Resource Programming Guide

Cocoa > Resource Management

2009-01-06

Contents

Chapter 4   **Image, Sound, and Video Resources   39**

Chapter 5   **Carbon Resources   47**

Document Revision History   53

Figures, Tables, and Listings

Introduction

Applied to computer programs, resources are data files that accompany a program's executable code. Resources simplify the code you have to write by moving the creation of complex sets of data or graphical content outside of your code and into more appropriate tools. For example, rather than creating images pixel by pixel using code, it is much more efficient (and practical) to create them in an image editor. To take advantage of a resource, all your code has to do is load it at runtime and use it.

In addition to simplifying your code, resources are also an intimate part of the internationalization process for all applications. Rather than hard-coding strings and other user-visible content in your application, you can place that content in external resource files. Localizing your application then becomes a simple process of creating new versions of each resource file for each supported language. The bundle mechanism used in both Mac OS X and iPhone OS provides a way to organize localized resources and to facilitate the loading of resource files that match the user's preferred language.

This document provides information about the types of resources supported in Mac OS X and iPhone OS and how you use those resources in your code. This document does not focus on the resource-creation process. Most resources are created using either third-party applications or the developer tools provided in the `/Developer/Applications` directory. In addition, although this document refers to the use of resources in applications, the information also applies to other types of bundled executables, including frameworks and plug-ins.

Before reading this document, you should be familiar with the organizational structure imposed by application bundles. Understanding this structure makes it easier to organize and find the resource files your application uses. For information on the structure of bundles, see *Bundle Programming Guide*.

Organization of This Document

This document includes the following chapters:

- "About Resources" (page 9) provides an introduction to the resource types supported in Mac OS X and iPhone OS.
- "Nib Files" (page 13) describes the Cocoa-specific support for nib files.
- "Carbon Resources" (page 47) describes the Carbon-specific support for nib files.
- "String Resources" (page 29) describes the support for localized string resources in applications.
- "Image, Sound, and Video Resources" (page 39) describes the support for image, sound, and video resources in applications.

See Also

The following ADC Reference Library documents are conceptually related to *Resource Programming Guide*:

- *Bundle Programming Guide* describes the bundle structure used by applications to store executable code and resources.

- *Internationalization Programming Topics* describes the process of preparing an application (and it's resources) for translation into other languages.

- *Interface Builder User Guide* describes the application used to create nib file resources.

- *Property List Programming Guide* describes the facilities in place for loading property-list resource files into a Cocoa application.

- *Property List Programming Topics for Core Foundation* describes the facilities in place for loading property-list resource files into a C-based application.

About Resources

There are several reasons to use resources in your application:

- They can reduce the amount of code needed to create your application's user interface.

- They make it possible to change your application's user interface without changing any code.

- They make it easy to localize your application's user-visible content.

- They can store other types of custom data that might be difficult or time-consuming to create at runtime.

The following sections describe the types of resources typically found in applications and how you use them.

Nib Files

Nib files are the quintessential resource type used to create graphical applications. A nib file is a data archive describing the objects, configuration, and layout information associated with an application's user interface. You create nib files using the Interface Builder application, which provides a graphical assembly area for the windows and menus that comprise applications. You assemble windows and menus in Interface Builder by dragging and dropping custom views, controls, and other components from the provided library of objects. In addition to positioning items inside a window or view, you can configure the properties of those items as well so that they have the custom look and behavior you want for your application. In Cocoa applications, you can also create connections between objects to facilitate the passing of messages between them.

When you load a nib file, the nib-loading code recreates your objects exactly as they were when you designed them, including any custom configuration or connection information. Because the objects are given to you fully configured, it is possible to create complex user interfaces (see Figure 1-1) with very little code.

Figure 1-1   User interface resources

Custom views

Custom controller objects

Binding controllers

Menus
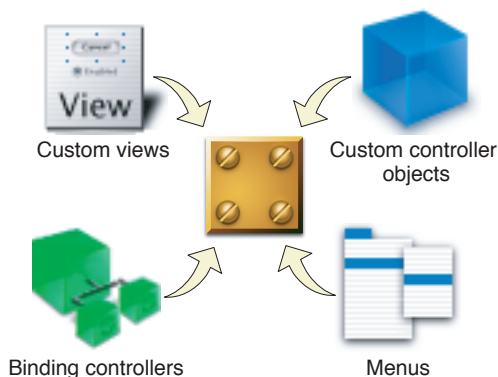
Nib files are supported widely by both Mac OS X and iPhone OS, although there are some subtle differences in how nib files are supported from environment to environment. Mac OS X and iPhone OS both provide the same basic level of support for loading nib files and using their contents. For example, both provide nib-loading support through the `NSBundle` class and automated support for loading the application's main nib file. In Mac OS X, Cocoa provides additional support for loading nib files associated with documents and for using the `NSNib` class to load nibs. The Carbon environment also supports the use of nib files, although the semantics for using them are different. For Carbon applications, nib files are a repository of user interface items that can be loaded one-by-one, instead of all at once.

> **Note:** The term "nib" and the corresponding file extension are an acronym for "NeXT Interface Builder." The Interface Builder application was originally developed at NeXT Computer, whose OPENSTEP operating system was used as the basis for creating Mac OS X.

For general information on how to use nib files in Cocoa-based applications (including iPhone applications), see "Nib Files" (page 13). For information on how to use nib files in Carbon applications, see "Carbon Resources" (page 47). For information on how to create nib files, see *Interface Builder User Guide*.

String Resources

Text strings are a prominent part of most user interfaces. Text strings are commonly found in an application's nib files but may also be found in other places as well. For example, if an error occurs, an application might load a string corresponding to that error and display the string in an alert panel. Instead of hard-coding such strings inside source files, which would make localization much more difficult, an application can instead load them from a strings resource file.

A strings resource file (also known as a **strings file** because its file extension is `.strings`) is a human-readable text file (in the UTF-16 encoding) containing a set of string resources for an application. The purpose of strings files is to provide an external repository for an application's localizable text. An application can have any number of strings files and each strings file can contain any number of strings. Each entry in a strings file consists of a key-value pair where both the key and value are themselves strings. The key portion never changes and represents the identifier that your application uses to retrieve the string; however, the value for that key is typically translated to one of the languages your application supports.

Mac OS X provides tools to help you automatically generate strings files for your application. The tools search your code for any usage of specific string-loading routines and use that code to generate strings files for you. For more information about loading string resources and generating strings files, see "String Resources" (page 29).

Image, Sound, and Multimedia Resources

Mac OS X and iPhone OS make extensive use of image resources (and to a lesser extent sound and multimedia resources) to create a unique visual style for the entire system. Some of these image resources are used to implement the glossy, three-dimensional texture commonly found in system components, such as the Aqua controls. Apple applications make extensive use of high-quality images to create the look and feel typically associated with the underlying system. Developers are similarly encouraged to use high-quality images to

create beautiful and easy-to-use interfaces for their applications. The use of images can not only simplify your drawing code but for complex visual elements can improve performance by providing a prerendered version that can be cached and reused.

Because images are such an important part of graphical user interfaces in general, and Mac OS X and iPhone OS in particular, each system provides extensive support for loading and drawing image resource files. Both Mac OS X and iPhone OS provide support for loading and decoding image files saved in a variety of different formats. For resource files, however, the most commonly used formats include PNG, TIFF, PDF, GIF, and JPEG.

Just as you use images, you can use sound and multimedia resources to create a unique presentation style for your application. Although used less frequently than images, sounds can be used to provide feedback or to alert the user to special events. Audio support is provided by the Core Audio family of frameworks and also by custom classes in the AppKit framework. Similarly, you can use movie clips to present video-based content. Mac OS X also provides extensive use for video and multimedia resources through the QuickTime and QuickTime Kit frameworks. In iPhone OS, similar support is provided by the Media Player framework.

For information about how to use image, sound, and video resources in your applications, see "Image, Sound, and Video Resources" (page 39).

Property Lists

Property list files are a way to store custom configuration data outside of your application code. Mac OS X and iPhone OS use property lists extensively to implement features such as user preferences and information property list files for bundles. You can similarly use property lists to store private (or public) configuration data for your applications.

A property-list file is essentially a set of structured data values. You can create and edit property lists either programmatically or using the Property List Editor application (located in `/Developer/Applications/Utilities`). The structure of of custom property-list files is completely up to you. You can use property lists to store string, number, Boolean, date, and raw data values. By default, a property list stores data in a single dictionary structure, but you can assign additional dictionaries and arrays as values to create a more hierarchical data set.

For information about using property lists, see *Property List Programming Guide* and *Property List Programming Topics for Core Foundation*.

Other Resource Files

In addition to the resource types listed in the preceding sections, Table 1-1 lists some additional resource file types you might find in an application bundle.

Table 1-1   Other resource types

Resource Type	Description
AppleScript files	In Mac OS X, AppleScript terminology and suite files contain information about the scriptability of an application. These files can use the file extensions `.sdef`, `.scriptSuite`, or `.scriptTerminology`. Because the actual AppleScript commands used to script an application are visible in user scripts and the Script Editor application, these resources need to be localized. For information on supporting AppleScript, see *AppleScript Overview*.
Help files	In Mac OS X, help content typically consists of a set of HTML files created using a standard text-editing program and registered with the Help Viewer application. (For information on how to register with Help Viewer, see *Apple Help Programming Guide*.) It is also possible to embed PDF files, RTF files, HTML files or other custom documents in your bundle and open them using an external application, such as Preview or Safari. For information on how to open files, see *Launch Services Programming Guide*.
Custom files	Your application can put custom data files or templates inside the appropriate bundle directories.

Legacy Resource Files

In earlier versions of Mac OS, resources were stored in files that used a `.rsrc` extension. These files were capable of storing multiple resources, including images, sounds, user-interface content, configuration data, and many others. Although support for these files is still available in Mac OS X through the Resource Manager routines, their use is deprecated and strongly discouraged. Old-style resource files are usually holdovers from Carbon applications that were ported from Mac OS 9. Even if you are just now porting such an application to Mac OS X, it still makes much more sense to replace your old resources with new ones. Improvements in Mac OS X, particularly in the area of high-resolution graphics, mean that graphics resources found in these files would look out of place in the current system. In addition, most of the other resource types are obsolete or can be easily replaced by property lists and other resource types.

If you have legacy code and need to know how to access old-style resource files, see *Resource Manager Reference*.

Nib Files

Nib files play an important role in the creation of applications in Mac OS X and iPhone OS. Nib files let you create and manipulate your user interfaces graphically, using the Interface Builder application, instead of programmatically. Because you can see the results of your changes instantly, this gives you the ability to experiment with different layouts and configurations very quickly. It also gives you the flexibility to change many aspects of your user interface later without rewriting any code.

For applications built using the AppKit or UIKit frameworks, nib files take on an extra significance. Both of these frameworks support the use of nib files both for the visual layout of windows, views, and controls and also for the integration of those items with the application's event handling code. Interface Builder works in conjunction with these frameworks, and with Xcode, to help you connect the controls of your user interface to the objects in your project that respond to those controls. This integration significantly reduces the amount of setup that is required after a nib file is loaded and also makes it easy to change the relationships between your code and user interface later.

Because the level of support for nib files is more extensive in Cocoa applications than it is for Carbon applications, the following sections focus on the use of nib files with the AppKit and UIKit frameworks. Although Carbon applications can also use nib files, they do so in a very different way than both AppKit and UIKit, and so the use of nib files in Carbon applications is discussed separately in "Carbon Resources" (page 47).

> **Note:** Although you can create an Objective-C application without using nib files, doing so is very rare and not recommended. Depending on your application, avoiding the use of nib files can involve overriding large amounts of framework behavior to achieve the same results you would get using a nib file.

Anatomy of a Nib File

Nib files are the documents produced by the Interface Builder application. A nib file describes the visual elements of your application's user interface, including windows, views, controls, and many others. It can also describe non-visual elements, such as the objects in your application that manage your windows and views.Most importantly, a nib file describes these objects exactly as they were configured in Interface Builder. At runtime, these descriptions are used to recreate the objects and their configuration inside your application. When you load a nib file at runtime, you get an exact replica of the objects that were in your Interface Builder document. The nib-loading code instantiates the objects reestablishes all of the properties and connections that were present between the objects in Interface Builder.

The following sections describe how nib files used with the AppKit and UIKit frameworks are organized, the types of objects found in them, and how you use those objects effectively.

About Your Interface Objects

Interface objects are what you add to an Interface Builder document to implement your user interface. When a nib is loaded at runtime, the interface objects are the objects actually instantiated by the nib-loading code. Most new documents in Interface Builder have at least one interface object by default, typically a window or menu resource, and you add more interface objects to a nib file as part of your interface design. This is the most common type of object in a nib file and is typically why you create nib files in the first place.

Besides representing visual objects, such as windows, views, controls, and menus, interface objects can also represent non-visual objects. In nearly all cases, the non-visual objects you add to a nib file are extra controller objects that your application uses to manage the visual objects. Although you could create these objects in your application, it is often more convenient to add them to a nib file and configure them there. Interface Builder provides a generic object that you use specifically when adding controllers and other non-visual objects to a nib file. It also provides the controller objects that are typically used to manage Cocoa bindings.

About the File's Owner

One of the most important objects in a nib file is the File's Owner object. Unlike interface objects, the File's Owner object is a proxy object that is not created when the nib file is loaded. Instead, you create this object in your code and pass it to the nib-loading code. The reason this object is so important is that it is the main link between your application code and the contents of the nib file. More specifically, it is the controller object that is responsible for the contents of the nib file.

In Interface Builder, you can create connections between the File's Owner and the other interface objects in your nib file. When you load the nib file, the nib-loading code recreates these connections using the replacement object you specify. This allows your object to reference objects in the nib file and receive messages from the interface objects automatically.

About the First Responder

In Interface Builder, the First Responder is a proxy object that represents the first object in your application's dynamically determined responder chain. Because the responder chain of an application cannot be determined at design time, the First Responder proxy acts as a stand-in target for any action messages that need to be directed at the application's responder chain. Menu items commonly target the First Responder proxy. For example, the Minimize menu item in the Window menu hides the frontmost window in an application, not just a specific window, and the Copy menu item should copy the current selection, not just the selection of a single control or view. Other objects in your application can target the First Responder as well.

When you load a nib file into memory, there is nothing you have to do to manage or replace the First Responder proxy object. The AppKit and UIKit frameworks automatically set and maintain the first responder based on the application's current configuration.

For more information about the responder chain and how it is used to dispatch events in AppKit–based applications, see Event Architecture in*Cocoa Event-Handling Guide*. For information about the responder chains and handling actions in iPhone applications, see *iPhone Application Programming Guide*.

About the Top-Level Objects

When your program loads a nib file, Cocoa recreates the entire graph of objects you created in Interface Builder. This object graph includes all of the windows, views, controls, cells, menus, and custom objects found in the nib file. The **top-level objects** are the subset of these objects that do not have a parent object. The top-level objects typically include only the windows, menubars, and custom controller objects that you add to the nib file. (Objects such as File's Owner, First Responder, and Application are proxy objects and not considered top-level objects.) In Interface Builder, you can see the top-level objects in the nib document window when it is set to icon mode (see the circled items in Figure 2-1).

Figure 2-1   Top-level objects in a nib file

About Image and Sound Resources

In Interface Builder, you can associate external image and sound resources with the contents of your nib files. Some controls and views are able to display images or play sounds as part of their default configuration. The Interface Builder library provides access to the image and sound resources of your Xcode projects so that you can link your nib files to these resources. The nib file does not store these resources directly. Instead, it stores the name of the resource file so that the nib-loading code can find it later.

Typically, you use outlets in the File's Owner object to store references to the top-level objects of a nib file. If you do not use outlets, however, you can retrieve the top-level objects from the nib-loading routines directly. You should always keep a pointer to these objects somewhere because your application is responsible for releasing them when it is through using them. For more information about the nib object behavior at load time, see "Nib Object Retention" (page 19).

When you load a nib file that contains references to image or sound resources, the nib-loading code also reads in those resource files and caches them in memory. In Mac OS X, image and sound resources are stored in named caches so that you can access them later if needed. In iPhone OS, only image resources are stored in named caches. To access images, you use the `imageNamed:` method of `NSImage` or `UIImage`, depending on your platform. To access cached sounds in Mac OS X, use the `soundNamed:` method of `NSSound`.

Nib File Design Guidelines

When creating your nib files, it is important to think carefully about how you intend to use the objects in that file. A very simple application might be able to store all of its user interface components in a single nib file, but for most applications, it is better to distribute components across multiple nib files. Creating smaller nib files lets you load only those portions of your interface that you need immediately. Smaller nib files results in better performance for your application. They also make it easier to debug any problems you might encounter, since there are fewer places to look for problems.

When creating your nib files, try to keep the following guidelines in mind:

- Design your nib files with lazy loading in mind. Plan on loading nib files that contain only those objects you need right away.

- In the main nib file for a Mac OS X application, consider storing only the application menu bar and an optional application delegate object in the nib file. Avoid including any windows or user-interface elements that will not be used until after the application has launched. Instead, place those resources in separate nib files and load them as needed after launch.

- Store repeated user-interface components (such as document windows) in separate nib files.

- For a window or menu that is used only occasionally, store it in a separate nib file. By storing it in a separate nib file, you load the resource into memory only if it is actually used.

- Make the File's Owner the single point-of-contact for anything outside of the nib file; see "Accessing the Contents of a Nib File" (page 27).

The Nib Object Life Cycle

When a nib file is loaded into memory, the nib-loading code takes several steps to that ensure the objects in the nib file are created and initialized properly. Understanding these steps can help you write better controller code to manage your user interfaces.

The Object Loading Process

When you use the methods of `NSNib` or `NSBundle` to load and instantiate the objects in a nib file, the underlying nib-loading code does the following:

1. It loads the contents of the nib file and any referenced resource files into memory:

    - The raw data for the entire nib object graph is loaded into memory but is not unarchived.

- Any custom image resources associated with the nib file are loaded and added to the Cocoa image cache; see "About Image and Sound Resources" (page 15).

- Any custom sound resources associated with the nib file are loaded and added to the Cocoa sound cache; see "About Image and Sound Resources" (page 15).

2. It unarchives the nib object graph data and instantiates the objects. How it initializes each new object depends on the type of the object and how it was encoded in the archive by Interface Builder. The nib-loading code uses the following rules (in order) to determine which initialization method to use.

    a. Standard Interface Builder objects (and custom subclasses of those objects) receive an `initWithCoder:` message.

        In Mac OS X, the list of standard objects includes the views, cells, menus, and view controllers that are provided by the system and available in the default Interface Builder library. It also includes any third-party objects that were added to the Interface Builder library using a custom plug-in. Even if you change the class of such an object, Interface Builder encodes the standard object into the nib file and then tells the archiver to swap in your custom class when the object is unarchived.

        In iPhone OS, any object that conforms to the `NSCoding` protocol is initialized using the `initWithCoder:` method. This includes all subclasses of `UIView` and `UIViewController` whether they are part of the default Interface Builder library or custom classes you define.

    b. Custom views in Mac OS X receive an `initWithFrame:` message.

        Custom views are subclasses of `NSView` for which Interface Builder does not have an available implementation. Typically, these are views that you define in your application and use to provide custom visual content. Custom views do not include standard system views (like `NSSlider`) that are part of the default Interface Builder library or part of an integrated third-party plug-in.

        When it encounters a custom view, Interface Builder encodes a special `NSCustomView` object into your nib file. The custom view object includes the information it needs to build the real view subclass you specified. At load time, the `NSCustomView` object sends an `alloc` and `initWithFrame:` message to the real view class and then swaps the resulting view object in for itself. The net effect is that the real view object handles subsequent interactions during the nib-loading process.

        Custom views in iPhone OS do not use the `initWithFrame:` method for initialization.

    c. Custom objects other than those described in the preceding steps receive an `init` message.

3. It reestablishes all connections (actions, outlets, and bindings) between objects in the nib file. This includes connections to File's Owner and other proxy objects. The approach for establishing connections differs depending on the platform:

    - Outlet connections

        ❑ In Mac OS X, the nib-loading code tries to reconnect outlets using the object's own methods first. For each outlet, Cocoa looks for a method of the form set*OutletName:* and calls it if such a method is present. If it cannot find such a method, Cocoa searches the object for an instance variable with the corresponding outlet name and tries to set the value directly. If the instance variable cannot be found, no connection is created.

In Mac OS X v10.5 and later, setting an outlet also generates a key-value observing (KVO) notification for any registered observers. These notifications may occur before all inter-object connections are reestablished and definitely occur before any `awakeFromNib` methods of the objects have been called. Prior to v10.5, these notifications are not generated. For more information about KVO notifications, see *Key-Value Observing Programming Guide*.

❑ In iPhone OS, the nib-loading code uses the `setValue:forKey:` method to reconnect each outlet. That method similarly looks for an appropriate accessor method and falls back on other means when that fails. For more information about how this method sets values, see its description in *NSKeyValueCoding Protocol Reference*.

Setting an outlet in iPhone OS also generates a KVO notification for any registered observers. These notifications may occur before all inter-object connections are reestablished and definitely occur before any `awakeFromNib` methods of the objects have been called. For more information about KVO notifications, see *Key-Value Observing Programming Guide*.

- Action connections

  ❑ In Mac OS X, the nib-loading code uses the source object's `setTarget:` and `setAction:` methods to establish the connection to the target object. If the target object does not respond to the action method, no connection is created. If the target object is `nil`, the action is handled by the responder chain.

  ❑ In iPhone OS, the nib-loading code uses the `addTarget:action:forControlEvents:` method of the `UIControl` object to configure the action. If the target is `nil`, the action is handled by the responder chain.

- Bindings

  ❑ In Mac OS X, Cocoa uses the `bind:toObject:withKeyPath:options:` method of the source object to create the connection between it and its target object.

  ❑ Bindings are not supported in iPhone OS.

4. It sends an `awakeFromNib` message to the appropriate objects in the nib file that define the matching selector:

- In Mac OS X, this message is sent to any interface objects that define the method. It is also sent to the File's Owner and any proxy objects that define it as well.

- In iPhone OS, this message is sent only to the interface objects that were instantiated by the nib-loading code. It is not sent to File's Owner, First Responder, or any other proxy objects.

5. It displays any windows whose "Visible at launch time" attribute was enabled in Interface Builder.

The order in which the nib-loading code calls the `awakeFromNib` methods of objects is not guaranteed. In Mac OS X, Cocoa tries to call the `awakeFromNib` method of File's Owner last but does not guarantee that behavior. If you need to configure the objects in your nib file further at load time, the most appropriate time to do so is after your nib-loading call returns. At that point, all of the objects are created, initialized, and ready for use.

Nib Object Retention

Each time you ask the `NSBundle` or `NSNib` class to load a nib file, the underlying code creates a new copy of the objects in that file and returns them to you. The nib-loading code does not recycle nib file objects from a previous load attempt. Because each set of objects is a new copy, your code is responsible for releasing those objects when it is done with them. How you release the objects depends on the platform and on the memory model in use. Table 2-1 lists the supported platform and memory model configurations and the nib retention behavior associated with each one.

Table 2-1   Object retention rules for nib objects

Configuration	Description
Mac OS X - managed memory model	Objects in the nib file are initially created with a retain count of 1. As it rebuilds the object hierarchy, however, AppKit autoreleases any objects that have a parent or owning object, such as views nested inside view hierarchies. By the time the nib-loading code is done, only the top-level objects in the nib file have a positive retain count and no owning object. Your code is responsible for releasing these top-level objects.
Mac OS X - garbage collected memory model	Most objects in the graph are kept in memory through strong references between the objects. Only the top-level objects in the nib file do not have strong references initially. Thus, your code must create strong references to these objects to prevent the object graph from being released.
iPhone OS - managed memory model	Objects in the nib file are created with a retain count of 1 and then autoreleased. As it rebuilds the object hierarchy, however, UIKit reestablishes connections between the objects using the `setValue:forKey:` method, which uses the available setter method or retains the object by default if no setter method is available. If you define outlets for nib-file objects, you should also define a setter method for accessing that outlet. Setter methods for outlets should retain their values, and setter methods for outlets containing top-level objects must retain their values to prevent them from being deallocated. If you do not store the top-level objects in outlets, you must retain either the array returned by the `loadNibNamed:owner: options:` method or the objects inside the array to prevent those objects from being released prematurely.

For both Mac OS X and UIKit, the recommended way to manage the top-level objects in a nib file is to create outlets for them in the File's Owner object and then define setter methods to retain and release those objects as needed. Setter methods give you an appropriate place to include your memory-management code, even in situations where your application uses garbage collection. One easy way to implement your setter methods is to use the `@property` syntax and let the compiler create them for you. For more information on how to define properties, see *The Objective-C 2.0 Programming Language*.

Built-In Support For Nib Files

The AppKit and UIKit frameworks both provide a certain amount of automated behavior for loading and managing nib files in an application. Both frameworks provide infrastructure for loading an application's main nib file. In addition, the AppKit framework provides support for loading other nib files through the `NSDocument` and `NSWindowController` classes. The following sections describe the built-in support for nib files, how you can take advantage of it, and ways to modify that support in your own applications.

Loading the Main Nib File

Most of the Xcode project templates for applications come preconfigured with a main nib file already in place. All you have to do is modify this default nib file in Interface Builder and build your application. At launch time, the application's default configuration data tells the application object where to find this nib file so that it can load it. In applications based on either AppKit and UIKit, this configuration data is located in the application's `Info.plist` file. When an application is first loaded, the default application startup code looks in the `Info.plist` file for the `NSMainNibFile` key. If it finds it, it looks in the application bundle for a nib file whose name (with or without the filename extension) matches the value of that key and loads it.

Document and Window Controller Nib Files

In the AppKit framework, the `NSDocument` class works with the default window controller to load the nib file containing your document window. The `windowNibName` method of `NSDocument` is a convenience method that you can use to specify the nib file containing the corresponding document window. When a new document is created, the document object passes the nib file name you specify to the default window controller object, which loads and manages the contents of the nib file. If you use the standard templates provided by Xcode, the only thing you have to do is add the contents of your document window to the nib file.

The `NSWindowController` class also provides automatic support for loading nib files. If you create custom window controllers programmatically, you have the option of initializing them with an `NSWindow` object or with the name of a nib file. If you choose the latter option, the `NSWindowController` class automatically loads the specified nib file the first time a client tries to access the window. After that, the window controller keeps the window around in memory; it does not reload it from the nib file, even if the window's "Release when closed" attribute is set in Interface Builder.

> **Important:** When using either `NSWindowController` or `NSDocument` to load windows automatically, it is important that your nib file be configured correctly. Both classes include a `window` outlet that you must connect to the window you want them to manage. If you do not connect this outlet to a window object, the nib file is loaded but the document or window controller does not display any windows. For more information about the Cocoa document architecture, see *Document-Based Applications Overview*.

Changing the Nib Files from the Xcode Defaults

When you create a new Cocoa application project in Xcode, the project template comes preconfigured with one or more nib files. If you have existing nib files you want to use instead, you can replace the template nib files with your custom nib files. Changing the nib file associated with a document object is easy but changing the application's main nib file is somewhat more involved.

Each document in a Cocoa document-based application has its own nib file for storing the document window and any supporting objects. The `NSDocument` class finds this nib file by calling its own `windowNibName` method, the declaration for which is included in the default document class that comes with the Xcode template. To change the nib file associated with the document, simply change the string returned by the `windowNibName` method of your document class.

The main nib file is the only nib file that is required in a Cocoa application. It is loaded immediately before the application enters its main event loop, and the File's Owner for this nib file is the `NSApplication` object itself. A typical main nib contains only the application menu bar and perhaps an application delegate object to handle any application-related events (such as launch-time notifications). Although many applications include other objects in this nib file, doing so is generally not recommended. Instead, it is always preferable to lazily load other resources only as they are needed. Extra objects consume more memory and require more time to load from disk, both of which can degrade launch-time performance.

In Xcode, every new Cocoa project comes configured with a main nib file, called `MainMenu.xib`, that contains a default menu bar for your project. To change the main nib file for your Cocoa application, do the following:

1. In Interface Builder, open the nib file that you want to make the main nib.

2. Select the File's Owner object of the nib and open the identity inspector.

3. In the identity inspector, set the class of File's Owner to `NSApplication` (or to your custom subclass of `NSApplication` if you define one).

4. Open your application project in Xcode.

5. In the Targets section of the Groups & Files pane, select your application target.

6. Open an inspector (or Info) window for the target and select the Properties tab.

7. In the Main Nib File field, enter the name of your new nib file.

The preceding set of steps also work if you want to change the main nib file for an iPhone application. Instead of the `NSApplication` class, set the class of the File's Owner proxy to `UIApplication`. In addition, you might also want to add a custom object and connect it to the delegate outlet of the `UIApplication`. (You could create the delegate object programmatically from the `UIApplicationMain` function but creating it in the nib file is much more common.)

Loading Nib Files Programmatically

Both Mac OS X and iPhone OS provide convenience methods for loading nib files into your application. Both the AppKit and UIKit framework define additional methods on the NSBundle class that support the loading of nib files. In addition, the AppKit framework also provides the `NSNib` class, which provides similar nib-loading behavior as `NSBundle` but offers some additional advantages that might be useful in specific situations.

As you plan out your application, make sure any nib files you plan to load manually are configured in a way that simplifies the loading process. Choosing an appropriate object for File's Owner and keeping your nib files small can greatly improve their ease of use and memory efficiency. For more tips on configuring your nib files, see "Nib File Design Guidelines" (page 16).

Loading Nib Files Using NSBundle

The AppKit and UIKit frameworks define additional methods on the `NSBundle` class (using Objective-C categories) to support the loading of nib file resources. The semantics for how you use the methods differs between the two platforms as does the syntax for the methods. In AppKit, there are more options for accessing bundles in general and so there are correspondingly more methods for loading nib files from those bundles. In UIKit, applications can load nib files only from their main bundle and so fewer options are needed. The methods available on the two platforms are as follows:

- AppKit

  - `loadNibNamed:owner:` class method

  - `loadNibFile:externalNameTable:withZone:` class method

  - `loadNibFile:externalNameTable:withZone:` instance method

- UIKit

  - `loadNibNamed:owner:options:` instance method

Whenever loading a nib file, you should always specify an object to act as File's Owner of that nib file. The role of the File's Owner is an important one. It is the primary interface between your running code and the new objects that are about to created in memory. All of the nib-loading methods provide a way to specify the File's Owner, either directly or as a parameter in an options dictionary.

One of the semantic differences between the way the AppKit and UIKit frameworks handle nib loading is the way the top-level nib objects are returned to your application. In the AppKit framework, you must explicitly request them using one of the `loadNibFile:externalNameTable:withZone:` methods. In UIKit, the `loadNibNamed:owner:options:` method returns an array of these objects directly. The simplest way to avoid having to worry about the top-level objects in either case is to store them in outlets of your File's Owner object and to make sure the setter methods for those outlets retain their values. Because each platform uses different retain semantics, however, you must be sure to send the proper retain or release messages when appropriate. For information about the retention semantics for nib objects, see "Nib Object Retention" (page 19).

Listing 2-1 shows a simple example of how to load a nib file using the `NSBundle` class in an AppKit–based application. As soon as the `loadNibNamed:owner:` method returns, you can begin using any outlets that refer to the nib file objects. In other words, the entire nib-loading process occurs within the confines of that single call. The nib-loading methods in the AppKit framework return a Boolean value to indicate whether the load operation was successful.

Listing 2-1   Loading a nib file from the current bundle

```
- (BOOL)loadMyNibFile
{
    // The myNib file must be in the bundle that defines self's class.
    if (![NSBundle loadNibNamed:@"myNib" owner:self])
    {
        NSLog(@"Warning! Could not load myNib file.\n");
        return NO;
    }
    return YES;
}
```

Listing 2-2 shows an example of how to load a nib file in a UIKit–based application. In this case, the method checks the returned array to see if the nib objects were loaded successfully. (Every nib file should have at least one top-level object representing the contents of the nib file.) This example shows the simple case when the nib file contains no proxy objects other than the File's Owner object. For an example of how to specify additional proxy objects, see "Replacing Proxy Objects at Load Time" (page 26).

Listing 2-2   Loading a nib in an iPhone application

```
- (BOOL)loadMyNibFile
{
    NSArray*    topLevelObjs = nil;

    topLevelObjs = [[NSBundle mainBundle] loadNibNamed:@"myNib" owner:self
options:nil];
    if (topLevelObjs == nil)
    {
        NSLog(@"Error! Could not load myNib file.\n");
        return NO;
    }
    return YES;
}
```

Getting a Nib File's Top-Level Objects

The easiest way to get the top-level objects of your nib file is to define outlets in the File's Owner object along with setter methods (or better yet, properties) for accessing those objects. This approach ensures that the top-level objects are retained by your object and that you always have references to them.

Listing 2-3 shows the interface and implementation of a stripped down Cocoa class that uses an outlet to retain the nib file's only top-level object. In this case, the only top-level object in the nib file is an `NSWindow` object. Because top-level objects in Cocoa have an initial retain count of 1, an extra release message is included. This is fine because by the time the release call is made, the property has already retained the window. You would not want to release top-level objects in this manner in an iPhone application.

Listing 2-3   Using outlets to get the top-level objects

```
// Class interface
@interface MyController : NSObject {
    NSWindow *window;
}
@property(retain) IBOutlet NSWindow *window;
- (void)loadMyWindow;

@end

// Class implementation
@implementation MyController
// The synthesized property retains the window automatically.
@synthesize window;

- (void)loadMyWindow
{
    [NSBundle loadNibNamed:@"myNib" owner:self];

    // The window starts off with a retain count of 1
    // and is then retained by the property, so add an extra release.
    [window release];
}
@end
```

If you do not want to use outlets to store references to your nib file's top-level objects, you must retrieve those objects manually in your code. The technique for obtaining the top-level objects differs depending on the target platform. In Mac OS X, you must ask for the objects explicitly, whereas in iPhone OS they are returned to you automatically.

Listing 2-4 shows the process for getting the top-level objects of a nib file in Mac OS X. This method places a mutable array into the nameTable dictionary and associates it with the NSNibTopLevelObjects key. The nib-loading code looks for this array object and, if present, places the top-level objects in it. Because each object starts with a retain count of 1 before it is added to the array, simply releasing the array is not enough to release the objects in the array as well. As a result, this method sends a release message to each of the objects to ensure that the array is the only entity holding a reference to them.

Listing 2-4   Getting the top-level objects from a nib file at runtime

```
- (NSArray*)loadMyNibFile
{
    NSBundle*           aBundle = [NSBundle mainBundle];
    NSMutableArray*     topLevelObjs = [NSMutableArray array];
    NSDictionary*       nameTable = [NSDictionary dictionaryWithObjectsAndKeys:
                                        self, NSNibOwner,
                                        topLevelObjs, NSNibTopLevelObjects,
                                        nil];

    if (![aBundle loadNibFile:@"myNib" externalNameTable:nameTable withZone:nil])
    {
        NSLog(@"Warning! Could not load myNib file.\n");
        return nil;
    }

    // Release the objects so that they are just owned by the array.
    [topLevelObjs makeObjectsPerformSelector:@selector(release)];
```

```
    return topLevelObjs;
}
```

Obtaining the top-level objects in an iPhone application is much simpler and is shown in Listing 2-2 (page 23). In the UIKit framework, the `loadNibNamed:owner:options:` method of `NSBundle` automatically returns an array with the top-level objects. In addition, by the time the array is returned, the retain counts on the objects are adjusted so that you do not need to send each object an extra release message. The returned array is the only owner of the objects.

Loading Nib Files Using NSNib

In Mac OS X, the AppKit framework supports the loading of nib files using the `NSNib` class. You can use the `NSNib` class to load nib files that reside outside of a bundle or in situations where you plan to load the same nib file more than once. Loading a nib file with this class is always a two-step process. First, you create an instance of the `NSNib` class, initializing it with the nib file's location information. Second, you instantiate the contents of the nib file to load the objects into memory. Each time you instantiate the nib file, you specify a different File's Owner object and can also receive a new set of top-level objects.

> **Note:** The `NSNib` class is not available in iPhone OS. You should use the `NSBundle` class to load nib files instead.

When you want to load the same nib file multiple times, the two-step process used by `NSNib` offers some advantages over the `NSBundle` methods. An `NSNib` object loads the nib data into memory only once, but each time you call one of its instantiate methods, you receive a unique copy of the nib file objects. If you need to create several copies of a nib file's objects in quick succession, the caching provided by `NSNib` can improve performance.

Listing 2-5 shows one way to load the contents of a nib file using the `NSNib` class. The array returned to you by the `instantiateNibWithOwner:topLevelObjects:` method comes already autoreleased. If you intend to use that array for any period of time, you should make a copy of it.

Listing 2-5   Loading a nib file using NSNib

```
- (NSArray*)loadMyNibFile
{
    NSNib*      aNib = [[NSNib alloc] initWithNibNamed:@"MyPanel" bundle:nil];
    NSArray*    topLevelObjs = nil;

    if (![aNib instantiateNibWithOwner:self topLevelObjects:&topLevelObjs])
    {
        NSLog(@"Warning! Could not load nib file.\n");
        return nil;
    }
    // Release the raw nib data.
    [aNib release];

    // Release the top-level objects so that they are just owned by the array.
    [topLevelObjs makeObjectsPerformSelector:@selector(release)];

    // Do not autorelease topLevelObjs.
    return topLevelObjs;
}
```

Replacing Proxy Objects at Load Time

In iPhone OS, it is possible to create nib files that include proxy objects besides the File's Owner. Proxy objects represent objects created outside of the nib file but which have some connection to the nib file's contents. Proxies are commonly used to support navigation controllers in iPhone applications. When working with navigation controllers, you typically connect the File's Owner object to some common object such as your application delegate. Proxy objects therefore represent the parts of the navigation controller object hierarchy that are already loaded in memory, either because they were created programmatically or loaded from a different nib file.

> **Note:** Custom proxy objects (other than File's Owner) are not supported in Mac OS X nib files.

Each proxy object you add to a nib file must have a unique name. To assign a name to an object, select the object in Interface Builder and open the inspector window. The Attributes pane of the inspector contains a Name field, which you use to specify the name for your proxy object. The name you assign should be descriptive of the object's behavior or type, but really it can be anything you want.

When you are ready to load a nib file containing proxy objects, you must specify the replacement objects for any proxies when you call the `loadNibNamed:owner:options:` method. The *options* parameter of this method accepts a dictionary of additional information. You use this dictionary to pass in the information about your proxy objects. The dictionary must contain the `UINibProxiedObjectsKey` key whose value is another dictionary containing the name and object for each proxy replacement.

Listing 2-6 shows a sample version of an `applicationDidFinishLaunching:` method that loads the application's main nib file manually. Because the application's delegate object is created by the `UIApplicationMain` function, this method uses a proxy (with the name "AppDelegate") in the main nib file to represent that object. The proxies dictionary stores the proxy object information and the options dictionary wraps that dictionary.

Listing 2-6   Replacing proxy objects in a nib file

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    NSArray*    topLevelObjs = nil;
    NSDictionary*    proxies = [NSDictionary dictionaryWithObject:self
forKey:@"AppDelegate"];
    NSDictionary*    options = [NSDictionary dictionaryWithObject:proxies
forKey:UINibProxiedObjectsKey];

    topLevelObjs = [[NSBundle mainBundle] loadNibNamed:@"Main" owner:self
options:options];
    if ([topLevelObjs count] == 0)
    {
        NSLog(@"Warning! Could not load myNib file.\n");
        return;
    }

    // Show window
    [window makeKeyAndVisible];
}
```

For more information about the options dictionary of the `loadNibNamed:owner:options:` method, see *NSBundle UIKit Additions Reference*.

Accessing the Contents of a Nib File

Upon the successful loading a nib file, its contents become ready for you to use immediately. If you configured outlets in your File's Owner to point to nib file objects, you can now use those outlets. If you did not configure your File's Owner with any outlets, you should make sure you obtain a reference to the top-level objects in some manner so that you can release them later.

Because outlets are populated with real objects when a nib file is loaded, you can subsequently use outlets as you would any other object you created programmatically. For example, if you have an outlet pointing to a window, you could send that window a `makeKeyAndOrderFront:` message to show it on the user's screen. When you are done using the objects in your nib file, you must release them like any other objects.

> **Important:** You are responsible for releasing the top-level objects of any nib files you load when you are finished with those objects. Failure to do so is a cause of memory leaks in many applications. After releasing the top-level objects, it is a good idea to clear any outlets pointing to objects in the nib file by setting them to `nil`. You should clear outlets associated with all of the nib file's objects, not just the top-level objects.

Connecting Menu Items Across Nib Files

The items in a Mac OS X application's menu bar often need to interact with many different objects, including your application's documents and windows. The problem is that many of these objects cannot (or should not) be accessed directly from the main nib file. The File's Owner of the main nib file is always set to an instance of the `NSApplication` class. And although you might be able to instantiate a number of custom objects in your main nib file, doing so is hardly practical or necessary. In the case of document objects, connecting directly to a specific document object is not even possible because the number of document objects can change dynamically and can even be zero.

Most menu items send action messages to one of the following:

- A fixed object that always handles the command
- A dynamic object, such as a document or window

Messaging fixed objects is a relatively straightforward process that is usually best handled through the application delegate. The application delegate object assists the `NSApplication` object in running the application and is one of the few objects that rightfully belongs in the main nib file. If the menu item refers to an application-level command, you can implement that command directly in the application delegate or just have the delegate forward the message to the appropriate object elsewhere in your application.

If you have a menu item that acts on the contents of the frontmost window, you need to link the menu item to the First Responder proxy object. If the action method associated with the menu item is specific to one of your objects (and not defined by Cocoa), you must add that action to the First Responder before creating the connection. To set up a First Responder connection in Interface Builder v3.0 and later, do the following:

1. For custom actions, add the action to the First Responder proxy object:

    a. In your main nib file, select the First Responder proxy object.

    b. Open the inspector window and select the Identity pane.

    c.   In the Class Actions section, click the plus (+) button to add the new action method. The new action's name is initially selected.

    d.   Enter the name of your action and press Return.

2.   Open the menu bar resource.

3.   If you have not already done so, add your menu item to the desired menu.

4.   Control-click the menu item and drag it to the First Responder proxy object in the nib document window.

5.   In the Connections tab of the inspector window, select your action and click Connect.

After creating the connection, you need to implement the action method in your `NSDocument` or `NSResponder` subclass. That object should also implement the `validateMenuItem:` method to enable the menu item at appropriate times. For more information about how the responder chain handles commands, see *Cocoa Event-Handling Guide*.

String Resources

An important part of the localization process is to localize all of the text strings displayed by your application. By their nature, strings located in nib files can be readily localized along with the rest of the nib file contents. Strings embedded in your code, however, must be extracted, localized, and then reinserted back into your code. To simplify this process—and to make the maintenance of your code easier—Mac OS X and iPhone OS provide the infrastructure needed to separate strings from your code and place them into resource files where they can be localized easily.

Resource files that contain localizable strings are referred to as **strings** files because of their filename extension, which is `.strings`. You can create strings files manually or programmatically depending on your needs. The standard strings file format consists of one or more key-value pairs along with optional comments. The key and value in a given pair are strings of text enclosed in double quotation marks and separated by an equal sign. (You can also use a property list format for strings files. In such a case, the top-level node is a dictionary and each key-value pair of that dictionary is a string entry.)

Listing 3-1 shows a simple strings file that contains non-localized entries for the default language. When you need to display a string, you pass the string on the left to one of the available string-loading routines. What you get back is the matching value string containing the text translation that is most appropriate for the current user. For the development language, it is common to use the same string for both the key and value, but doing so is not required.

Listing 3-1   A simple strings file

```
/* Insert Element menu item */
"Insert Element" = "Insert Element";
/* Error string used for unknown error types. */
"ErrorString_1" = "An unknown error occurred.";
```

A typical application has at least one strings file per localization, that is, one strings file in each of the bundle's `.lproj` subdirectories. The name of the default strings file is `Localizable.strings` but you can create strings files with any file name you choose. Creating strings files is discussed in more depth in "Creating Strings Resource Files" (page 30).

> **Note:** It is recommended that you save strings files using the UTF-16 encoding, which is the default encoding for standard strings files. It is possible to create strings files using other property-list formats, including binary property-list formats and XML formats that use the UTF-8 encoding, but doing so is not recommended. For more information about the standard strings file format, see "Creating Strings Resource Files" (page 30). For more information about Unicode and its text encodings, go to http://www.unicode.org/ or http://en.wikipedia.org/wiki/Unicode.

The loading of string resources (both localized and nonlocalized) ultimately relies on the bundle and internationalization support found in both Mac OS X and iPhone OS. For information about bundles, see *Bundle Programming Guide*. For more information about internationalization and localization, see *Internationalization Programming Topics*.

Creating Strings Resource Files

Although you can create strings files manually, it is rarely necessary to do so. The easiest way to create strings files is to write your code using the appropriate string-loading macros and then use the `genstrings` command-line tool to extract those strings and create strings files for you.

The following sections describe the process of how to set up your source files to facilitate the use of the `genstrings` tool. For detailed information about the tool, see `genstrings` man page.

Choosing Which Strings to Localize

When it comes to localizing your application's interface, it is not always appropriate to localize every string used by your application. Translation is a costly process, and translating strings that are never seen by the user is a waste of time and money. Strings that are not displayed to the user, such as notification names used internally by your application, do not need to be translated. Consider the following example:

```
if (CFStringHasPrefix(value, CFSTR("-")) {     CFArrayAppendValue(myArray,
value);}};
```

In this example, the string "-" is used internally and is never seen by the user; therefore, it does not need to be placed in a strings file.

The following code shows another example of a string the user would not see. The string "%d %d %s" does not need to be localized, since the user never sees it and it has no effect on anything that the user does see.

```
matches = sscanf(s, "%d %d %s", &first, &last, &other);
```

Because nib files are localized separately, you do not need to include strings that are already located inside of a nib file. Some of the strings you should localize, however, include the following:

■ Strings that are programmatically added to a window, panel, view, or control and subsequently displayed to the user. This includes strings you pass into standard routines, such as those that display alert boxes.

■ Menu item title strings if those strings are added programmatically. For example, if you use custom strings for the Undo menu item, those strings should be in a strings file.

■ Error messages that are displayed to the user.

■ Any boilerplate text that is displayed to the user.

■ Some strings from your application's information property list (`Info.plist`) file; see *Runtime Configuration Guidelines*.

■ New file and document names.

About the String-Loading Macros

The Foundation and Core Foundation frameworks define the following macros to make loading strings from a strings file easier:

■ Core Foundation macros:

   ❑ `CFCopyLocalizedString`

- ❑ `CFCopyLocalizedStringFromTable`

- ❑ `CFCopyLocalizedStringFromTableInBundle`

- ❑ `CFCopyLocalizedStringWithDefaultValue`

- ■ Foundation macros:

  - ❑ `NSLocalizedString`

  - ❑ `NSLocalizedStringFromTable`

  - ❑ `NSLocalizedStringFromTableInBundle`

  - ❑ `NSLocalizedStringWithDefaultValue`

You use these macros in your source code to load strings from one of your application's strings files. The macros take the user's current language preferences into account when retrieving the actual string value. In addition, the `genstrings` tool searches for these macros and uses the information they contain to build the initial set of strings files for your application.

For detailed information about how to use these macros, see "Loading String Resources Into Your Code" (page 33).

Using the Genstrings Tool to Create Strings Files

At some point during your development, you need to create the strings files needed by your code. If you wrote your code using the Core Foundation and Foundation macros, the simplest way to create your strings files is using the `genstrings` command-line tool. You can use this tool to generate a new set of strings files or update a set of existing files based on your source code.

To use the `genstrings` tool, you typically provide at least two arguments:

- ■ A list of source files

- ■ An optional output directory

The `genstrings` tool can parse C, Objective-C, and Java code files with the `.c`, `.m`, or `.java` filename extensions. Although not strictly required, specifying an output directory is recommended and is where `genstrings` places the resulting strings files. In most cases, you would want to specify the directory containing the project resources for your development language.

The following example shows a simple command for running the `genstrings` tool. This command causes the tool to parse all Objective-C source files in the current directory and put the resulting strings files in the `en.lproj` subdirectory, which must already exist.

```
genstrings -o en.lproj *.m
```

The first time you run the `genstrings` tool, it creates a set of new strings files for you. Subsequent runs replace the contents of those strings files with the current string entries found in your source code. For subsequent runs, it is a good idea to save a copy of your current strings files before running `genstrings`. You can then diff the new and old versions to determine which strings were added to (or changed in) your project. You can then use this information to update any already localized versions of your strings files, rather than replacing those files and localizing them again.

Within a single strings file, each key must be unique. Fortunately, the `genstrings` tool is smart enough to coalesce any duplicate entries it finds. When it discovers a key string used more than once in a single strings file, the tool merges the comments from the individual entries into one comment string and generates a warning. (You can suppress the duplicate entries warning with the `-q` option.) If the same key string is assigned to strings in different strings files, no warning is generated.

For more information about using the `genstrings` tool, see the `genstrings` man page.

Creating Strings Files Manually

Although the `genstrings` tool is the most convenient way to create strings files, you can also create them manually. To create a strings file manually, create a new file in TextEdit (or your preferred text-editing application) and save it using the Unicode UTF-16 encoding. (When saving files, TextEdit usually chooses an appropriate encoding by default. To force a specific encoding, you must change the save options in the application preferences.) The contents of this file consists of a set of key-value pairs along with optional comments describing the purpose of each key-value pair. Key and value strings are separated by an equal sign, and the entire entry must be terminated with a semicolon character. By convention, comments are enclosed inside C-style comment delimiters (`/*` and `*/`) and are placed immediately before the entry they describe.

Listing 3-2 shows the basic format of a strings file. The entries in this example come from the English version of the `Localizable.strings` file from the TextEdit application. The left side of each equal sign represents the key, and the right side represents the value. A common convention when developing applications is to use a key name that equals the value in the language used to develop the application. Therefore, because TextEdit was developed using the English language, the English version of the `Localizable.strings` file has keys and values that match.

Listing 3-2   Strings localized for English

```
/* Menu item to make the current document plain text */
"Make Plain Text" = "Make Plain Text";
/* Menu item to make the current document rich text */
"Make Rich Text" = "Make Rich Text";
```

Listing 3-3 shows the German translation of the same entries. These entries also live inside a file called `Localizable.strings`, but this version of the file is located in the German language project directory of the TextEdit application. Notice that the keys are still in English, but the values assigned to those keys are in German. This is because the key strings are never seen by end users. They are used by the code to retrieve the corresponding value string, which in this case is in German.

Listing 3-3   Strings localized for German

```
/* Menu item to make the current document plain text */
"Make Plain Text" = "In reinen Text umwandeln";
/* Menu item to make the current document rich text */
"Make Rich Text" = "In formatierten Text umwandeln";
```

Detecting Nonlocalizable Strings

AppKit–based applications can take advantage of built-in support to detect strings that do not need to be localized and those that need to be localized but currently are not. To use this built-in support, you must launch your application from the command line. In addition to entering the path to your executable, you must also include the name of the desired setting along with a Boolean value to indicate whether the setting should be enabled or disabled. The available settings are as follows:

- The `NSShowNonLocalizableStrings` setting identifies strings that are not localizable. The strings are logged to the shell in upper case. This option occasionally generates some false positives but is still useful overall.

- The `NSShowNonLocalizedStrings` setting locates strings that were meant to be localized but could not be found in the application's existing strings files. You can use this setting to catch problems with out-of-date localizations.

For example, to use the `NSShowNonLocalizedStrings` setting with the TextEdit application, you would enter the following in Terminal:

```
/Applications/TextEdit.app/Contents/MacOS/TextEdit -NSShowNonLocalizedStrings
YES
```

Loading String Resources Into Your Code

The Core Foundation and Foundation frameworks provide macros for retrieving both localized and nonlocalized strings stored in strings files. Although the main purpose of these macros is to load strings at runtime, they also serve a secondary purpose by acting as markers that the `genstrings` tool can use to locate your application's string resources. It is this second purpose that explains why many of the macros let you specify much more information than would normally be required for loading a string. The `genstrings` tool uses the information you provide to create or update your application's strings files automatically. Table 3-1 lists the types of information you can specify for these routines and describes how that information is used by the `genstrings` tool.

Table 3-1   Common parameters found in string-loading routines

Parameter	Description
Key	The string used to look up the corresponding value. This string must not contain any characters from the extended ASCII character set, which includes accented versions of ASCII characters. If you want the initial value string to contain extended ASCII characters, use a routine that lets you specify a default value parameter. (For information about the extended ASCII character set, see the corresponding Wikipedia entry.)
Table name	The name of the strings file in which the specified key is located. The `genstrings` tool interprets this parameter as the name of the strings file in which the string should be placed. If no table name is provided, the string is placed in the default `Localizable.strings` file. (When specifying a value for this parameter, include the filename without the `.strings` extension.)

Parameter	Description
Default value	The default value to associate with a given key. If no default value is specified, the `genstrings` tool uses the key string as the initial value. Default value strings may contain extended ASCII characters.
Comment	Translation comments to include with the string. You can use comments to provide clues to the translation team about how a given string is used. The `genstrings` tool puts these comments in the strings file and encloses them in C-style comment delimiters (`/*` and `*/`) immediately above the associated entry.
Bundle	An `NSBundle` object or `CFBundleRef` type corresponding to the bundle containing the strings file. You can use this to load strings from bundles other than your application's main bundle. For example, you might use this to load localized strings from a framework or plug-in.

When you request a string from a strings file, the string that is returned depends on the available localizations (if any). The Cocoa and Core Foundation macros use the built-in bundle support to retrieve the string whose localization matches the user's current language preferences. As long as your localized resource files are placed in the appropriate language-specific project directories, loading a string with these macros should yield the appropriate string automatically. If no appropriate localized string resource is found, the bundle's loading code automatically chooses the appropriate nonlocalized string instead.

For information about internationalization in general and how to create language-specific project directories, see *Internationalization Programming Topics*. For information about the bundle structure and how resource files are chosen from a bundle directory, see *Bundle Programming Guide*.

Using the Core Foundation Framework

The Core Foundation framework defines a single function and several macros for loading localized strings from your application bundle. The `CFBundleCopyLocalizedString` function provides the basic implementation for retrieving the strings. However, it is recommended that you use the following macros instead:

- `CFCopyLocalizedString(key, comment)`
- `CFCopyLocalizedStringFromTable(key, tableName, comment)`
- `CFCopyLocalizedStringFromTableInBundle(key, tableName, bundle, comment)`
- `CFCopyLocalizedStringWithDefaultValue(key, tableName, bundle, value, comment)`

There are several reasons to use the macros instead of the `CFBundleCopyLocalizedString` function. First, the macros are easier to use for certain common cases. Second, the macros let you associate a comment string with the string entry. Third, the macros are recognized by the `genstrings` tool but the `CFBundleCopyLocalizedString` function is not.

For additional information on how to use these macros, see Working With Localized Strings in *Bundle Programming Guide*. For macro and function syntax, see *CFBundle Reference*.

Using the Foundation Framework

The Foundation framework defines a single method and several macros for loading string resources. The `localizedStringForKey:value:table:` method of the `NSBundle` class loads the specified string resource from a strings file residing in the current bundle. Cocoa also defines the following macros for getting localized strings:

- `NSLocalizedString(key, comment)`

- `NSLocalizedStringFromTable(key, tableName, comment)`

- `NSLocalizedStringFromTableInBundle(key, tableName, bundle, comment)`

- `NSLocalizedStringWithDefaultValue(key, tableName, bundle, value, comment)`

As with Core Foundation, Apple recommends that you use the Cocoa convenience macros for loading strings. The main advantage to these macros is that they can be parsed by the `genstrings` tool and used to create your application's strings files. They are also simpler to use and let you associate translation comments with each entry.

For information about the syntax of the preceding macros, see *Foundation Functions Reference*. Additional methods for loading strings are also defined in *NSBundle Class Reference*.

Examples of Getting Strings

The following examples demonstrate the basic techniques for using the Foundation and Core Foundation macros to retrieve strings. Each example assumes that the current bundle contains a strings file with the name `Custom.strings` that has been translated into French. This translated file includes the following strings:

```
/* A comment */
"Yes" = "Oui";
"The same text in English" = "Le même texte en anglais";
```

Using the Foundation framework, you can get the value of the "Yes" string using the `NSLocalizedStringFromTable` macro, as shown in the following example:

```
NSString* theString;
theString = NSLocalizedStringFromTable (@"Yes", @"Custom", @"A comment");
```

Using the Core Foundation framework, you could get the same string using the `CFCopyLocalizedStringFromTable` macro, as shown in this example:

```
CFStringRef theString;
theString = CFCopyLocalizedStringFromTable(CFSTR("Yes"), CFSTR("Custom"), "A
comment");
```

In both examples, the code specifies the key to retrieve, which is the string "Yes". They also specify the strings file (or table) in which to look for the key, which in this case is the `Custom.strings` file. During string retrieval, the comment string is ignored.

Advanced Strings File Tips

The following sections provide some additional tips for working with strings files and string resources.

Searching for Custom Functions With genstrings

The `genstrings` tool searches for the Core Foundation and Foundation string macros by default. It uses the information in these macros to create the string entries in your project's strings files. You can also direct `genstrings` to look for custom string-loading functions in your code and use those functions in addition to the standard macros. You might use custom functions to wrap the built-in string-loading routines and perform some extra processing or you might replace the default string handling behavior with your own custom model.

If you want to use `genstrings` with your own custom functions, your functions must use the naming and formatting conventions used by the Foundation macros. The parameters for your functions must match the parameters for the corresponding macros exactly. When you invoke genstrings, you specify the `-s` option followed by the name of the function that corresponds to the `NSLocalizedString` macro. Your other function names should then build from this base name. For example, if you specified the function name `MyStringFunction`, your other function names should be `MyStringFunctionFromTable`, `MyStringFunctionFromTableInBundle`, and `MyStringFunctionWithDefaultValue`. The `genstrings` tool looks for these functions and uses them to build the corresponding strings files.

Formatting String Resources

For some strings, you may not want to (or be able to) encode the entire string in a string resource because portions of the string might change at runtime. For example, if a string contains the name of a user document, you need to be able to insert that document name into the string dynamically. When creating your string resources, you can use any of the formatting characters you would normally use for handling string replacement in the Foundation and Core Foundation frameworks. "Note" shows several string resources that use basic formatting characters:

Listing 3-4   Strings with formatting characters

```
"Windows must have at least %d columns and %d rows." =
"Les fenêtres doivent être composes au minimum de %d colonnes et %d lignes.";
"File %@ not found." = "Le fichier %@ n'existe pas.";
```

To replace formatting characters with actual values, you use the `stringWithFormat:` method of `NSString` or the `CFStringCreateWithFormat` function, using the string resource as the format string. Foundation and Core Foundation support most of the standard formatting characters used in `printf` statements. In addition, you can use the `%@` specifier shown in the preceding example to insert the descriptive text associated with arbitrary Objective-C objects. See Formatting String Objects in *String Programming Guide for Cocoa* for the complete list of specifiers.

One problem that often occurs during translation is that the translator may need to reorder parameters inside translated strings to account for differences in the source and target languages. If a string contains multiple arguments, the translator can insert special tags of the form *n*$ (where *n* specifies the position of the original

argument) in between the formatting characters. These tags let the translator reorder the arguments that appear in the original string. The following example shows a string whose two arguments are reversed in the translated string:

```
/* Message in alert dialog when something fails */
"%@ Error! %@ failed!" = "%2$@ blah blah, %1$@ blah!";
```

Using Special Characters in String Resources

Just as in C, some characters must be prefixed with a backslash before you can include them in the string. These characters include double quotation marks, the backslash character itself, and special control characters such as linefeed (\n) and carriage returns (\r).

```
"File \"%@\" cannot be opened" = " ... ";
"Type \"OK\" when done" = " ... ";
```

You can include arbitrary Unicode characters in a value string by specifying \\U followed immediately by up to four hexadecimal digits. The four digits denote the entry for the desired Unicode character; for example, the space character is represented by hexadecimal 20 and thus would be \\U0020 when specified as a Unicode character. This option is useful if a string must include Unicode characters that for some reason cannot be typed. If you use this option, you must also pass the -u option to genstrings in order for the hexadecimal digits to be interpreted correctly in the resulting strings file. The genstrings tool assumes your strings are low-ASCII by default and only interprets backslash sequences if the -u option is specified.

Note: The genstrings tool always generates strings files using the UTF-16 encoding. If you include Unicode characters in your strings and do not use genstrings to create your strings files, be sure to save your strings files in the UTF-16 encoding.

Debugging Strings Files

If you run into problems during testing and find that the functions and macros for retrieving strings are always returning the same key (as opposed to the translated value), run the /usr/bin/plutil tool on your strings file. A strings file is essentially a property-list file formatted in a special way. Running plutil with the -lint option can uncover hidden characters or other errors that are preventing strings from being retrieved correctly.

Image, Sound, and Video Resources

The Mac OS X and iPhone OS platforms were built to provide a rich multimedia experience. To support that experience, both platforms provide plenty of support for loading and using image, sound, and video resources in your application. Image resources are commonly used to draw portions of an application's user interface. Sound and video resources are used less frequently but can also enhance the basic appearance and appeal of an application. The following sections describe the support available for working with image, sound, and video resources in your applications.

Images and Sounds in Nib Files

Using Interface Builder, you can reference your application's sound and image files from within nib files. You might do so to associate those images or sounds with different properties of a view or control. For example, you might set the default image to display in an image view or set the image to display for a button. Creating such a connection in Interface Builder saves you the hassle of having to make that connection later when the nib file is loaded.

To make image and sound resources available in Interface Builder 3.0 and later, all you have to do is add them to your Xcode project. Interface Builder automatically searches your Xcode project for resources and lists them in the library window. When you make a connection to a given resource file, Interface Builder makes a note of that connection in the nib file. At load time, the nib-loading code looks for that resource in the project bundle, where it should have been placed by Xcode at build time.

When you load a nib file that contains references to image and sound resources, the nib-loading code caches resources whenever possible for easy retrieval later. For example, after loading a nib file, you can retrieve an image associated with that nib file using the `imageNamed:` method of either `NSImage` or `UIImage` (depending on your platform). Similarly, you can retrieve cached sound resources in Mac OS X using the `soundNamed:` method of `NSSound`. Interface Builder does not cache sound resources in iPhone OS nor does it cache image and sound resources associated with Carbon nib files.

Loading Image Resources

Image resources are commonly used in most applications. Even very simple applications use images to create a custom look for controls and views. Mac OS X and iPhone OS provide extensive support for manipulating image data using Objective-C objects. These objects make using image images extremely easy, often requiring only a few lines of code to load and draw the image. If you prefer not to use the Objective-C objects, you can also use Quartz to load images using a C-based interface. The following sections describe the process for loading image resource files using each of the available techniques.

Loading Images in Objective-C

To load images in Objective-C, you use either the `NSImage` or `UIImage` object, depending on the current platform. Applications built for Mac OS X using the AppKit framework use the `NSImage` object to load images and draw them. Applications built for iPhone OS use the `UIImage` object. Functionally, both of these objects provide almost identical behavior when it comes to loading existing image resources. You initialize the object by passing it a pointer to the image file in your application bundle and the image object takes care of the details of loading and drawing the image data.

Listing 4-1 shows how to load an image resource using the `NSImage` class. After you locate the image resource, which in this case is in the application bundle, you simply use that path to initialize the image object. After initialization, you can draw the image using the methods of `NSImage` or pass that object to other methods that can use it. To perform the exact same task in iPhone OS, all you would need to do is change references of `NSImage` to `UIImage`.

Listing 4-1   Loading an image resource

```
NSString* imageName = [[NSBundle mainBundle] pathForResource:@"image1"
ofType:@"png"];
NSImage* imageObj = [[NSImage alloc] initWithContentsOfFile:imageName];
```

You can use image objects to open any type of image supported on the target platform. Each object is typically a lightweight wrapper for more advanced image handling code. To draw an image in the current graphics context, you would simply use one of its drawing related methods. Both `NSImage` and `UIImage` have methods for drawing the image in several different ways. The `NSImage` class also provides extra support for manipulating the images you load.

For information about the methods of the `NSImage` and `UIImage` classes, see *NSImage Class Reference* and *UIImage Class Reference*. For more detailed information about the additional features of the `NSImage` class, see Images in *Cocoa Drawing Guide*.

Loading Images Using Quartz

If you are writing C-based code, you can use a combination of Core Foundation and Quartz calls to load image resources into your applications. Core Foundation provides the initial support for locating image resources and loading the corresponding image data into memory. Quartz takes the image data you load into memory and turns it into a usable `CGImageRef` that your code can then use to draw the image.

There are two ways to load images using Quartz: data providers and image source objects. Data providers are available in both iPhone OS and Mac OS X. Image source objects are available only in Mac OS X v10.4 and later but take advantage of the Image I/O framework to enhance the basic image handling capabilities of data providers. When it comes to loading and displaying image resources, both technologies are well suited for the job. The only time you might prefer image sources over data providers is when you want greater access to the image-related data.

Listing 4-2 shows how to use a data provider to load a JPEG image. This method uses the Core Foundation bundle support to locate the image in the application's main bundle and get a URL to it. It then uses that URL to create the data provider object and then create a `CGImageRef` for the corresponding JPEG data. (For brevity this example omits any error-handling code. Your own code should make sure that any referenced data structures are valid.)

Listing 4-2   Using data providers to load image resources

```
CGImageRef MyCreateJPEGImageRef (const char *imageName);
{
    CGImageRef image;
    CGDataProviderRef provider;
    CFStringRef name;
    CFURLRef url;
    CFBundleRef mainBundle = CFBundleGetMainBundle();

    // Get the URL to the bundle resource.
    name = CFStringCreateWithCString (NULL, filename, kCFStringEncodingUTF8);
    url = CFBundleCopyResourceURL(mainBundle, name, CFSTR("jpg"), NULL);
    CFRelease(name);

    // Create the data provider object
    provider = CGDataProviderCreateWithURL (url);
    CFRelease (url);

    // Create the image object from that provider.
    image = CGImageCreateWithJPEGDataProvider (provider, NULL, true,
                                kCGRenderingIntentDefault);
    CGDataProviderRelease (provider);

    return (image);
}
```

For detailed information about working with Quartz images, see *Quartz 2D Programming Guide*. For reference information about data providers, see *Quartz 2D Reference Collection* (Mac OS X) or *Core Graphics Framework Reference* (iPhone OS).

Playing Audio Files

Audio resources are typically used to provide audio feedback for different parts of your application. Several technologies are available to handle the loading and playback of audio. Which technology you use is going to be determined by the underlying platform and the level of sophistication you need for handling the audio. The following sections describe the key technologies you might use and when you would use them.

Using Core Audio to Play Sounds

Both Mac OS X and iPhone OS support the playback of audio files using the Core Audio family of frameworks. Core Audio provides a wide range of audio services, including the playback of essentially any kind of audio file you can imagine. For basic playback, Core Audio offers two mechanisms, both available in the Audio Toolbox framework:

- To play short sound files of under five seconds duration when you do not need level control or other control, use System Audio Services.

- To play longer sound files, to exert control over playback including level adjustments, or to play multiple sounds simultaneously, use Audio Queue Services.

Listing 4-3 shows a short program that uses the interfaces in System Audio Services to play a sound. Before playing the sound, it registers it and creates a sound ID for it. To play the sound, it then passes this sound ID to the `AudioServicesPlaySystemSound` function. When the sound is finished playing, Core Audio notifies the application by calling its audio completion callback routine. This routine handles the clean up of the sound ID prior to the program exiting.

Listing 4-3   Playing a sound using System Audio Services

```
#include <AudioToolbox/AudioToolbox.h>
#include <CoreFoundation/CoreFoundation.h>

// Define a callback to be called when the sound is finished
// playing. Useful when you need to free memory after playing.
static void MyCompletionCallback (
    SystemSoundID  mySSID,
    void * myURLRef
) {
        AudioServicesDisposeSystemSoundID (mySSID);
        CFRelease (myURLRef);
        CFRunLoopStop (CFRunLoopGetCurrent());
}

int main (int argc, const char * argv[]) {
    // Set up the pieces needed to play a sound.
    SystemSoundID    mySSID;
    CFURLRef         myURLRef;
    myURLRef = CFURLCreateWithFileSystemPath (
        kCFAllocatorDefault,
        CFSTR ("../../ComedyHorns.aif"),
        kCFURLPOSIXPathStyle,
        FALSE
    );

    // create a system sound ID to represent the sound file
    OSStatus error = AudioServicesCreateSystemSoundID (myURLRef, &mySSID);

    // Register the sound completion callback.
    // Again, useful when you need to free memory after playing.
    AudioServicesAddSystemSoundCompletion (
        mySSID,
        NULL,
        NULL,
        MyCompletionCallback,
        (void *) myURLRef
    );

    // Play the sound file.
    AudioServicesPlaySystemSound (mySSID);

    // Invoke a run loop on the current thread to keep the application
    // running long enough for the sound to play; the sound completion
    // callback later stops this run loop.
    CFRunLoopRun ();
    return 0;
}
```

For more information about the features of Core Audio, see *Core Audio Overview*. For information and examples of how to play sounds using the Audio Queue Services technology, see *Audio Queue Services Programming Guide*.

Using the AppKit Framework to Play Audio

In Mac OS X, the AppKit framework provides support for loading and playing sound files through the `NSSound` class. You can use this class to play back sounds stored as AIFF, WAV, and NeXT `.snd` files. For sound resources located in your application's bundle, the simplest way to load a sound is using the `soundNamed:` method, as shown in the following example:

```
NSSound* aSound = [NSSound soundNamed:@"mySound"];
```

The `soundNamed:` method checks the application's sound cache for an existing sound resource with the specified name. If the specified resource is not currently in the sound cache, `NSSound` automatically searches for it in several other locations, including your application's main bundle and any system `Library/Sounds` directories.

Because the `soundNamed:` method also loads system sound names, you should avoid using the names of system sounds when naming any of your custom sound files. Cocoa populates the sound cache with any sound files it needs, such as the file used for the current system alert sound. It caches these sounds under the filename of the sound (minus its filename extension). If one of your custom sounds matches the name of a different sound file that is already cached, the `soundNamed:` method returns the cached file instead of your custom one.

If you want to ensure that the correct sound file resource is loaded every time, you can always load the sound file using an explicit path string, as shown in the following example.

```
NSString* soundFile = [[NSBundle mainBundle] pathForResource:@"mySound"
ofType:@"aiff"];
NSSound* sound = [[NSSound alloc] initWithContentsOfFile:soundFile
byReference:YES];
```

> **Note:** Sound files associated with a nib file are loaded automatically when the nib file is loaded. To access those sounds, use the `soundNamed:` method of `NSSound`, passing in the name of the sound. For more information, see "About Image and Sound Resources" (page 15).

For more information about using the `NSSound` class, see *Sound Programming Topics for Cocoa* and *NSSound Class Reference*. If you want to load sound resources for Carbon-based applications, you must use QuickTime or Core Audio to do so. For information about the QuickTime Kit framework, see *QuickTime Kit Programming Guide* and *QTKit Framework Reference*. For general information about the QuickTime framework, see *QuickTime Overview*.

Playing Video Resources

Video resources are prerendered movie files that you can play from your application's user interface. Games often use prerendered movies as cut scenes between different levels. The following sections provide information about how to load these types of resources and play them in your applications.

Playing Video Files in Mac OS X

Video files are like any other resource files in your application. Once you locate the resource file, you can use an appropriate technology to open and play it. In Mac OS X, you use the QuickTime or Quicktime Kit frameworks to open video files, associate them with a graphics context, and play their contents. These frameworks support the playback of both video and audio files in either C or Objective-C code.

The following example loads a video file from an application's bundle and associates it with a view using the QuickTime Kit framework. The view object returned by the `getMyQTMovieView` method is assumed to be a `QTMovieView` object located in one of the caller's windows.

```
NSString*    movieFile = [[NSBundle mainBundle] pathForResource:@"myMovie"
ofType:@"mov"];
QTMovie*     aMovie = [QTMovie movieWithFile:movieFile error:nil];

// Install the movie in a custom movie view associated with the caller.
QTMovieView*    myView = [self getMyQTMovieView];
[myView setMovie:aMovie];
```

Prior to Mac OS X v10.4, you can use the `NSMovie` and `NSMovieView` classes in Cocoa to load and display video files. In Mac OS X v10.4 and later, it is recommended that you use the classes of the QuickTime Kit framework instead.

For C-based applications, you can load video files using either the QuickTime framework or the QuickTime Kit framework. If you choose to use the QuickTime Kit framework, you must incorporate Objective-C code into your project. For information on how to use Objective-C code in Carbon applications, see *Carbon-Cocoa Integration Guide*.

For information about the QuickTime Kit framework, see *QuickTime Kit Programming Guide* and *QTKit Framework Reference*. For general information about the QuickTime framework, see *QuickTime Overview*. For details of how to incorporate movie content into your application, see *QuickTime Movie Basics*.

Playing Video Files in iPhone OS

iPhone OS supports the ability to play back video files directly from your application using the Media Player framework (`MediaPlayer.framework`). Video playback is supported in full screen mode only and can be used by game developers who want to play cut scene animations or by other developers who want to play media files. When you start a video from your application, the media player interface takes over, fading the screen to black and then fading in the video content. You can play a video with or without transport controls; enabling transport controls lets the user pause or adjust the playback of the video. If you do not enable these controls, the video plays until completion or until you explicitly stop it in your code.

To initiate video playback, you must know the URL of the file you want to play. For files your application provides, this would typically be a pointer to a file in your application's bundle; however, it can also be a pointer to a file on a remote server or elsewhere in the directory containing your application. You use this URL to instantiate a new instance of the `MPMoviePlayerController` class. This class presides over the playback of your video file and manages user interactions, such user taps in the transport controls (if shown). To initiate playback, simply call the `play` method of the controller.

Listing 4-4 shows a sample method that playbacks the video at the specified URL. The play method is an asynchronous call that returns control to the caller while the movie plays. The movie controller loads the movie in a full-screen view, and animates the movie into place on top of the application's existing content. When playback is finished, the movie controller sends a notification to the object, which releases the movie controller now that it is no longer needed.

Listing 4-4   Playing full screen movies.

```
-(void)playMovieAtURL:(NSURL*)theURL
{
    MPMoviePlayerController* thePlayer = [[MPMoviePlayerController alloc]
initWithContentURL:theURL];

    thePlayer.scalingMode = MPMovieScalingModeAspectFill;
    thePlayer.userCanShowTransportControls = NO;

    // Register for the playback finished notification.
    [[NSNotificationCenter defaultCenter] addObserver:self
                selector:@selector(myMovieFinishedCallback:)
                name:MPMoviePlayerPlaybackDidFinishNotification
                thePlayer];

    // Movie playback is asynchronous, so this method returns immediately.
    [thePlayer play];
}

// When the movie is done, release the controller.
-(void)myMovieFinishedCallback:(NSNotification*)aNotification
{
    MPMoviePlayerController* thePlayer = [aNotification object];

    [[NSNotificationCenter defaultCenter] removeObserver:self
                name:MPMoviePlayerPlaybackDidFinishNotification
                thePlayer];

    // Release the movie instance created in playMovieAtURL:
    [thePlayer release];
}
```

Carbon Resources

Carbon applications use all of the same string, image, sound, and video resources that Cocoa applications use. Carbon applications also support the use of nib files for loading the application's user interface. Unlike Cocoa applications, however, Carbon applications typically do not instantiate all of the objects in a nib file at once. Instead, applications load individual windows and menus from one or more nib files at different points in the program. This chapter provides examples of how to load nib files containing Carbon objects.

> **Note:** Carbon supports the loading of nib files through Interface Builder Services, which is part of the HIToolbox in the Carbon framework (`Carbon.framework`). For information about the functions in Interface Builder Services, see *Interface Builder Services Reference*.

Nib File Design Guidelines

When creating your Carbon nib files, keep the following guidelines in mind:

- Store only launch-related resources in your application's main nib file. The main nib file should contain the application menu bar and any resources that are needed at launch time only. Other resources should be stored separately.

- Store frequently used user-interface components (such as document windows) in their own separate nib files. This nib file should contain only the resources needed immediately by that component.

- Store infrequently used user-interface components (such as alert panels) in separate nib files. Avoid storing infrequently used components in nib files with frequently used components. When a nib file is loaded, the data for the entire file is loaded into memory and objects are instantiated from that data as needed. Thus, if an object is not used it is still occupying space in memory.

Loading Objects from Nib Files

There are three main ways to load objects. You can:

- Load a menu bar and main window from your application's main nib file.

- Load objects from an auxiliary nib file in the main bundle.

- Load objects from nib files in other bundles.

In this section, you'll find steps and code examples for each technique.

Unarchiving Objects from the Main Nib File

When your application starts up, you need to call Interface Builder Services functions to open the main nib file and unarchive the interface objects that should be open after startup. The main nib file should contain only those items that are essential when your application starts up. In most cases, the main nib file should contain only the menu bar and perhaps a main window.

The steps below outline how to open a main nib file and unarchive the objects in it. Listing 5-1 (page 48) shows how to implement the steps for a nib file that contains a menu bar and a main window. If your application needs only one of these objects at startup, you can easily modify the sample code.

1. Call the function `CreateNibReference` to create a reference to the main nib file.

2. Unarchive the menu bar from the main nib file by calling the function `SetMenuBarFromNib`. This function also sets the menu bar so users can use the menu bar when your application has started up.

3. If your application has a main window, call the function `CreateWindowFromNib` to unarchive the main window.

4. After you have unarchived the objects from the main nib file, dispose of the nib reference by calling the function `DisposeNibReference`.

5. The function `CreateWindowFromNib` unarchives the window so that it is hidden. If you want the window to be visible, you must call the Window Manager function `ShowWindow`.

It is good practice to check for errors each step of the way, as shown in Listing 5-1. If the main user interface cannot be created, your application should halt the start up process and exit. This example assumes that the name of your application's main nib file is `main.nib`.

Listing 5-1   Unarchiving the menu bar and main window from the main nib file

```
int main (int argc, char* argv[])
{
    IBNibRef        nibRef;
    WindowRef       window;
    OSStatus        err;

    // Create a nib reference to a nib file.
    err = CreateNibReference (CFSTR ("main"), &nibRef);
    // Call the macro require_noerr to make sure no errors occurred
    require_noerr (err, CantGetNibRef);

    // Unarchive the menu bar and make it ready to use.
    err = SetMenuBarFromNib (nibRef, CFSTR("MainMenu"));
    require_noerr (err, CantSetMenuBar);

    // Unarchive the main window.
    err = CreateWindowFromNib (nibRef, CFSTR("MainWindow"), &window);
    require_noerr (err, CantCreateWindow);

    // Dispose of the nib reference as soon as you don't need it any more.
    DisposeNibReference (nibRef);
    // Make the unarchived window visible.
    ShowWindow (window);
```

```
      // Start the event loop. RunApplicationEventLoop is a
       // Carbon Event Manager function.
      RunApplicationEventLoop ();

      // You'll jump to one of the "Cant" statements only if there's
      // an error.
      CantCreateWindow:
      CantSetMenuBar:
      CantGetNibRef:
      return err;
}
```

Unarchiving an Object from an Auxiliary Nib File

For most applications, it is useful to factor your application's interface objects into several nib files. The main nib file should contain at the most, the menu bar and the window that opens (if any) when your application starts up. Document windows, palettes, toolbars, contextual menus, and other interface objects should be stored in separate nib files.

The steps for opening an auxiliary nib file and unarchiving an object from it are similar to those used for your main nib file:

1. Call the function `CreateNibReference` to create a reference to the auxiliary nib file that contains the object you want to unarchive.

2. Call the appropriate function to unarchive the object from the nib file. To unarchive a window, call the function `CreateWindowFromNib`; to unarchive a menu, call the function `CreateMenuFromNib`.

3. After you have unarchived the object from the auxiliary nib file, dispose of the nib reference by calling the function `DisposeNibReference`.

One common use of an auxiliary nib file is to store an object that's used repeatedly in an application, such as a document window. Another use is to store objects that are rarely needed, such as an About window. Listing 5-2 shows how to implement a `MyCreateNewDocument` function that your application would call each time the user creates a new document. The code uses macro-based error checking to abort the operation if an error occurs.

Listing 5-2   Unarchiving a document window from an auxiliary nib file

```
WindowRef MyCreateNewDocument (CFStringRef inName)
{
    IBNibRef documentNib;
    OSStatus err;
    WindowRef theWindow;

    // Create a nib reference to an auxiliary nib file with
    // the name document.nib.
    err = CreateNibReference (CFSTR ("document"), &documentNib);
    // Call the macro require_noerr to make sure no errors occurred
    require_noerr (err, CantGetNibRef);

    // Unarchive the document window. Use the name you gave to the
    // window object in the Instances pane in Interface Builder.
    err = CreateWindowFromNib (documentNib, CFSTR("MyDocument"),
```

```
                                                    &theWindow);
    require_noerr (err, CantCreateWindow);

    // Dispose of the nib reference as soon as you don't need it anymore.
    DisposeNibReference (documentNib);

    // Call the Window Manager function to set the title shown in the
    // window's title bar to the name passed to MyCreateNewDocument.
    err = SetWindowTitleWithCFString (theWindow, inName);
    // In this example, the window gets returned. Remember, it's been
    // unarchived, but it is still not visible. It won't be visible
    // until you call the Window Manager function ShowWindow.
    return theWindow;

    // You'll jump to one of the "Cant" statements only if there's
    // an error.
    CantCreateWindow:
    CantGetNibRef:
     return NULL;
}
```

Unarchiving an Object from an External Bundle

Your application is not limited to using interface objects contained within its own bundle. You can unarchive interface objects from another bundle or framework to which your application has access. For example, you could unarchive a tools palette or other object provided by a plug-in bundle.

The steps for unarchiving an object from a nib file in a framework or other bundle are similar to those used to open an auxiliary nib file and are listed below. The main difference is that you must first create a a reference to the external bundle. You must also call the function `CreateNibReferenceWithCFBundle` instead of `CreateNibReference` to create a reference to the nib file. The steps are as follows:

1. Call the Core Foundation URL Services function `CFURLCreateWithFileSystemPath` to create a URL that points to the desired bundle. (For reference documentation, see *CFURL Reference*.)

2. Call the Core Foundation Bundle Services function `CFBundleCreate` to create a reference to the bundle that contains nib file you want to open. (For reference documentation, see *CFBundle Reference*.)

3. Call the function `CreateNibReferenceWithCFBundle` to create a reference to the nib file that contains the object you want to unarchive.

4. Call the appropriate function to unarchive the object from the nib file. To unarchive a window, call the function `CreateWindowFromNib`; to unarchive a menu, call the function `CreateMenuFromNib`.

5. After you have unarchived the object from the nib file, dispose of the nib reference by calling the function `DisposeNibReference`.

The function `MyCreateWidgetFromFramework`, shown in Listing 5-3, shows how to unarchive a "widget window" from a bundle whose path you pass to the function.

Listing 5-3   Unarchiving a widget window from a nib file in a bundle

```
WindowRef MyCreateWidgetFromBundle (CFStringRef widgetBundlePath
                                    CFStringRef widgetFileName,
```

```
                                   CFStringRef  widgetWindowName)
{
    IBNibRef widgetNib;
    OSStatus err;
    WindowRef theWindow;
    CFBundleRef mainBundle;
    CFURLRef    bundleURL;
    CFBundleRef widgetBundle;

    // Look for a resource in the bundle passed to
    // the function MyCreateWidgetFromBundle
    bundleURL = CFURLCreateWithFileSystemPath(
                        kCFAllocatorDefault,
                        widgetBundlePath,
                        kCFURLPOSIXPathStyle,
                        TRUE);
    // Make a bundle instance using the URL Reference
    widgetBundle = CFBundleCreate (kCFAllocatorDefault, bundleURL);

    // Create a nib reference to the nib file.
    err = CreateNibReferenceWithCFBundle (widgetBundle,
                        widgetFileName, &widgetNib);
    // Call the macro require_noerr to make sure no errors occurred
    require_noerr (err, CantGetNibRef);

    // Unarchive the widget window.
    err = CreateWindowFromNib (widgetNib, widgetWindowName, &theWindow);
    require_noerr (err, CantCreateWindow );

    // Dispose of the nib reference as soon as you don't need it anymore.
    DisposeNibReference (widgetNib);
    // Release the Core Foundation objects
    CFRelease (bundleURL);
    CFRelease (widgetBundle);

    // In this example, the window gets returned. Remember, it's been
    // unarchived, but it is still not visible. It won't be visible
    // until you call the Window Manager function ShowWindow.
    return theWindow;

    // You'll jump to one of the "Cant" statements only if there's
    // an error.
    CantCreateWindow:
    CantGetNibRef:
     return NULL;
}
```

Document Revision History

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

Date	Notes
2009-01-06	Added information about KVO notifications during nib loading.
2008-06-26	Updated for iPhone OS.
2007-09-04	Clarified the process of how objects are instantiated when a nib file is loaded.
2007-02-08	Reorganized content and added new information. Changed title from "Loading Resources".
2005-11-09	Corrected the misidentification of a class method as an instance method.
2003-07-09	Added "Instantiating Nibs From Memory" and the link to the `NSNib` class reference.
2003-05-28	Section on initializing nib file objects corrected and expanded.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.