NSBundle

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSBundle.h

Class at a Glance

Purpose

An NSBundle represents a location in the file system that groups code and resources that can be used in a program. NSBundles locate program resources, dynamically load executable code, and assist in localization. You build a bundle in Project Builder using one of these project types: Application, pathForRFramework, Loadable Bundle, Palette.

Principal Attribute

- Directory path

Creation

– initWithPath:	(designated initializer)
+ mainBundle	Returns the NSBundle for the application wrapper.
+ bundleForClass:	Returns the NSBundle in which the class is implemented.
+ bundleWithPath:	Returns the NSBundle at a location in the file system.

Commonly Used Methods

– localizedStringForKey:value:table:	Returns a localized version of a string.
– pathForResource:ofType:	Returns the path for the specified resource.
– principalClass	Returns the principal class, dynamically loading code if needed.

Class Description

An NSBundle is an object that corresponds to a directory where related resources—including executable code—are stored. The directory, in essence, "bundles" a set of resources used by an application into convenient chunks, and the NSBundle object makes those resources available to the application. NSBundle can find requested resources in the directory and can dynamically load executable code. The term *bundle* refers both to the object and to the directory it represents.

Bundles are useful in a variety of contexts. Since bundles combine executable code with the resources used by that code, they facilitate installation and localization. NSBundles are also used to locate specific resources, to obtain localized strings, to load code dynamically, and to determine which classes are loaded.

Each resource in a bundle usually resides in its own file. Bundled resources include such things as:

- Executable code
- Images—TIFF or EPS images used by an application's user interface
- Sounds
- Localized character strings
- Nib files—Interface Builder files describing user-interface objects and their relationships

The Project Builder application defines four types of projects that build bundles as *file packages*. A file package is a directory that the Workspace Manager presents to users as if it were a simple file; the contents of the directory are hidden. The four types of Project Builder bundles are:

- **Application**. The application wrapper is a bundle that contains the resources needed to launch the application, including the application executable. This bundle is also known as the main bundle. Its extension is ".app".
- **Framework**. A framework is a directory containing a dynamic shared library and all the resources that go with that library, such as header files, images, and documentation. Its extension is ".framework".
- **Loadable Bundle**. Like an application, a loadable bundle usually contains executable code and associated resources. Loadable bundles differ from applications and frameworks because they must be explicitly loaded into a running application. (See "Loadable Bundles," below for more information.) The extension of a loadable bundle is conventionally ".bundle" but can be something else (for example, ".preference").
- **Palette**. A palette is a type of loadable bundle specialized for Interface Builder. It contains custom user-interface objects and compiled code that are loaded into an Interface Builder palette.

For all types of bundles, the executable-code file of a bundle (of which there can be only one) is in the immediate bundle directory and takes the same name as the bundle, minus the extension. Bundles also encode (as a property list) the important attributes of the bundle, such as the main nib file name, executable name, document extensions, and so forth. You can access these attributes with NSBundle's **infoDictionary** method, which returns the file's contents as an NSDictionary.

You shouldn't attempt subclassing NSBundle since the designated initializer, **initWithPath:**, might substitute another NSBundle for **self**.

The Main Bundle

Every application has at least one bundle—its *main bundle*—which is the ".app" directory where its executable file is located. This file is loaded into memory when the application is launched. It includes at least the **main()** function and other code necessary to start up the application. You obtain an NSBundle object corresponding to the main bundle with the class method **mainBundle**.

Framework Bundles

Frameworks are bundles that package dynamic shared libraries along with the nib files, images, and other resources that support the executable code and with the header files and documentation that describe the associated APIs. As long as your applications are dynamically linked with frameworks, you should have little need to do anything explicitly with those frameworks thereafter; in a running application, the framework code is automatically loaded, as needed. You can however, get an NSBundle object associated with a framework by invoking the class method **bundleForClass:** specifying, as the argument, a class that's defined in the framework.

Loadable Bundles and Dynamic Loading

An application can be organized into any number of other bundles in addition to the main bundle and the bundles of linked-in frameworks. Although these loadable bundles usually reside inside the application file package, they can be located anywhere in the file system. Each loadable-bundle directory—by convention, with a ".bundle" extension—is represented in the application by a separate NSBundle object. Through this object the application can dynamically load the code and resources in the bundle when it needs them. For example, an application for managing PostScript printers may have a bundle full of PostScript code to be downloaded to printers.

The executable code files in loadable bundles hold class (and category) definitions that the NSBundle object can dynamically load while the application runs. When asked for a certain class (through the invocation of **classNamed:** or **principalClass**), the NSBundle loads the object file that contains the class definition (if it's not already loaded) and returns the class object; it also loads other classes and categories that are stored in the file.

The major advantage of bundles is application extensibility. A set of bundled classes often supports a small collection of objects that can be integrated into the larger object network already in place. (NEXTSTEP Preferences is one example of this.) The linkage is established through an instance of the *principal class*. This object might have methods to return other objects that the application can talk to, but typically all messages from the application to the subnetwork are funneled through the one instance.

Since each bundle can have only one executable file, that file should be kept free of localizable content. Anything that needs to be localized should be segregated into separate resource files and stored in localized-resource subdirectories.

Note: To create a loadable bundle—a bundle with dynamically loadable code—without using Project Builder, use the **ld**(1) **-bundle** flag on the **cc** command line.

Localized Resources

If an application is to be used in more than one part of the world, its resources may need to be customized, or "localized," for language, country, or cultural region. An application may need, for example, to have separate Japanese, English, French, Hindu, and Swedish versions of the character strings that label menu commands.

Resource files specific to a particular language are grouped together in a subdirectory of the bundle directory. The subdirectory has the name of the language (in English) followed by a ".lproj" extension (for "language project"). The application mentioned above, for example, would have **Japanese.lproj**, **English.lproj**, **French.lproj**, **Hindi.lproj**, and **Swedish.lproj** subdirectories. Each ".lproj" subdirectory in a bundle has the same set of files; all versions of a resource file must have the same name. Thus, **Hello.snd** in **French.lproj** should be the French counterpart to the Swedish **Hello.snd** in **Swedish.lproj**, and so on. If a resource doesn't need to be localized at all, it's stored in the bundle directory itself, not in the ".lproj" subdirectories.

The user determines which set of localized resources will actually be used by the application. NSBundle objects rely on the language preferences set by the user in the Preferences application. Preferences lets users order a list of available languages so that the most preferred language is first, the second most preferred language is second, and so on.

When an NSBundle is asked for a resource file, it provides the path to the resource that best matches the user's language preferences. For details, see the descriptions of **pathForResource:ofType:inDirectory** and **pathForResource:ofType:.**

Application Kit Additions to NSBundle

The Application Kit defines two categories of NSBundle, one for locating image resources and the other for loading nib files. The methods in these categories become part of the NSBundle class only for those applications that use the Application Kit. For details, see the specifications file **NSBundleAdditions.rtf** in the Application Kit reference documentation.

Method Types

Initializing an NSBundle	– initWithPath:
Getting an NSBundle	+ bundleForClass: + bundleWithPath: + mainBundle
Getting a bundled class	– classNamed: – principalClass

Finding a resource	– pathForResource:ofType: – pathForResource:ofType:inDirectory: – pathsForResourcesOfType:inDirectory: – resourcePath
Getting the bundle directory	– bundlePath
Getting bundle information	– infoDictionary
Managing localized resources	– localizedStringForKey:value:table:

Class Methods

bundleForClass:

+ (NSBundle *)**bundleForClass:**(Class)*aClass*

Returns the NSBundle that dynamically loaded *aClass* (a loadable bundle), the NSBundle for the framework in which *aClass* is defined, or the main bundle object if *aClass* was not dynamically loaded or is not defined in a framework.

See also: + **mainBundle**, + **bundleWithPath:**

bundleWithPath:

+ (NSBundle *)**bundleWithPath:**(NSString *)*path*

Returns an NSBundle that corresponds to the specified directory *path* or **nil** if *path* does not identify an accessible bundle directory. This method allocates and initializes the returned object if it doesn't already exist.

See also: + **mainBundle**, + **bundleForClass:**

mainBundle

+ (NSBundle *)**mainBundle**

Returns an NSBundle that corresponds to the directory where the application executable is located or **nil** if this executable is not located in a accessible bundle directory. This method allocates and initializes the returned NSBundle if it doesn't already exist.

In general, the main bundle corresponds to an application file package or application wrapper: a directory that bears the name of the application and is marked by a ".app" extension.

See also: + **bundleForClass:**, + **bundleWithPath:**

Instance Methods

bundlePath

– (NSString *)**bundlePath**

Returns the full pathname of the receiver's bundle directory.

classNamed:

– (Class)**classNamed:**(NSString *)*className*

Returns the class named *className*. If the bundle's executable code is not yet loaded, this method dynamically loads it into memory. The method returns **nil** if *className* isn't one of the classes associated with the receiver or if there is an error in loading the executable code containing the class implementation. Classes (and categories) are loaded from just one file within the bundle directory; this code file has the same name as the directory, but without the extension (".bundle," ".app," ".framework"). As a side-effect of code loading, the receiver posts NSBundleNotification for each class and category loaded; see "Notifications," below for details.

The following example loads a bundle's executable code containing the class "FaxWatcher."

```
- (void)loadBundle:(id)sender
{
    Class exampleClass;
    id newInstance;
    NSString *str = @"/me/Projects/BundleExample/BundleExample.bundle";
    NSBundle *bundleToLoad = [NSBundle bundleWithPath:str];
    if (exampleClass = [bundleToLoad classNamed:@"FaxWatcher"]) {
        newInstance = [[exampleClass alloc] init];
        // [newInstance doSomething];
     }
}
```

See also: – **principalClass**

NS+ infoDictionary

– (NSDictionary *)**infoDictionary**

Returns a dictionary that contains information about the receiver. This information is extracted from the property list (**Info.plist**) associated with the bundle. The returned dictionary is empty if no **Info.plist** can be found. Common keys for accessing the values of the dictionary are NSExecutable, NSExtensions, NSIcon, NSMainNibFile, and NSPrincipalClass.

See also: – **principalClass**

initWithPath:

– (id)**initWithPath:**(NSString *)*fullPath*

Returns an NSBundle corresponding to the directory *fullPath*. This method initializes and returns a new instance only if there is no existing NSBundle associated with *fullPath*, in which case it deallocates **self** and returns the existing object. *fullPath* must be a full pathname for a directory; if it contains any symbolic links, they must be resolvable. If the directory doesn't exist or the user doesn't have access to it, this method returns **nil**.

It's not necessary to allocate and initialize an instance for the main bundle; use the **mainBundle** class method to get this instance. You can also use the **bundleWithPath:** class method to obtain a bundle identified by its directory path.

See also: + **bundleForClass:**

localizedStringForKey:value:table:

– (NSString *)**localizedStringForKey:**(NSString *)*key*
value:(NSString *)*value*
table:(NSString *)*tableName*

Returns a localized version of the string designated by *key* in table *tableName*. The argument *tableName* specifies the receiver's string table to search. If *tableName* is **nil** or is an empty string, the method attempts to use the table in **Localizable.strings**; if this table is not present, it looks for any file with a ".strings" extension. The *value* argument specifies the value to return if a localized string can't be found in the table. If *value* is **nil** or an empty string, and a localized string is not found in the table, the method returns the key with all characters in uppercase; this will facilitate updating the table.

Note: You can toggle the feature that returns the keys of non-localized strings in uppercase by turning on the defaults (**dwrite**) variable **NSShowNonLocalizedStrings** per-application or globally.

This example cycles through a static array of keys when a button is clicked, gets the value for each key from a strings table named **Buttons.strings**, and sets the button title with the returned value.

```
- (void)changeTitle:(id)sender
{
    static int keyIndex = 0;
    NSBundle *thisBundle = [NSBundle bundleForClass:[self class]];

    NSString *locString = [thisBundle
       localizedStringForKey:assortedKeys[keyIndex++]
       value:@"" table:@"Buttons"];
    [sender setTitle:locString];
    if (keyIndex == MAXSTRINGS) keyIndex=0;
}
```

See also: – **pathForResource:ofType:, – pathForResource:ofType:inDirectory:**

pathForResource:ofType:

– (NSString *)**pathForResource:**(NSString *)*name* **ofType:**(NSString *)*extension*

Returns the full pathname for the resource identified by *name* and having the specified file name *extension*. The *extension* argument can be **nil** or an empty string (@""); in either case the file name returned is the first one encounted with *name*, regardless of the extension. The method first looks for the resource in the language-specific ".lproj" directory (the local language is determined by user defaults); if the resource is not there, it looks for a non-localized resource in the immediate bundle directory.

The following code fragment gets the path to a localized sound, creates an Sound instance from it, and plays the sound.

```
NSString *soundPath;
Sound *thisSound;
NSBundle *thisBundle = [NSBundle bundleForClass:[self class]];
if (soundPath = [thisBundle pathForResource:@"Hello" ofType:@"snd"]) {
    thisSound = [[[Sound alloc] initFromSoundfile:soundPath] autorelease];
    [thisSound play];
}
```

– **localizedStringForKey:value:table:**

pathForResource:ofType:inDirectory:

– (NSString *)**pathForResource:**(NSString *)*name*
ofType:(NSString *)*extension*
inDirectory:(NSString *)*bundlePath*

Returns the full pathname for the resource identified by *name*, having the specified file name *extension*, and residing in the directory *bundlePath*; returns **nil** if no matching resource file exists in the bundle. The argument *bundlePath* must be a valid bundle directory or **nil**. The argument *extension* can be an empty string or **nil**; in either case the pathname returned is the first one encounted with *name*, regardless of the extension. If *bundlePath* is specified, the method searches in this order:

<main bundle path>/**Resources**/*bundlePath*/<language.lproj>/*name.extension*
<main bundle path>/**Resources**/*bundlePath*/*name.extension*
<main bundle path>/*bundlePath*/<language.lproj>/*name.extension*
<main bundle path>/*bundlePath*/*name.extension*

The order of language directories searched corresponds to the user's preferences. If *bundlePath* is **nil**, the same search order as described above is followed, minus *bundlePath*.

See also: – **localizedStringForKey:value:table:**

pathsForResourcesOfType:inDirectory:

– (NSArray *)**pathsForResourcesOfType:**(NSString *)*extension*
inDirectory:(NSString *)*bundlePath*

Returns an array containing pathnames for all bundle resources having the specified file name *extension* and residing in the directory *bundlePath*; returns an empty array if no matching resource files are found. This method provides a means for dynamically discovering bundle resources. The argument *bundlePath* must be a valid bundle directory or **nil**. The *extension* argument can be an empty string or **nil**; if you specify either of these for *extension*, however, all bundle resources are returned. Although there is no guaranteed search order, all of the following directories will be searched:

<main bundle path>/**Resources**/*bundlePath*/<language.lproj>/*name.extension*
<main bundle path>/**Resources**/*bundlePath*/*name.extension*
<main bundle path>/*bundlePath*/<language.lproj>/*name.extension*
<main bundle path>/*bundlePath*/*name.extension*

The language directories searched corresponds to the user's preferences. If *bundlePath* is **nil**, the same search order as described above is followed, minus *bundlePath*.

See also: – **localizedStringForKey:value:table:**

principalClass

– (Class)**principalClass**

Returns the NSBundle's principal class after ensuring that the code containing the definition of that class is dynamically loaded. If the NSBundle encounters errors in loading or if it can't find the executable code file in the bundle directory, it returns **nil**. The principal class typically controls all the other classes in the bundle; it should mediate between those classes and classes external to the bundle. Classes (and categories) are loaded from just one file within the bundle directory. Obtain the name of the code file to load from the dictionary returned from **infoDictionary**, using "NSExecutable" as the key. The NSBundle determines its principal class in one of two ways:

- It first looks in its own information dictionary, which extracts the information encoded in the bundle's property list (**Info.plist**). NSBundle obtains the principal class from the dictionary using the key NSPrincipalClass. For non-loadable bundles (applications and frameworks), if the principal class is not specified in the property list, the method returns **nil**.
- If the principal class is not specified in the information dictionary, NSBundle identifies the first class loaded as the principal class. When several classes are linked into a dynamically loadable file, the default principal class is the first one listed on the **ld** command line. In the following example, Reporter would be the principal class:

```
ld -o myBundle -r Reporter.o NotePad.o QueryList.o
```

Note: The order of classes in Project Builder's project browser is the order in which they will be linked. To designate the principal class, Control-drag the file containing its implementation to the top of the list.

As a side-effect of code loading, the receiver posts NSBundleDidLoadNotification after each class and category is loaded; see "Notifications," below for details.

The following method obtains a bundle by specifying its path (**bundleWithPath:**), then loads the bundle with **principalClass** and uses the returned class object to allocate and initialize an instance of that class.

```
- (void)loadBundle:(id)sender
{
    Class exampleClass;
    id newInstance;
    NSString *path = @"/tmp/Projects/BundleExample/BundleExample.bundle";
    NSBundle *bundleToLoad = [NSBundle bundleWithPath:path];
    if (exampleClass = [bundleToLoad principalClass]) {
        newInstance = [[exampleClass alloc] init];
        [newInstance doSomething];
     }
 }
```

See also: – **classNamed:**, – **infoDictionary**

resourcePath

– (NSString *)**resourcePath**

Returns the full pathname of the receiving bundle's subdirectory containing resources.

See also: – **bundlePath**

Notifications

The following notification is declared and posted by NSBundle.

NSBundleDidLoadNotification

Notification Object The NSBundle that dynamically loads classes

userInfo Dictionary

Key	Value
NSLoadedClasses	An NSArray containing the names (as NSStrings) of each class that was loaded

NSBundle posts NSBundleDidLoadNotification to notify observers which classes have been dynamically loaded. When a request is made to an NSBundle for a class (**classNamed:** or **principalClass**), the bundle

dynamically loads the executable code file that contain the class implementation and all other class definitions contained in the file. After the module is loaded, the NSBundle posts a notification with a **userInfo** dictionary containing all classes that were loaded.

In a typical use of this notification, an object might want to enumerate the **userInfo** NSArray to check if each loaded class conformed to a certain protocol (say, a protocol for a plug-and-play tool set); if a class does conform, the object would create an instance of that class and add the instance to another NSArray.