Resource Programming Guide

Developer

Contents

Tables and Listings

About Resources

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 OS X and iOS 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 OS X and iOS 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*.

At a Glance

Applications can contain many types of resources but there are several that are supported directly by iOS and OS X.

Nib Files Store the Objects of Your Application's User Interface

Nib files are the quintessential resource type used to create iOS and Mac apps. A **nib file** is a data archive that essentially contains a set of freeze-dried objects that you want to recreate at runtime. Nib files are used most commonly to store preconfigured windows, views, and other visually oriented objects but they can also store nonvisual objects such as controllers.

You create nib files using the Interface Builder application, which provides a graphical assembly area for assembling your objects. When you subsequently load a nib file into your application, the nib-loading code instantiates each object in the file and restores it to the state you specified in Interface Builder. Thus, what you see in Interface Builder is really what you get in your application at runtime.

Relevant Chapters "Nib Files" (page 9)

String Resources Containing Localizable Text

Text is a prominent part of most user interfaces but also a resource that is most affected by localization changes. Rather than hard-coding text into your code, iOS and OS X support the storage of user-visible text in **strings files**, which are human-readable text files (in the UTF-16 encoding) containing a set of string resources for an application. (The use of the plural "strings" in is deliberate and due to the `.strings` filename extension used by files of that type.) Strings files greatly simplify the internationalization and localization process by allowing you to write your code once and then load the appropriately localized text from resource files that can be changed easily.

The Core Foundation and Foundation frameworks provide the facilities for loading text from strings files. Applications that use these facilities can also take advantage of tools that come with Xcode to generate and maintain these resource files throughout the development process.

Relevant Chapters "String Resources" (page 32)

Images, Sounds, and Movies Represent Pre-rendered Content

Images, sound, and movie resources play an important role in iOS and Mac apps. Images are responsible for creating the unique visual style used by each operating system; they also help simplify your drawing code for complex visual elements. Sound and movie files similarly help enhance the overall user experience of your application while simplifying the code needed to create that experience. Both operating systems provide extensive support for loading and presenting these types of resources in your applications.

Relevant Chapters "Image, Sound, and Video Resources" (page 43)

Property Lists and Data Files Separate Data from Code

A property list file is a structured file used to store string, number, Boolean, date, and raw data values. Data items in the file are organized using array and dictionary structures with most items associated with a unique key. The system uses property lists to store simple data sets. For example, the `Info.plist` file found in nearly every application is an example of a property list file. You can also use property list files for simple data storage needs.

In addition to property lists, OS X supports some specially structured files for specific uses. For example, AppleScript data and user help are stored using specially formatted data files. You can also create custom data files of your own.

Relevant Chapters "Data Resource Files" (page 48)

iOS Supports Device-Specific Resources

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

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

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

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

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

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

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

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

See Also

The following Apple Developer 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 its 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.

Nib Files

Nib files play an important role in the creation of applications in OS X and iOS. With nib files, you create and manipulate your user interfaces graphically, using Xcode, instead of programmatically. Because you can see the results of your changes instantly, you can experiment with different layouts and configurations very quickly. You can also 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 laying out windows, views, and controls and for integrating those items with the application's event handling code. Xcode works in conjunction with these frameworks 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.

> **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 nib files might require you to replace large amounts of framework behavior to achieve the same results you would get using a nib file.

Anatomy of a Nib File

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 Xcode. 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 Xcode document. The nib-loading code instantiates the objects, configures them, and reestablishes any inter-object connections that you created in your nib file.

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 nib file 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 nib files 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. Xcode 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 placeholder 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 Xcode, 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 a nib file, the First Responder is a placeholder 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 placeholder 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 placeholder. 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 placeholder 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 *Event Handling Guide for iOS* .

About the Top-Level Objects

When your program loads a nib file, Cocoa recreates the entire graph of objects you created in Xcode. 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 placeholder objects and not considered top-level objects.)

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 done using them. For more information about the nib object behavior at load time, see "Managing the Lifetimes of Objects from Nib Files" (page 15).

About Image and Sound Resources

In Xcode, you can refer to external image and sound resources from within 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 Xcode 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.

When you load a nib file that contains references to image or sound resources, the nib-loading code reads the actual image or sound file into memory and and caches it. In OS X, image and sound resources are stored in named caches so that you can access them later if needed. In iOS, 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 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. 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 an 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 30).

The Nib Object Life Cycle

When a nib file is loaded into memory, the nib-loading code takes several steps to 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 11).
    - 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 11).
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. The nib-loading code uses the following rules (in order) to determine which initialization method to use.
    a. By default, objects receive an `initWithCoder:` message.

In 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 Xcode library. It also includes any third-party objects that were added to the library using a custom plug-in. Even if you change the class of such an object, Xcode 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 iOS, 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 Xcode library or custom classes you define.

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

Custom views are subclasses of `NSView` for which Xcode 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 library or part of an integrated third-party plug-in.

When it encounters a custom view, Xcode 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 iOS 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 placeholder objects. The approach for establishing connections differs depending on the platform:
    - Outlet connections
        - In 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 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 iOS, 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 iOS 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 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 iOS, 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 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 iOS.

4. It sends an `awakeFromNib` message to the appropriate objects in the nib file that define the matching selector:
   - In 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 placeholder objects that define it as well.
   - In iOS, 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 placeholder objects.
5. It displays any windows whose "Visible at launch time" attribute was enabled in the nib file.

The order in which the nib-loading code calls the `awakeFromNib` methods of objects is not guaranteed. In 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.

Managing the Lifetimes of Objects from Nib Files

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.) You need to ensure that you maintain the new object graph as long as necessary, and disown it when you are finished with it. You typically need strong references to top-level objects to ensure that they are not deallocated; you don't need strong references to objects lower down in the graph because they're owned by their parents, and you should minimize the risk of creating strong reference cycles.

From a practical perspective, in iOS and OS X outlets should be defined as declared properties. Outlets should generally be `weak`, except for those from File's Owner to top-level objects in a nib file (or, in iOS, a storyboard scene) which should be `strong`. Outlets that you create should therefore typically be `weak`, because:

- Outlets that you create to subviews of a view controller's view or a window controller's window, for example, are arbitrary references between objects that do not imply ownership.
- The strong outlets are frequently specified by framework classes (for example, `UIViewController`'s `view` outlet, or `NSWindowController`'s `window` outlet).

```
@property (weak) IBOutlet MyView *viewContainerSubview;
@property (strong) IBOutlet MyOtherClass *topLevelObject;
```

> **Note** In OS X, not all classes support weak references; these are `NSATSTypesetter`, `NSColorSpace`, `NSFont`, `NSFontManager`, `NSFontPanel`, `NSImage`, `NSMenuView`, `NSParagraphStyle`, `NSSimpleHorizontalTypesetter`, `NSTableCellView`, `NSTextView`, `NSViewController`, `NSWindow`, and `NSWindowController`, and all classes in the AV Foundation framework.
>
> In cases where you cannot therefore specify `weak`, you should instead use `assign`:
>
> ```
> @property (assign) IBOutlet NSTextView *textView;
> ```

Outlets may be considered private to the defining class; if you prefer, you can hide the property declarations a class extension. For example:

```
// MyClass.h

@interface MyClass : MySuperclass
@end
```

```
// MyClass.m

@interface MyClass ()
@property (weak) IBOutlet MyView *viewContainerSubview;
@property (strong) IBOutlet MyOtherClass *topLevelObject;
@end
```

These patterns extend to references from a container view to its subviews where you have to consider the internal consistency of your object graph. For example, in the case of a table view cell, outlets to specific subviews should again typically be weak. If a table view contains an image view and a text view, then these remain valid so long as they are subviews of the table view cell itself.

Outlets should be changed to `strong` when the outlet should be considered to own the referenced object:

- As indicated previously, this is often the case with File's Owner—top level objects in a nib file are frequently considered to be owned by the File's Owner.
- You may in some situations need an object from a nib file to exist outside of its original container. For example, you might have an outlet for a view that can be temporarily removed from its initial view hierarchy and must therefore be maintained independently.

Classes that you expect to be subclassed (in particular abstract classes) expose outlets publicly so that they can be used appropriately by subclasses (e.g. `UIViewController`'s `view` outlet). Outlets might also be exposed where there is an expectation that consumers of the class will need to interact with the property; for example a table view cell might expose subviews. In this latter case, it may be appropriate to expose a read-only public outlet that is redefined privately as read-write, for example:

```
// MyClass.h

@interface MyClass : UITableViewCell
@property (weak, readonly) MyType *outletName;
@end

// MyClass.m

@interface MyClass ()
@property (weak, readwrite) IBOutlet MyType *outletName;
@end
```

Top-level Objects in OS X May Need Special Handling

For historical reasons, in OS X the top-level objects in a nib file are created with an additional reference count. The Application Kit offers a couple of features that help to ensure that nib objects are properly released:

- `NSWindow` objects (including panels) have an `isReleasedWhenClosed` attribute, which if set to `YES` instructs the window to release itself (and consequently all dependent objects in its view hierarchy) when it is closed. In the nib file, you set this option through the "Release when closed" check box in the Attributes pane of the Xcode inspector.
- If the File's Owner of a nib file is an `NSWindowController` object (the default in document nibs in document-based applications—recall that `NSDocument` manages an instance of `NSWindowController`) or an `NSViewController` object, it automatically disposes of the windows it manages.

If the File's Owner is not an instance of `NSWindowController` *or* `NSViewController`, then you need to decrement the reference count of the top level objects yourself. With manual reference counting, it was possible to achieve this by sending top-level objects a `release` message. You cannot do this with ARC. Instead, you cast references to top-level objects to a Core Foundation type and use `CFRelease`. (If you don't want to have outlets to all top-level objects, you can use the `instantiateNibWithOwner:topLevelObjects:` method of the `NSNib` class to get an array of a nib file's top-level objects.)

Legacy Patterns

Prior to ARC, the rules for managing nib objects are different from those described above. How you manage the objects depends on the platform and on the memory model in use. Whichever platform you develop for, you should define outlets using the Objective-C declared properties feature.

The general form of the declaration should be:

```
@property (attributes) IBOutlet UserInterfaceElementClass *anOutlet;
```

Because the behavior of outlets depends on the platform, the actual declaration differs:

- For iOS, you should use:

  ```
  @property (nonatomic, retain) IBOutlet UserInterfaceElementClass *anOutlet;
  ```

- For OS X, you should use:

  ```
  @property (assign) IBOutlet UserInterfaceElementClass *anOutlet;
  ```

You should then either synthesize the corresponding accessor methods, or implement them according to the declaration, and (in iOS) release the corresponding variable in `dealloc`.

This pattern also works if you use the modern runtime and synthesize the instance variables, so it remains consistent across all situations.

Managing Nib Objects in iOS

Top-Level Objects

Objects in the nib file are created with a retain count of 1 and then autoreleased. As it rebuilds the object hierarchy, UIKit reestablishes connections between the objects using `setValue:forKey:`, which uses the available setter method or retains the object by default if no setter method is available. This means that (assuming you follow the pattern shown above) *any object for which you have an outlet* remains valid. If there are any *top-level* objects you do not store in outlets, however, 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.

Memory Warnings

When a view controller receives a memory warning (`didReceiveMemoryWarning`), it should relinquish ownership of resources that are currently not needed and that can be recreated later if required. One such resource is the view controller's view itself. If it does not have a superview, the view is disposed of (in its implementation of `didReceiveMemoryWarning`, `UIViewController` invokes `[self setView:nil]`).

Because outlets to elements within the nib file are typically retained, however, even though the main view is disposed of, absent any further action the outlets are not disposed of. This is not in and of itself a problem—if and when the main view is reloaded, they will simply be replaced—but it does mean that the beneficial effect of the `didReceiveMemoryWarning` is reduced. To ensure that you properly relinquish ownership of outlets, in your custom view controller class you can implement `viewDidUnload` to invoke your accessor methods to set outlets to `nil`.

```
- (void)viewDidUnload {
    self.anOutlet = nil;
    [super viewDidUnload];
}
```

Managing Nib Objects in OS X

In OS X, the File's Owner of a nib file is by default responsible for releasing the top-level objects in a nib file as well as any non-object resources created by the objects in the nib. The release of the root object of an object graph sets in motion the release of all dependent objects. The File's Owner of an application's main nib file (which contains the application menu and possibly other items) is the global application object `NSApp`. However, when a Cocoa application terminates, top level objects in the main nib do not automatically get `dealloc` messages just because `NSApp` is being deallocated. In other words, even in the main nib file, you have to manage the memory of top-level objects.

The Application Kit offers a couple of features that help to ensure that nib objects are properly released:

- `NSWindow` objects (including panels) have an `isReleasedWhenClosed` attribute, which if set to `YES` instructs the window to release itself (and consequently all dependent objects in its view hierarchy) when it is closed. In the nib file, you set this option through the "Release when closed" check box in the Attributes pane of the inspector.
- If the File's Owner of a nib file is an `NSWindowController` object (the default in document nibs in document-based applications—recall that `NSDocument` manages an instance of `NSWindowController`), it automatically disposes of the windows it manages.

So in general, you are responsible for releasing top-level objects in a nib file. But in practice, if your nib file's owner is an instance of `NSWindowController` it releases the top-level object for you. If one of your objects loads the nib itself (and the owner is not an instance of `NSWindowController`), you can define outlets to each top-level object so that at the appropriate time you can release them using those references. If you don't want to have outlets to all top-level objects, you can use the `instantiateNibWithOwner:topLevelObjects:` method of the `NSNib` class to get an array of a nib file's top-level objects.

The issue of responsibility for nib object disposal becomes clearer when you consider the various kinds of applications. Most Cocoa applications are of two kinds: single window applications and document-based applications. In both cases, memory management of nib objects is automatically handled for you to some degree. With single-window applications, objects in the main nib file persist through the runtime life of the application and are released when the application terminates; however, `dealloc` is not guaranteed to be automatically invoked on objects from the main nib file when an application terminates. In document-based applications each document window is managed by an `NSWindowController` object which handles memory management for a document nib file.

Some applications may have a more complex arrangement of nib files and top-level objects. For example, an application could have multiple nib files with multiple window controllers, loadable panels, and inspectors. But in most of these cases, if you use `NSWindowController` objects to manage windows and panels or if you set the "released when closed" window attribute, memory management is largely taken care of. If you decide against using window controllers and do not want to set the "release when closed" attribute, you should

explicitly free your nib file's windows and other top-level objects when the window is closed. Also, if your application uses an inspector panel, (after being lazily loaded) the panel should typically persist throughout the lifetime of the application—there is no need to dispose of the inspector and its resources.

Action Methods

Broadly speaking, action methods (see Target-Action in OS X or Target-Action in iOS) are methods that are typically invoked by another object in a nib file. Action methods use type qualifier `IBAction`, which is used in place of the `void` return type, to flag the declared method as an action so that Xcode is aware of it.

```
@interface MyClass
- (IBAction)myActionMethod:(id)sender;
@end
```

You may choose to regard action methods as being private to your class and thus not declare them in the public `@interface`. (Because Xcode parses implementation files, there is no need to declare them in the header.)

```
// MyClass.h

@interface MyClass
@end

// MyClass.m

@implementation MyClass
- (IBAction)myActionMethod:(id)sender {
    // Implementation.
}
@end
```

You should typically not invoke an action method programmatically. If your class needs to perform the work associated with the action method, then you should factor the implementation into a different method that is then invoked by the action method.

```
// MyClass.h
```

```
@interface MyClass
@end

// MyClass.m

@interface MyClass (PrivateMethods)
- (void)doSomething;
- (void)doWorkThatRequiresMeToDoSomething;
@end

@implementation MyClass
- (IBAction)myActionMethod:(id)sender {
    [self doSomething];
}

- (void)doSomething {
    // Implementation.
}

- (void)doWorkThatRequiresMeToDoSomething {
    // Pre-processing.
    [self doSomething];
    // Post-processing.
}

@end
```

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.

The Application Loads 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 the nib file 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.

Each View Controller Manages its Own Nib File

The `UIViewController` (iOS) and `NSViewController` (OS X) classes support the automatic loading of their associated nib file. If you specify a nib file when creating the view controller, that nib file is loaded automatically when you try to access the view controller's view. Any connections between the view controller and the nib file objects are created automatically, and in iOS, the `UIViewController` object also receives additional notifications when the views are finally loaded and displayed on screen. To help manage memory better, the `UIViewController` class also handles the unloading of its nib file (as appropriate) during low-memory conditions.

For more information about how you use the `UIViewController` class and how you configure it, see *View Controller Programming Guide for iOS* .

Document and Window Controllers Load Their Associated Nib File

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.

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 the window. For more information about the Cocoa document architecture, see *Document-Based App Programming Guide for Mac*.

Loading Nib Files Programmatically

Both OS X and iOS 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 11).

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 be 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 "Managing the Lifetimes of Objects from Nib Files" (page 15).

Listing 1-1 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 1-1 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 1-2 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 placeholder objects other than the File's Owner object. For an example of how to specify additional placeholder objects, see "Replacing Proxy Objects at Load Time" (page 28).

Listing 1-2 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;
}
```

Note If you are developing a Universal application for iOS, you can use the device-specific naming conventions to load the correct nib file for the underlying device automatically. For more information about how to name your nib files, see "iOS Supports Device-Specific Resources" (page 7).

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 1-3 shows the interface and implementation of a simplified 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 been retained the window. You would not want to release top-level objects in this manner in an iPhone application.

Listing 1-3 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 OS X, you must ask for the objects explicitly, whereas in iOS they are returned to you automatically.

Listing 1-4 shows the process for getting the top-level objects of a nib file in 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 1-4 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 1-2 (page 25). 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 UINib and NSNib

The `UINib` (iOS) and `NSNib` (OS X) classes provide better performance in situations where you want to create multiple copies of a nib file's contents. The normal nib-loading process involves reading the nib file from disk and then instantiating the objects it contains. However, with the `UINib` and `NSNib` classes, the nib file is read from disk once and the contents are stored in memory. Because they are in memory, creating successive sets of objects takes less time because it does not require accessing the disk.

Using the `UINib` and `NSNib` classes is always a two-step process. First, you create an instance of the class and initialize 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 receive a new set of top-level objects.

Listing 1-5 shows one way to load the contents of a nib file using the `NSNib` class in OS X. 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 1-5 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 iOS, it is possible to create nib files that include placeholder 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, because they were created programmatically or loaded from a different nib file.

Note Custom placeholder objects (other than File's Owner) are not supported in OS X nib files.

Each placeholder object you add to a nib file must have a unique name. To assign a name to an object, select the object in Xcode and open the inspector window. The Attributes pane of the inspector contains a Name field, which you use to specify the name for your placeholder 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 placeholder 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 placeholder objects. The dictionary must contain the `UINibExternalObjects` key whose value is another dictionary containing the name and object for each placeholder replacement.

Listing 1-6 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 placeholder (with the name "AppDelegate") in the main nib file to represent that object. The proxies dictionary stores the placeholder object information and the options dictionary wraps that dictionary.

Listing 1-6 Replacing placeholder 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:UINibExternalObjects];

    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 an 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 placeholder 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.

After creating the connection, you need to implement the action method in your custom class. 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—OS X and iOS 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 2-1 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 2-1 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 33).

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 33). 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 OS X and iOS. 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 additional information about how to use these macros, see "Loading String Resources Into Your Code" (page 37).

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 2-2 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 string on the left side of each equal sign represents the key, and the string on 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 2-2 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 2-3 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 2-3 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 Non-localizable 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 2-1 lists the types of information you can specify for these routines and describes how that information is used by the `genstrings` tool.

Table 2-1 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.)

Parameter	Description
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.)
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 internationalization 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 information about the syntax of the preceding macros, 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. Listing 2-4 shows several string resources that use basic formatting characters:

Listing 2-4 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 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 OS X and iOS 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 Xcode, 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 a nib file 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 a nib file, all you have to do is add them to your Xcode project; Xcode then lists them in the library pane. When you make a connection to a given resource file, Xcode 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). In OS X you can retrieve cached sound resources using the `soundNamed:` method of `NSSound`.

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. OS X and iOS 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 OS X using the AppKit framework use the `NSImage` object to load images and draw them. Applications built for iOS 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 the image data.

Listing 3-1 shows how to load an image resource using the `NSImage` class in OS X. 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 iOS, all you would need to do is change references of `NSImage` to `UIImage`.

Listing 3-1 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 iOS and OS X. Image source objects are available only in 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 3-2 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 3-2 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, imageName, 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* (OS X) or *Core Graphics Framework Reference* (iOS).

Specifying High-Resolution Images in iOS

An iOS app should include high-resolution versions of its image resources. When the app is run on a device that has a high-resolution screen, high-resolution images provide extra detail and look better because they do not need to be scaled to fit the space. You provide high-resolution images for each image resource in your application bundle, including icons and launch images.

To specify a high-resolution version of an image, create a version whose width and height (measured in pixels) are twice that of the original. You use the extra pixels in the image to provide additional detail. When saving the image, use the same base name but include the string `@2x` between the base filename and the filename extension. For example, if you have an image named `MyImage.png`, the name of the high-resolution version would be `MyImage@2x.png`. Put the high-resolution and original versions of your image in the same location in your application bundle.

The bundle- and image-loading routines automatically look for image files with the `@2x` string when the underlying device has a high-resolution screen. If you combine the `@2x` string with other modifiers, the `@2x` string should come before any device modifiers but after all other modifiers, such as launch orientation or URL scheme modifiers. For example:

- `MyImage.png` - Default version of an image resource.
- `MyImage@2x.png` - High-resolution version of an image resource for devices with Retina displays.
- `MyImage~iphone.png` - Version of an image for iPhone and iPod touch.
- `MyImage@2x~iphone.png` - High-resolution version of an image for iPhone and iPod touch devices with Retina displays.

When you want to load an image, do not include the `@2x` or any device modifiers when specifying the image name in your code. For example, if your application bundle included the image files from the preceding list, you would ask for an image named `MyImage.png`. The system automatically determines which version of the image is most appropriate and loads it. Similarly, when using or drawing that image, you do not have to know whether it is the original resolution or high-resolution version. The image-drawing routines automatically adjust based on the image that was loaded. However, if you still want to know whether an image is the original or high-resolution version, you can check its scale factor. If the image is the high-resolution version, its scale factor is set to a value other than `1.0`.

For more information about how to support high-resolution devices, see "Supporting High-Resolution Screens".

Data Resource Files

Separating your application's data from its code can make it easier to modify your application later. If you store the configuration data for your application in resource files, you can change that configuration without having to recompile your application. Data resource files can be used to store any type of information. The following sections highlight some of the data resource types supported by iOS and OS X.

Property List Files

Property list files are a way to store custom configuration data outside of your application code. OS X and iOS 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 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* .

OS X Data Resource Files

Table 4-1 lists some additional resource file types that are supported in Mac apps.

Table 4-1 Other resource types

Resource Type	Description
AppleScript files	In 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 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*.

Document Revision History

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

Date	Notes
2012-06-11	Modified discussion of high-resolution image resources to include all Retina displays.
2011-10-12	Updated for ARC and iOS 5.
2010-09-15	Corrected information about how you specify high-resolution image resource filenames.
2010-05-25	Updated references to the Apple developer website.
2009-01-06	Added information about KVO notifications during nib loading.
2008-06-26	Updated for iOS.
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.

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