This type of bundle is known as a **versioned bundle**. Supporting multiple versions of a framework allows older applications to continue running even as the framework shared library continues to evolve. The bundle's `Versions` subdirectory contains the individual framework revisions while symbolic links at the top of the bundle directory point to the latest revision.

The system identifies a framework bundle by the `.framework` extension on its directory name. The system also uses the `Info.plist` file inside the framework's `Resources` directory to gather information about the configuration of the framework. Listing 2-5 shows the basic structure of a framework bundle. The arrows (–>) in the listing indicate symbolic links to specific files and subdirectories. These symbolic links provide convenient access to the latest version of the framework.

Listing 2-5 A simple framework bundle

```
MyFramework.framework/
   MyFramework  -> Versions/Current/MyFramework
   Resources    -> Versions/Current/Resources
   Versions/
      A/
         MyFramework
         Headers/
            MyHeader.h
         Resources/
            English.lproj/
               InfoPlist.strings
            Info.plist
      Current  -> A
```

Frameworks are not required to include a `Headers` directory but doing so allows you to include the header files that define the framework's exported symbols. Frameworks can store other resource files in both standard and custom directories.

Creating a Framework Bundle

If you are developing software for OS X, you can create your own custom frameworks and use them privately or make them available for other applications to use. You can create a new framework using a separate Xcode project or by adding a framework target to an existing project.

For information about how to create a framework, see *Framework Programming Guide*.

Loadable Bundles

Plug-ins and other types of loadable bundles provide a way for you to extend the behavior of an application dynamically. A loadable bundle consists of executable code and any resources needed to support that code stored in a bundle directory. You can use loadable bundles to load code lazily into your application or to allow other developers to extend the basic behavior of your application.

Note The creation and use of loadable bundles is not supported in iOS.

Anatomy of a Loadable Bundle

Loadable bundles are based on the same structure as application bundles. At the top-level of the bundle is a single `Contents` directory. Inside this directory are several subdirectories for storing executable code and resources. The `Contents` directory also contains the bundle's `Info.plist` file with information about the bundle's configuration.

Unlike the executable of an application, loadable bundles generally do not have a `main` function as their main entry point. Instead, the application that loads the bundle is responsible for defining the expected entry point. For example, a bundle could be expected to define a function with a specific name or it could be expected to include information in its `Info.plist` file identifying a specific function or class to use. This choice is left to the application developer who defines the format of the loadable bundle.

Listing 2-6 shows the layout of a loadable bundle. The top-level directory of a loadable bundle can have any extension, but common extensions include `.bundle` and `.plugin`. OS X always treats bundles with those extensions as packages, hiding their contents by default.

Listing 2-6 A simple loadable bundle

```
MyLoadableBundle.bundle
   Contents/
      Info.plist
      MacOS/
         MyLoadableBundle
      Resources/
         Lizard.jpg
         MyLoadableBundle.icns
         en.lproj/
            MyLoadableBundle.nib
            InfoPlist.strings
         jp.lproj/
            MyLoadableBundle.nib
            InfoPlist.strings
```

In addition to the `MacOS` and `Resources` directories, loadable bundles may contain additional directories such as `Frameworks`, `PlugIns`, `SharedFrameworks`, and `SharedSupport`—all the features supported by full-fledged application packages.

The basic structure of a loadable bundle is the same regardless of which language that bundle uses in its implementation. For more information about the structure of loadable bundles, see *Code Loading Programming Topics* .

Creating a Loadable Bundle

If you are developing software for OS X, you can create your own custom loadable bundles and incorporate them into your applications. If other applications export a plug-in API, you can also develop bundles targeted at those APIs. Xcode includes template projects for implementing bundles using either C or Objective-C, depending on the intended target application.

For more information about how to design loadable bundles using Objective-C, see *Code Loading Programming Topics* . For information about how to design loadable bundles using the C language, see *Plug-in Programming Topics* .

Localized Resources in Bundles

Within the `Resources` directory of an OS X bundle (or the top-level directory of an iOS application bundle), you can create one or more language-specific project subdirectories to store language- and region-specific resources. The name of each directory is based on the language and region of the desired localization followed by the `.lproj` extension. To specify the language and region, you use the following format:

language`_`*region*`.lproj`

The *language* portion of the directory name is a two-letter code that conforms to the ISO 639 conventions. The *region* portion is also a two-letter code but it conforms to the ISO 3166 conventions for designating specific regions. Although the region portion of the directory name is entirely optional, it can be a useful way to tune your localizations for specific parts of the world. For example, you could use a single `en.lproj` directory to support all English speaking nations. However, providing separate localizations for Great Britain (`en_GB.lproj`), Australia (`en_AU.lproj`), and the United States (`en_US.lproj`) lets you tailor your content for each of those countries.

> **Note** For backwards compatibility, the `NSBundle` class and `CFBundleRef` functions also support human-readable directory names for several common languages, including `English.lproj`, `German.lproj`, `Japanese.lproj`, and others. Although the human-readable names are supported, the ISO names are preferred.

If most of your resource files are the same for all regions of a given language, you can combine a language-only resource directory with one or more region-specific directories. Providing both types of directories alleviates the need to duplicate every resource file for each region you support. Instead, you can customize only the subset of files that are needed for a particular region. When looking for resources in your bundle, the bundle-loading code looks first in any region-specific directories, followed by the language-specific directory. And if neither localized directory contains the resource, the bundle-loading code looks for an appropriate nonlocalized resource.

Listing 2-7 shows the potential structure of a Mac app that contains both language- and region-specific resource files. (In an iOS application, the contents of the `Resources` directory would be at the top-level of the bundle directory.) Notice that the region-specific directories contain only a subset of the files in the `en.lproj` directory. If a region-specific version of a resource is not found, the bundle looks in the language-specific directory (in this case `en.lproj`) for the resource. The language-specific directory should always contain a complete copy of any language-specific resource files.

Listing 2-7 A bundle with localized resources

```
Resources/
```

```
MyApp.icns
en_GB.lproj/
   MyApp.nib
   bird.tiff
   Localizable.strings
en_US.lproj/
   MyApp.nib
   Localizable.strings
en.lproj/
   MyApp.nib
   bird.tiff
   Bye.txt
   house.jpg
   InfoPlist.strings
   Localizable.strings
   CitySounds/
      city1.aiff
      city2.aiff
```

For more information on language codes and the process for localizing resources, see *Internationalization Programming Topics*.

Accessing a Bundle's Contents

When writing bundle-based code, you never use string constants to refer to the location of files in your bundle. Instead, you use the `NSBundle` class or `CFBundleRef` opaque type to obtain the path to the file you want. The reason is that the path to the desired file can vary depending on the user's native language and the bundle's supported localizations. By letting the bundle determine the location of the file, you are always assured of loading the correct file.

This chapter shows you how to use the `NSBundle` class and `CFBundleRef` opaque type to locate files and obtain other information about your bundle. Although these types are not directly interchangeable, they provide comparable features. And at least in Objective-C applications, you can use whichever type suits your needs.

Locating and Opening Bundles

Before you can access a bundle's resources, you must first obtain an appropriate `NSBundle` object or `CFBundleRef` opaque type. The following sections outline the different ways you can get a reference to one of these types.

Getting the Main Bundle

The main bundle is the bundle that contains the code and resources for the running application. If you are an application developer, this is the most commonly used bundle. The main bundle is also the easiest to retrieve because it does not require you to provide any information.

To get the main bundle in a Cocoa application, call the `mainBundle` class method of the `NSBundle` class, as shown in Listing 3-1.

Listing 3-1 Getting a reference to the main bundle using Cocoa

```
NSBundle* mainBundle;

// Get the main bundle for the app.
mainBundle = [NSBundle mainBundle];
```

If you are writing a C-based application, you can use the `CFBundleGetMainBundle` function to retrieve the main bundle for your application, as shown in Listing 3-2.

Listing 3-2 Getting a reference to the main bundle using Core Foundation

```
CFBundleRef mainBundle;


// Get the main bundle for the app
mainBundle = CFBundleGetMainBundle();
```

When getting the main bundle, it is still a good idea to make sure the value you get back represents a valid bundle. When retrieving the main bundle from any application, the returned value might be `NULL` in the following situations:

- If a program is not bundled, attempting to get the main bundle might return a `NULL` value. The bundle code may try to create a main bundle to represent your program's contents, but doing so is not possible in all cases.
- If the agent that launched the program did not specify the full path to the program's executable in the `argv` parameters, the main bundle value might be `NULL`. Bundles rely on either the path to the executable being in `argv[0]` or the presence of the executable's path in the `PATH` environment variable. If neither of these is present, the bundle routines might not be able to find the main bundle directory. Programs launched by `xinetd` often experience this problem when `xinetd` changes the current directory to `/`.

Getting Bundles by Path

If you want to access a bundle other than the main bundle, you can create an appropriate bundle object if you know the path to the bundle directory. Creating a bundle by path is useful in situations where you are defining frameworks or other loadable bundles and know in advance where those bundles will be located.

To obtain the bundle at a specific path using Cocoa, call the `bundleWithPath:` class method of the `NSBundle` class. (You can also use the `initWithPath:` instance method to initialize a new bundle object.) This method takes a string parameter representing the full path to the bundle directory. Listing 3-3 shows an example that accesses a bundle in a local directory.

Listing 3-3 Locating a Cocoa bundle using its path

```
NSBundle* myBundle;


// Obtain a reference to a loadable bundle.
```

```
myBundle = [NSBundle bundleWithPath:@"/Library/MyBundle.bundle"];
```

To obtain the bundle at a specific path using Core Foundation, call the `CFBundleCreate` function. When specifying the path location in Core Foundation, you must do so using a `CFURLRef` type. Listing 3-4 shows an example that takes the fixed directory from the preceding example, converts it to a URL, and uses that URL to access the bundle.

Listing 3-4 Locating a Core Foundation bundle using its path

```
CFURLRef bundleURL;
CFBundleRef myBundle;

// Make a CFURLRef from the CFString representation of the
// bundle's path.
bundleURL = CFURLCreateWithFileSystemPath(
                kCFAllocatorDefault,
                CFSTR("/Library/MyBundle.bundle"),
                kCFURLPOSIXPathStyle,
                true );

// Make a bundle instance using the URLRef.
myBundle = CFBundleCreate( kCFAllocatorDefault, bundleURL );

// You can release the URL now.
CFRelease( bundleURL );

// Use the bundle...

// Release the bundle when done.
CFRelease( myBundle );
```

Getting Bundles in Known Directories

Even if you do not know the exact path to a bundle, you can still search for it in some known location. For example, an application with a `PlugIns` directory might want to get a list of all the bundles in that directory. Once you have the path to the directory, you can use the appropriate routines to iterate that directory and return any bundles.

The simplest way to find all of the bundles in a specific directory is to use the `CFBundleCreateBundlesFromDirectory` function. This function returns new `CFBundleRef` types for all of the bundles in a given directory. Listing 3-5 shows how you would use this function to retrieve all of the plug-ins in the application's `PlugIns` directory.

Listing 3-5 Obtaining bundle references for a set of plug-ins

```
CFBundleRef mainBundle = CFBundleGetMainBundle();
CFURLRef plugInsURL;
CFArrayRef bundleArray;

// Get the URL to the application's PlugIns directory.
plugInsURL = CFBundleCopyBuiltInPlugInsURL(mainBundle);

// Get the bundle objects for the application's plug-ins.
bundleArray = CFBundleCreateBundlesFromDirectory( kCFAllocatorDefault,
                    plugInsURL, NULL );

// Release the CF objects when done with them.
CFRelease( plugInsURL );
CFRelease( bundleArray );
```

Getting Bundles by Identifier

Locating bundles using a bundle identifier is an efficient way to locate bundles that were previously loaded into memory. A bundle identifier is the string assigned to the `CFBundleIdentifier` key in the bundle's `Info.plist` file. This string is typically formatted using reverse-DNS notation so as to prevent name space conflicts with developers in other companies. For example, a Finder plug-in from Apple might use the string `com.apple.Finder.MyGetInfoPlugin` as its bundle identifier. Rather than passing a pointer to a bundle object around your code, clients that need a reference to a bundle can simply use the bundle identifier to retrieve it.

To retrieve a bundle using a bundle identifier in Cocoa, call the `bundleWithIdentifier:` class method of the `NSBundle` class, as shown in Listing 3-6.

Listing 3-6 Locating a bundle using its identifier in Cocoa

```
NSBundle* myBundle = [NSBundle bundleWithIdentifier:@"com.apple.myPlugin"];
```

Listing 3-7 shows how to retrieve a bundle using its bundle identifier in Core Foundation.

Listing 3-7 Locating a bundle using its identifier in Core Foundation

```
CFBundleRef requestedBundle;

 // Look for a bundle using its identifier
 requestedBundle = CFBundleGetBundleWithIdentifier(
        CFSTR("com.apple.Finder.MyGetInfoPlugIn") );
```

Remember that you can only use a bundle identifier to locate a bundle that has already been opened. For example, you could use this technique to open the main bundle and bundles for all statically linked frameworks. You could not use this technique to get a reference to a plug-in that had not yet been loaded.

Searching for Related Bundles

If you are writing a Cocoa application, you can obtain a list of bundles related to the application by calling the `allBundles` and `allFrameworks` class methods of `NSBundle`. These methods create an array of `NSBundle` objects corresponding to the bundles or frameworks currently in use by your application. You can use these methods as convenience functions rather than maintain a collection of loaded bundles yourself.

The `bundleForClass:` class method is another way get related bundle information in a Cocoa application. This method returns the bundle in which a particular class is defined. Again, this method is mostly for convenience so that you do not have to retain a pointer to an `NSBundle` object that you may use only occasionally.

Getting References to Bundle Resources

If you have a reference to a bundle object, you can use that object to determine the location of resources inside the bundle. Cocoa and Core Foundation both provide different ways of locating resources inside a bundle. In addition, you should understand how those frameworks look for resource files within your bundle so as to make sure you put files in the right places at build time.

The Bundle Search Pattern

As long as you use an `NSBundle` object or a `CFBundleRef` opaque type to locate resources, your bundle code need never concern itself with how resources are retrieved from a bundle. Both `NSBundle` and `CFBundleRef` retrieve the appropriate language-specific resource automatically based on the available user settings and bundle configuration. However, you still have to put all those language-specific resources into your bundle, so knowing how they are retrieved is important.

The bundle programming interfaces follow a specific search algorithm to locate resources within the bundle. Global resources have the highest priority, followed by region- and language-specific resources. When considering region- and language-specific resources, the algorithm takes into account both the settings for the current user and development region information in the bundle's `Info.plist` file. The following list shows the order in which resources are searched:

1. Global (nonlocalized) resources
2. Region-specific resources (based on the user's region preferences)
3. Language-specific resources (based on the user's language preferences)
4. Development language of the bundle (as specified by the `CFBundleDevelopmentRegion` in the bundle's `Info.plist` file.)

Important The bundle interfaces consider case when searching for resource files in the bundle directory. This case-sensitive search occurs even on file systems (such as HFS+) that are not case sensitive when it comes to file names.

Because global resources take precedence over language-specific resources, there should *never* be both a global and localized version of a given resource. If a global version of a resource exists, language-specific versions of the same resource are never returned. The reason for this precedence is performance. If localized resources were searched first, the bundle routines might search needlessly in several localized resource directories before discovering the global resource.

Device-Specific Resources in iOS

In iOS 4.0 and later, it is possible to mark individual resource files as usable only on a specific type of device. This capability simplifies the code you have to write for Universal applications. Rather than creating separate code paths to load one version of a resource file for iPhone and a different version of the file for iPad, you can let the bundle-loading routines choose the correct file. All you have to do is name your resource files appropriately.

To associate a resource file with a particular device, you add a custom modifier string to its filename. The inclusion of this modifier string yields filenames with the following format:

<basename>`~`*<device>*`.`*<filename_extension>*

The *<basename>* string represents the original name of the resource file. It also represents the name you use when accessing the file from your code. Similarly, the *<filename_extension>* string is the standard filename extension used to identify the type of the file. The *<device>* string is a case-sensitive string that can be one of the following values:

- `~ipad` - The resource should be loaded on iPad devices only.
- `~iphone` - The resource should be loaded on iPhone or iPod touch devices only.

You can apply device modifiers to any type of resource file. For example, suppose you have an image named `MyImage.png`. To specify different versions of the image for iPad and iPhone, you would create resource files with the names `MyImage~ipad.png` and `MyImage~iphone.png` and include them both in your bundle. To load the image, you would continue to refer to the resource as `MyImage.png` in your code and let the system choose the appropriate version, as shown here:

```
UIImage* anImage = [UIImage imageNamed:@"MyImage.png"];
```

On an iPhone or iPod touch device, the system loads the `MyImage~iphone.png` resource file, while on iPad, it loads the `MyImage~ipad.png` resource file. If a device-specific version of a resource is not found, the system falls back to looking for a resource with the original filename, which in the preceding example would be an image named `MyImage.png`.

Getting the Path to a Resource

In order to locate a resource file in a bundle, you need to know the name of the file, its type, or a combination of the two. Filename extensions are used to identify the type of a file; therefore, it is important that your resource files include the appropriate extensions. If you used custom subdirectories in your bundle to organize resource files, you can speed up the search by providing the name of the subdirectory that contains the desired file.

Even if you do not have a bundle object, you can still search for resources in directories whose paths you know. Both Core Foundation and Cocoa provide API for searching for files using only path-based information. (For example, in Cocoa you can use the `NSFileManager` object to enumerate the contents of directories and test for the existence of files.) However, if you plan to retrieve more than one resource file, it is always faster to use a bundle object. Bundle objects cache search information as they go, so subsequent searches are usually faster.

Using Cocoa to Find Resources

If you have an `NSBundle` object, you can use the following methods to find the location of resources in that bundle:

`pathForResource:ofType:`

`pathForResource:ofType:inDirectory:`

`pathForResource:ofType:inDirectory:forLocalization:`

`pathsForResourcesOfType:inDirectory:`

`pathsForResourcesOfType:inDirectory:forLocalization:`

Suppose you have placed an image called `Seagull.jpg` in your application's main bundle. Listing 3-8 shows you how to retrieve the path for this image file from the application's main bundle.

Listing 3-8 Finding a single resource file using `NSBundle`

```
NSBundle* myBundle = [NSBundle mainBundle];
NSString* myImage = [myBundle pathForResource:@"Seagull" ofType:@"jpg"];
```

If you wanted to look for all image resources in your top-level resource directory, instead of looking for just a single resource, you could use the `pathsForResourcesOfType:inDirectory:` method or one of its equivalents, as shown in Listing 3-9

Listing 3-9 Finding multiple resources using `NSBundle`

```
NSBundle* myBundle = [NSBundle mainBundle];
NSArray* myImages = [myBundle pathsForResourcesOfType:@"jpg"
                              inDirectory:nil];
```

Using Core Foundation to Find Resources

If you have a `CFBundleRef` opaque type, you can use the following methods to find the location of resources in that bundle:

`CFBundleCopyResourceURL`

`CFBundleCopyResourceURLInDirectory`

`CFBundleCopyResourceURLsOfType`

`CFBundleCopyResourceURLsOfTypeInDirectory`

`CFBundleCopyResourceURLsOfTypeForLocalization`

Suppose you have placed an image called `Seagull.jpg` in your application's main bundle. Listing 3-10 shows you how to search for this image by name and type using the Core Foundation function `CFBundleCopyResourceURL`. In this case, the code looks for the file named "Seagull" with the file type (filename extension) of "jpg" in the bundle's resource directory.

Listing 3-10 Finding a single resource using a `CFBundleRef`

```
CFURLRef    seagullURL;

// Look for a resource in the main bundle by name and type.
seagullURL = CFBundleCopyResourceURL( mainBundle,
                CFSTR("Seagull"),
                CFSTR("jpg"),
                NULL );
```

Suppose that instead of searching for one image file, you wanted to get the names of all image files in a directory called `BirdImages`. You could load all of the JPEGs in the directory using the function `CFBundleCopyResourceURLsOfType`, as shown in Listing 3-11.

Listing 3-11 Finding multiple resources using a `CFBundleRef`

```
CFArrayRef  birdURLs;

// Find all of the JPEG images in a given directory.
birdURLs = CFBundleCopyResourceURLsOfType( mainBundle,
                CFSTR("jpg"),
                CFSTR("BirdImages") );
```

Note You can search for resources that do not have a filename extension. To get the path to such a resource, specify the complete name of the resource and specify `NULL` for the resource type.

Opening and Using Resource Files

Once you have a reference to a resource file, you can load its contents and use it in your application. The steps you must take to load and use resource files depends on the type of resource, and as such is not covered in this document. For detailed information about loading and using resources, see *Resource Programming Guide*.

Finding Other Files in a Bundle

With a valid bundle object, you can retrieve the path to the top-level bundle directory as well as paths to many of its subdirectories. Using the available interfaces to retrieve directory paths insulates your code from having to know the exact structure of the bundle or its location in the system. It also allows you to use the same code on different platforms. For example, you could use the same code to retrieve resources from an iOS application or a Mac app, which have different bundle structures.

To get the path to the top-level bundle directory using Cocoa, you use the `bundlePath` method of the corresponding `NSBundle` object. You can also use the `builtInPlugInsPath`, `resourcePath`, `sharedFrameworksPath`, and `sharedSupportPath` methods to obtain the paths for key subdirectories of the bundle. These methods return path information using an `NSString` object, which you can pass directly to most other `NSBundle` methods or convert to an `NSURL` object as needed.

Core Foundation also defines functions for retrieving several different internal bundle directories. To get the path of the bundle itself, you can use the `CFBundleCopyBundleURL` function. You can also use the `CFBundleCopyBuiltInPlugInsURL`, `CFBundleCopyResourcesDirectoryURL`, `CFBundleCopySharedFrameworksURL`, and `CFBundleCopySupportFilesDirectoryURL` functions to obtain the locations of key subdirectories of the bundle. Core Foundation always returns bundle paths as a `CFURLRef` opaque type. You can use this type to extract a `CFStringRef` type that you can then pass to other Core Foundation routines.

Getting the Bundle's Info.plist Data

One file that every bundle should contain is an information property list (`Info.plist`) file. This file is an XML-based text file that contains specific types of key-value pairs. These key-value pairs specify information about the bundle, such as its ID string, version number, development region, type, and other important properties. (See *Runtime Configuration Guidelines* for the list of keys you can include in this file.) Bundles may also include other types of configuration data, mostly organized in XML-based property lists.

The `NSBundle` class provides the `objectForInfoDictionaryKey:` and `infoDictionary` methods for retrieving information from the `Info.plist` file. The `objectForInfoDictionaryKey:` method returns the localized value for a key and is the preferred method to call. The `infoDictionary` method returns an `NSDictionary` with all of the keys from the property list; however, it does not return any localized values for these keys. For more information, see the *NSBundle Class Reference*.

Core Foundation also offers functions for retrieving specific pieces of data from a bundle's information property list file, including the bundle's ID string, version, and development region. You can retrieve the localized value for a key using the `CFBundleGetValueForInfoDictionaryKey` function. You can also retrieve the entire dictionary of non-localized keys using `CFBundleGetInfoDictionary`. For more information about these and related functions, see *CFBundle Reference*.

> **Note** Because they take localized values into account, `CFBundleGetValueForInfoDictionaryKey` and `objectForInfoDictionaryKey:` are the preferred interfaces for retrieving keys.

Listing 3-12 demonstrates how to retrieve the bundle's version number from the information property list using Core Foundation functions. Though the value in the information property list may be written as a string, for example "2.1.0b7", the value is returned as an unsigned long integer.

Listing 3-12 Obtaining the bundle's version

```
// This is the 'vers' resource style value for 1.0.0
#define kMyBundleVersion1 0x01008000

UInt32  bundleVersion;

// Look for the bundle's version number.
bundleVersion = CFBundleGetVersionNumber( mainBundle );

// Check the bundle version for compatibility with the app.
if (bundleVersion < kMyBundleVersion1)
    return (kErrorFatalBundleTooOld);
```

Listing 3-13 shows you how to retrieve arbitrary values from the information property list using the `CFBundleGetInfoDictionary` function. The resulting information property list is an instance of the standard Core Foundation type `CFDictionaryRef`. For more information about retrieving information from a Core Foundation dictionary, see *CFDictionary Reference*.

Listing 3-13 Retrieving information from a bundle's information property list

```
CFDictionaryRef bundleInfoDict;
CFStringRef     myPropertyString;
```

```
// Get an instance of the non-localized keys.
bundleInfoDict = CFBundleGetInfoDictionary( myBundle );

// If we succeeded, look for our property.
if ( bundleInfoDict != NULL ) {
    myPropertyString = CFDictionaryGetValue( bundleInfoDict,
                CFSTR("MyPropertyKey") );
}
```

It is also possible to obtain an instance of a bundle's information dictionary without a bundle object. To do this you use either the Core Foundation function `CFBundleCopyInfoDictionaryInDirectory` or the Cocoa `NSDictionary` class. This can be useful for searching the information property lists of a set of bundles without first creating bundle objects.

Loading and Unloading Executable Code

The key to loading code from an external bundle is finding an appropriate entry point into the bundle's executable file. As with other plug-in schemes, this requires some coordination between the application developer and the plug-in developer. You can publish a custom API for bundles to implement or define a formal plug-in interface. In either case, once you have an appropriate bundle or plug-in, you use the `NSBundle` class (or the `CFBundleRef` opaque type) to access the functions or classes implemented by the external code.

> **Note** Another option for loading Mach-O code directly is to use the `NSModule` loading routines. However, these routines typically require more work to use and are less preferable than the `NSBundle` or `CFBundleRef` interfaces. For more information, see *OS X ABI Mach-O File Format Reference* in OS X Documentation or see the `NSModule` man pages.

For additional information about loading and unloading code, see *Code Loading Programming Topics*.

Loading Functions

If you are working in C, C++, or even in Objective-C, you can publish your interface as a set of C-based symbols, such as function pointers and global variables. Using the Core Foundation functions, you can load references to those symbols from a bundle's executable file.

You can retrieve symbols using any of several functions. To retrieve function pointers, call either `CFBundleGetFunctionPointerForName` or `CFBundleGetFunctionPointersForNames`. To retrieve a pointer to a global variable, call `CFBundleGetDataPointerForName` or `CFBundleGetDataPointersForNames`. For example, suppose a loadable bundle defines the function shown in Listing 3-14.

Listing 3-14 An example function for a loadable bundle

```
    // Add one to the incoming value and return it.
    long addOne(short number)
    {
        return ( (long)number + 1 );
    }
```

Given a `CFBundleRef` opaque type, you would need to search for the desired function before you could use it in your code. Listing 3-15 shows a code fragment that illustrates this process. In this example, the `myBundle` variable is a `CFBundleRef` opaque type pointing to the bundle.

Listing 3-15 Finding a function in a loadable bundle

```
// Function pointer.
AddOneFunctionPtr addOne = NULL;

// Value returned from the loaded function.
long result = 0;

// Get a pointer to the function.
addOne = (void*)CFBundleGetFunctionPointerForName(
            myBundle, CFSTR("addOne") );

    // If the function was found, call it with a test value.
if (addOne)
{
    // This should add 1 to whatever was passed in
    result = addOne ( 23 );
}
```

Loading Objective-C Classes

If you are writing a Cocoa application, you can load the code for an entire class using the methods of `NSBundle`. The `NSBundle` methods for loading a class are for use with Objective-C classes only and cannot be used to load classes written in C++ or other object-oriented languages.

If a loadable bundle defines a principal class, you can load it using the `principalClass` method of `NSBundle`. The `principalClass` method uses the `NSPrincipalClass` key of the bundle's `Info.plist` file to identify and load the desired class. Using the principal class alleviates the need to agree on specific naming conventions for external classes, instead letting you focus on the behavior of those interfaces. For example, you might use an instance of the principal class as a factory for creating other relevant objects.

If you want to load an arbitrary class from a loadable bundle, call the `classNamed:` method of `NSBundle`. This method searches the bundle for a class matching the name you specify. If the class exists in the bundle, the method returns the corresponding `Class` object, which you can then use to create instances of the class.

Listing 3-16 shows you a sample method for loading a bundle's principal class.

Listing 3-16 Loading the principal class of a bundle

```
- (void)loadBundle:(NSString*)bundlePath
{
    Class exampleClass;
    id newInstance;
    NSBundle *bundleToLoad = [NSBundle bundleWithPath:bundlePath];
    if (exampleClass = [bundleToLoad principalClass])
    {
        newInstance = [[exampleClass alloc] init];
        // [newInstance doSomething];
    }
}
```

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

Unloading Bundles

In OS X v10.5 and later, you can unload the code associated with an `NSBundle` object using the `unload` method. You might use this technique to free up memory in an application by removing plug-ins or other loadable bundles that you no longer need.

If you used Core Foundation to load your bundle, you can use the `CFBundleUnloadExecutable` function to unload it. If your bundle might be unloaded, you need to ensure that string constants are handled correctly by setting an appropriate compiler flag.

When you compile a bundle with a minimum deployment target of OS X v10.2 (or later), the compiler automatically switches to generating strings that are truly constant in response to `CFSTR("...")`. The compiler also generates these constant strings if you compile with the flag `-fconstant-cfstrings`. Constant strings have many benefits and should be used when possible, however if you reference constant strings after the executable containing them is unloaded, the references will be invalid and will cause a crash. This might happen even if the strings have been retained, for example, as a result of being put in data structures, retained directly, and, in some cases, even copied. Rather than trying to make sure all such references are cleaned up at unload time (and some references might be created within the libraries, making them hard to track), it is best to compile unloadable bundles with the flag `-fno-constant-cfstrings`.

Document Packages

If your document file formats are getting too complex to manage because of several disparate types of data, you might consider adopting a package format for your documents. Document packages give the illusion of a single document to users but provide you with flexibility in how you store the document data internally. Especially if you use several different types of standard data formats, such as JPEG, GIF, or XML, document packages make accessing and managing that data much easier.

Defining Your Document Directory Structure

Apple does not prescribe any specific structure for document packages. The contents and organization of the package directory are left to you. You are encouraged, however, to create either a flat directory structure or use the framework bundle structure, which involves placing your files in a top-level `Resources` subdirectory. (For more information about the bundle structure of frameworks, see "Framework Bundles" (page 27).)

Registering Your Document Type

To register a document as a package, you must modify the document type information in your application's information property list (`Info.plist`) file. The `CFBundleDocumentTypes` key stores information about the document types your application supports. For each document package type, include the `LSTypeIsPackage` key with an appropriate value. The presence of this key tells the Finder and Launch Services to treat directories with the given file extension as a package. For more information about `Info.plist` keys, see *Information Property List Key Reference*.

Document packages should always have an extension to identify them—even though that extension may be hidden by the user. The extension allows the Finder to identify your document directory and treat it as a package. You should never associate a document package with a MIME type or 4-byte OS type.

Creating a New Document Package

When it is time for your application to create a new document, it can do so in one of two ways:

- Use an `NSFileWrapper` object to create the document package.

- Create the document package manually.

If you are creating a Cocoa application, the `NSFileWrapper` class is the preferred way to create document packages because it ties in with the existing support in `NSDocument` for reading and writing your document contents. (The `NSFileWrapper` class is also available in iOS 4.0 and later as part of the Foundation framework.) For information on how to use this class, see *NSFileWrapper Class Reference*.

If you are writing a command-line tool or other type of application that does not use the higher-level Objective-C frameworks (such as AppKit or UIKit), you can still create document packages manually. The important thing to remember about creating a document package is that it is just a directory. As long as the type of the document package is registered (as described in "Registering Your Document Type" (page 48)), all you have to do is create a directory with the appropriate filename extension. (The Finder uses the filename extension as its cue to treat the directory as a package.) You can create the directory (and create any files you want to put inside that directory) using the standard BSD file system routines or using the `NSFileManager` class in the Foundation framework.

Accessing Your Document Contents

There are several ways to access the contents of a document package. Because a document package is a directory, you can access the document's contents using any appropriate file-system routines. If you use a bundle structure for your document package, you can also use the `NSBundle` or `CFBundleRef` routines. Use of a bundle structure is especially appropriate for documents that store multiple localizations.

If your document package uses a flat directory structure or contains a fixed set of content files, you might find using file-system routines faster and easier than using `NSBundle` or `CFBundleRef`. However, if the contents of your document can fluctuate, you should consider using a bundle structure and `NSBundle` or `CFBundleRef` to simplify the dynamic discovery of files inside your document.

If you are creating a Cocoa application, you must also remember to customize the way your `NSDocument` subclass loads the contents of the document package. The traditional technique of using the `readFromData:ofType:error:` and `dataOfType:error:` methods to read and write data are intended for a single file document. To handle a document package, you must use the `readFromFileWrapper:ofType:error:` and `fileWrapperOfType:error:` methods instead. For information about reading and writing document data from your `NSDocument` subclass, see *Document-Based Applications Overview*.

Glossary

bundle A directory that contains an "Glossary" (page 50) and whose contents are organized in one of several ways recognized by the system.

display name A user-visible string displayed in place of an item's actual name. Display names allow the user to customize the names of key items (like applications) without breaking parts of the system that rely on the original name.

framework A bundle structure containing a dynamic shared library and the header files and other resources to support that library.

information property list A specific type of property list that contains configuration information for a bundle. An information property list file must always have the name `Info.plist`. For more information, see *Runtime Configuration Guidelines*.

loadable bundle A bundle whose executable is designed to be loaded into memory dynamically by an application. Loadable bundles are also sometimes referred to as plug-ins.

package A directory that the Finder presents to the user as if it were a single file.

target An Xcode blueprint for creating a product. A target defines rules for compiling source files, copy resource files, and performing any other steps needed to build the resulting product.

versioned bundle A bundle that supports the inclusion of multiple versions of an executable and resources. Frameworks are the only type of bundle that support versions.

Document Revision History

This table describes the changes to *Bundle Programming Guide*.

Date	Notes
2010-07-08	Changed references of iOS to iOS.
2010-05-25	Added information about accessing device-specific bundle resources.
2009-07-14	Added sections on how to create bundles and document packages.
2009-05-22	Updated the document to reflect the bundle structure of iPhone applications. Reorganized the document contents. Added overview information about frameworks and loadable bundles. Removed references to Mac OS 9 and legacy resource types.
2005-11-09	Updated information on how to create document bundles. Updated guidance on how to build a bundle manually. Clarified the distinction between bundles and packages.
2005-07-07	Updated the list of conditions under which a main bundle might be NULL. Fixed typos. Changed title from *Bundles*. Added a link to *Framework Programming Guide*.
2005-03-03	Corrected typos.
2004-08-31	Added notes about the correct capitalization of files and directories in a bundle. Added section about unloadable bundles.

Date	Notes
2004-03-26	Merged content from OS X Bundles into this document. Update content to reflect both Cocoa and Core Foundation interfaces. Added guidelines for using bundles. Removed information about the anatomy of framework bundles. That information is now covered in *Framework Programming Guide*. Fixed minor bugs.
2003-10-22	Updated links for *Internationalizing Your Software*.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.

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